NETWARS Model Development Guide

Version 3.0

August 16, 2007

Prepared by:

Defense Information System Agency
NETWARS Program Management Office
5600 Columbia Pike,
Falls Church, VA 22041-2717

Prepared for:

Defense Contracting Command–
Washington
5200 Army Pentagon
Washington, DC 20310-5200

TABLE OF CONTENTS

1 EXECUTIVE OVERVIEW 1-1
1.1 PURPOSE OF THIS DOCUMENT 1-1
1.2 BENEFITS OF MAKING A NETWARS-COMPLIANT MODEL 1-1
1.2.1 Leveraging a Standard Modeling Framework 1-1
1.2.2 Use of Full NETWARS Functionality 1-2
1.3 MODELING BASICS 1-2
1.3.1 Defining the Purpose 1-3
1.3.2 Determining Model Requirements 1-3
1.3.3 Surveying Existing Models 1-4
1.3.4 Developing the Model 1-4
1.4 HOW TO USE THIS DOCUMENT 1-5
2 TECHNICAL OVERVIEW 2-1
2.1 INTRODUCTION TO NETWARS MODELS 2-2
2.1.1 Goals of Model Development 2-2
2.1.2 NETWARS Application Architecture 2-3
2.1.3 NETWARS/OPNET Model Hierarchy 2-8
2.2 MODEL DEVELOPMENT LIFE CYCLE 2-16
2.2.1 Model Development Roles and Responsibilities 2-16
2.2.2 Model Development Activities 2-17
3 NETWARS MODEL DEVELOPMENT 3-1
3.1 TRAFFIC MODEL DEVELOPMENT PROCESS 3-1
3.1.1 Development Approach 3-1
3.1.2 ACE Traffic Model 3-2
3.1.3 ACE Whiteboard Traffic Model 3-2
3.1.4 ACE and ACE Whiteboard Traffic Model Concerns 3-4
3.1.5 IER Text File 3-4
3.1.6 Traffic Model Deployment 3-6
3.2 COMMUNICATIONS DEVICE AND PROCESS MODEL DEVELOPMENT PROCESS 3-8
3.2.1 Development Approaches 3-8
3.2.2 Modifying the Existing OPNET Model to Be NETWARS Compatible 3-9
3.2.3 Surrogating From the Existing NETWARS Model 3-9
3.2.4 Developing a New Model 3-9
3.3 MODEL INTEROPERABILITY ISSUES 3-11
3.3.1 Compatibility Issues 3-11
3.3.2 Interfacing Issues 3-13
3.3.3 Communication Aspects 3-15
3.3.4 Self-Description Issues 3-17
3.3.5 Versioning Issues 3-18
3.4 NETWARS COMPLIANCE REQUIREMENTS 3-20
3.4.1 Compliance for OE Nodes 3-20
3.4.2 Compliance for Models for Non-Discrete Simulation (Capacity Planning) 3-24
3.5 COMPLIANCE FOR END-SYSTEM DEVICES 3-29
3.5.1 Attributes 3-29

3.5.2 Required Modules 3-29
3.5.3 End-System Devices Categories 3-32
3.5.4 Interfaces and Packet Formats 3-35
3.5.5 Initialization 3-35
3.5.6 Interfacing with Other Classes 3-35
3.5.7 Creating Custom Transport Protocols for End-Systems 3-37
3.5.8 Handling Background IERs 3-38
3.5.9 Handling Failure/Recovery 3-38
3.5.10 Collecting Statistics 3-39
3.5.11 NETWARS Standard SE Models 3-40
3.5.12 Example: Constructing a Computer Model 3-41
3.6 COMPLIANCE FOR LAYER 1 NETWORKING EQUIPMENT 3-42
3.6.1 Attributes 3-42
3.6.2 Required Modules 3-42
3.6.3 Interfacing with Devices 3-43
3.6.4 Handling Background Traffic 3-43
3.6.5 Handling Failure/Recovery 3-43
3.6.6 Collecting Statistics 3-44
3.6.7 Example: Constructing an Encryptor Model 3-44
3.7 COMPLIANCE FOR LAYER 2 NETWORKING EQUIPMENT 3-45
3.7.1 Attributes 3-45
3.7.2 Required Modules 3-45
3.7.3 Initialization 3-46
3.7.4 Interfacing with End-System Devices and Networking Equipment 3-47
3.7.5 Supported Protocols 3-47
3.7.6 Handling Background IERs 3-47
3.7.7 Handling Failure/Recovery 3-48
3.7.8 Collecting Statistics 3-48
3.7.9 Example: Constructing a Multi-Service Switch 3-48
3.8 COMPLIANCE FOR LAYER 3 NETWORKING EQUIPMENT 3-49
3.8.1 Attributes 3-49
3.8.2 Required Modules 3-49
3.8.3 Handling Security Classification 3-51
3.8.4 Interfacing with End-System Devices and Networking Equipment 3-51
3.8.5 Supported Protocols 3-52
3.8.6 Creating Custom Routing Protocols for IP 3-52
3.8.7 Handling Background IERs 3-54
3.8.8 Handling Failure/Recovery 3-54
3.8.9 Collecting Statistics 3-55
3.9 COMPLIANCE FOR DEVICES WITH CIRCUIT-SWITCHED TECHNOLOGY 3-56
3.9.1 Attributes 3-56
3.9.2 Initialization 3-56
3.9.3 Routing in Circuit-Switched Devices 3-57
3.9.4 Circuit-Switched Links 3-57
3.9.5 Interfacing with Packet-Switched Networks 3-57
3.9.6 Handling Background IERs 3-59

3.9.7 Handling Failure/Recovery....3-59
3.9.8 Collecting Statistics....3-59
3.10 COMPLIANCE FOR WIRELESS INTERFACES....3-60
3.10.1 Attributes....3-60
3.10.2 Required Modules....3-62
3.10.3 Initialization....3-62
3.10.4 Interfacing with Other Classes....3-62
3.10.5 Interfacing with TIREM....3-63
3.10.6 Restrictions in Building Radio Devices....3-63
3.10.7 Handling Failure/Recovery....3-64
3.10.8 Collecting Statistics....3-64
3.10.9 Building Custom Pipeline Stages....3-64
3.10.10 Satellite Considerations....3-64
3.10.11 NETWARS Standard Geostationary Satellite Communications System Models....3-65
3.10.12 Generic Satellite Device Model (for Bent Pipe Links)....3-66
3.10.13 Generic Satellite Ground Terminal Device Model (for Bent Pipe Links)3-66
3.10.14 TSSP Satellite Terminal Device Model....3-66
3.10.15 Broadcast Radio Considerations....3-66
3.11 COMPLIANCE FOR LINK MODELS....3-68
3.11.1 Attributes....3-68
3.11.2 Building Custom Pipeline Stages....3-69
3.11.3 Handling Background Routed Traffic....3-70
3.11.4 Handling Failure/Recovery....3-70
3.11.5 Building Simplex Links, Buses, and Bus Taps....3-70
3.11.6 Collecting Statistics....3-70
3.11.7 Documentation....3-71
3.12 COMPLIANCE FOR UTILITY NODES....3-72
3.12.1 Attributes....3-72
3.12.2 Required Modules....3-72
3.12.3 Interfacing with Other Classes....3-72
3.12.4 Interfacing with the Scenario Builder GUI....3-72

4 EXAMPLES.... 4-1

4.1 TRAFFIC MODEL EXAMPLE.... 4-2
4.2 ROUTING PROTOCOL EXAMPLE.... 4-4
4.2.1 High-Level Design.... 4-4
4.2.2 Interfacing with the IP Discussion.... 4-6
4.2.3 Notes....4-12
4.3 WIRED END DEVICE EXAMPLE....4-13
4.3.1 Problem Statement....4-13
4.3.2 High-Level Design....4-13
4.3.3 Detailed Design: Event Response Table....4-14
4.3.4 Implementation.... 4-5
4.4 WIRED END DEVICE EXAMPLE 2....4-12
4.4.1 Overview....4-12
4.4.2 Steps....4-12

4.4.3 Process Model: SE 4-15
4.4.4 Statistics 4-16
4.5 LAYER 1 DEVICE EXAMPLE: BULK ENCRYPTOR 4-17
4.5.1 Overview 4-17
4.5.2 Steps 4-17
4.5.3 Process Model 4-18
4.6 LAYER 2 DEVICE EXAMPLE: MULTI-SERVICE SWITCH 4-20
4.6.1 Overview 4-20
4.6.2 Steps 4-20
4.6.3 Process Models: Voice Dispatch and Voice Over ATM 4-21
4.7 LAYER 3 DEVICE EXAMPLE: CUSTOM ROUTER 4-23
4.7.1 Overview 4-23
4.7.2 Steps 4-23
4.7.3 Process Model: Custom Routing Protocol 4-26
4.8 CIRCUIT-SWITCHED DEVICE EXAMPLE: END SYSTEM 4-27
4.8.1 Overview 4-27
4.8.2 Steps 4-27
4.8.3 Process Model: se 4-28
4.9 WIRELESS DEVICE EXAMPLE 4-31
4.9.1 Overview 4-31
4.9.2 Steps 4-31
4.9.3 SE Process Model 4-33
4.10 WIRELESS DEVICE EXAMPLE 2 4-34
4.10.1 Problem Statement 4-34
4.10.2 High-Level Design 4-34
4.10.3 fwd module: Detailed Design 4-35
4.10.4 mac Module 4-38
4.10.5 se Module 4-39
4.10.6 Addressing and Other Issues 4-44
4.10.7 Optimization and Efficiency Considerations 4-44
4.11 SATELLITE TERMINAL GENERIC EXAMPLE 4-45
4.11.1 Node Model Contents 4-45
4.11.2 Core Self-Description Attributes 4-45
4.11.3 Additional Attributes 4-45
4.11.4 Antenna Aim Process 4-47
4.11.5 Key Code Snippets from Antenna Aim Process 4-47
4.12 SATELLITE TERMINAL WITH TSSP EXAMPLE 4-49
4.12.1 Overview 4-49
4.12.2 Node Model Contents 4-49
4.12.3 Core Self-Description Attributes 4-50
4.12.4 Additional Attributes 4-50
4.12.5 Node Model Specific Configuration 4-51
4.12.6 TSSP Process 4-55
4.12.7 Key Code Snippets from TSSP Process 4-57
4.13 SATELLITE GENERIC EXAMPLE 4-60
4.13.1 Overview 4-60

4.13.2 Node Model Contents....4-60
4.13.3 Additional Attributes....4-60
4.13.4 Satellite Switch Process....4-63
4.14 LINK MODEL EXAMPLE....4-66
4.14.1 Overview....4-66
4.14.2 Steps....4-66
4.14.3 Pipeline Stage: txdel....4-66
4.15 OE NODE EXAMPLE....4-68
4.15.1 Overview....4-68
4.15.2 Steps....4-68
4.15.3 Process Model....4-68
4.16 UTILITY NODE EXAMPLE....4-71
4.16.1 Overview....4-71
4.16.2 Details....4-71
4.16.3 Process Model....4-71
4.17 CONVERTING A DEVICE MODEL FROM THE OPNET STANDARD MODEL LIBRARY....4-73
4.17.1 Overview....4-73
4.17.2 Details....4-73
4.18 CP MODEL EXAMPLE....4-77
4.18.1 Overview....4-77
4.18.2 CP Implementation....4-77

5 VERIFICATION AND VALIDATION....5-1

5.1 MODEL FUNCTIONAL V&V....5-2
5.1.1 Objectives....5-2
5.1.2 Steps....5-2
5.2 NETWARS COMPLIANCE V&V....5-4
5.2.1 NETWARS Model Development Checklist....5-4
5.2.2 NETWARS Static Testing....5-4
5.2.3 NETWARS Equipment String....5-5
5.2.4 Capacity Planner....5-6
5.2.5 DoD/Joint VV&A Documentation Tool (DVDT/JVDT)....5-15

APPENDIX A: ACRONYMS....A-1

APPENDIX B: GLOSSARY....B-1

APPENDIX C: ENUMERATED VALUES....C-1

APPENDIX D: PACKET FORMATS....D-1

APPENDIX E: INTERFACES AND PACKET FORMATS....E-1

APPENDIX F: INTERFACE CONTROL INFORMATION (ICI) FORMATS....F-1

APPENDIX G: CONSTANTS....G-1

APPENDIX H: OTHER FILE FORMATS....H-1

APPENDIX I: MEASURES OF PERFORMANCE IN NETWARS....I-1

APPENDIX J: NODE MODEL DOCUMENTATION....J-1

APPENDIX K: MODEL NAMING CONVENTIONS K-1
APPENDIX L: NETWARS SIMULATION API AND HELPER FUNCTIONS L-1
APPENDIX M: ATTRIBUTE TYPE DEFINITIONS M-1
APPENDIX N: EXAMPLES OF NETWARS MODELS N-1
APPENDIX O: NETWARS DOCUMENTATION SET O-1
APPENDIX P: CREATING MODEL REPOSITORIES IN NETWARS P-1
APPENDIX Q: TROUBLESHOOTING NETWARS SIMULATION Q-1
APPENDIX R: FREQUENTLY ASKED QUESTIONS R-1
APPENDIX S: MIGRATION FROM EARLIER OPNET VERSIONS S-1
APPENDIX T: SUPPORTED CLASSIFICATION VALUES T-1
APPENDIX U: SELF-DESCRIPTION GUIDELINES U-1
APPENDIX V: IP AUTO ADDRESSING IN CUSTOM MODELS V-1
APPENDIX W: REFERENCES W-1
APPENDIX X: NETWARS MODEL DEVELOPMENT GUIDE CHECKLIST X-1

LIST OF FIGURES

Figure 2-1: Repeatable Process..2-2
Figure 2-2: Model Repeatable Process..2-3
Figure 2-3: NETWARS Architecture..2-4
Figure 2-4: NETWARS Scenario-Network-Level Model 2-5
Figure 2-5: Editing Device Attributes..2-6
Figure 2-6: Statistics Available in DES 2-7
Figure 2-7: NETWARS/OPNET Model Hierarchy..2-9
Figure 2-8: Editing NETWARS Cisco 2514 Router Model..2-11
Figure 2-9: Process Models Within SINCGARS Device Model................................2-12
Figure 2-10: Process Model Editor..2-13
Figure 2-11: Editing C Code in Process Model Editor 2-14
Figure 2-12: Receive Pipeline Stages 2-15
Figure 2-13: NETWARS Model Development Life Cycle 2-16
Figure 3-1: ACE Whiteboard Screen Capture...3-3
Figure 3-2: ACE Whiteboard Python Logic..3-3
Figure 3-3: IER Text File Sample..3-4
Figure 3-4: ACE and ACE Whiteboard Traffic Model Deployment GUI.....................3-6
Figure 3-5: IER Import Manual..3-7
Figure 3-6: High-Level Model Development Process 3-8
Figure 3-7: Protocol Dependency (e.g., Ethernet Computer Model)...........................3-12
Figure 3-8: Module-Wide Memory (e.g., Ethernet Computer Model) 3-14
Figure 3-9: Default Interrupt Handling 3-17
Figure 3-10: Self-Description Port Objects...3-18
Figure 3-11: CP Layers 3-26
Figure 3-12: Ethernet End-System Device-Node Model 3-31
Figure 3-13: End-System Device with Frame Relay MAC Technology-Node Model..3-32
Figure 3-14: Valid End-System to End-System Connection.......................................3-33
Figure 3-15: Circuit-Switched End-System Device-Node Model..............................3-33
Figure 3-16: Circuit-Switched End-System Device-Voice Applications and IERs 3-34
Figure 3-17: End-System Device Generating Voice and Data Traffic-Node Model.....3-35
Figure 3-18: Remote Interrupt from OE to SE 3-36
Figure 3-19: oe_threads Process Model 3-40
Figure 3-20: Layer 1 Networking Equipment-Node Model...3-42
Figure 3-21: Layer 2 Networking Equipment-Node Model...3-46
Figure 3-22: Layer 3 Networking Equipment-Node Model...3-50
Figure 3-23: Networks with Different Security Classification Levels 3-51
Figure 3-24: Circuit-Switched and Packet-Switched Network Intercommunication 3-58
Figure 3-25: Radio End-System Device-Node Model 3-62
Figure 3-26: ATM Device Radio Interface 3-63
Figure 3-27: Internal Representation of ATM Device and Intermediate Node.............3-64
Figure 3-28: Channel Table...3-67
Figure 4-1: Time Sequence Diagram 4-2
Figure 4-2: Layer 3 Networking Equipment 4-5
Figure 4-3: IP Routing Parameters Attribute...4-9

Figure 4-4: Interface Information Attribute 4-9
Figure 4-5: Routing Protocol Attribute Properties........ 4-10
Figure 4-6: End-Device Node Model........ 4-14
Figure 4-7: Interfacing Modules of “se” 4-15
Figure 4-8: High-Level Functions of “se_tcp” Module 4-1
Figure 4-9: se_trafgen Process Model........ 4-5
Figure 4-10: Open Connection State........ 4-5
Figure 4-11: Receive Traffic State........ 4-7
Figure 4-12: Process Message State........ 4-8
Figure 4-13: Failure State........ 4-10
Figure 4-14: Ethernet_wkstn_adv-Node Model 4-13
Figure 4-15: Computer-Node Model 4-14
Figure 4-16: Workflow Diagram for SE Process Model........ 4-15
Figure 4-17: Process Model for SE Module in Computer 4-16
Figure 4-18: Code 1-Inform OE of IER Failure, Will Record Statistics 4-16
Figure 4-19: Encryptor-Node Model 4-17
Figure 4-20: Data Flow for Encryptor 4-18
Figure 4-21: Process Model for Encryptor........ 4-19
Figure 4-22: Code 2-Encrypting a Packet 4-19
Figure 4-23: Atm_uni_dest_adv Switch-Node Model........ 4-20
Figure 4-24: Multi-Service Switch-Node Model........ 4-21
Figure 4-25: CS_1005_1s_e_sl_adv Router-Node Model 4-24
Figure 4-26: Router with Custom Routing Protocol-Node Model 4-25
Figure 4-27: Process Model for Custom Routing Protocol........ 4-26
Figure 4-28: Phone-Node Model 4-27
Figure 4-29: Data Flow for Phone 4-29
Figure 4-30: Process Model for SE Module........ 4-30
Figure 4-31: wlan_station_adv-Node Model........ 4-31
Figure 4-32: Radio SE model-Node Model........ 4-32
Figure 4-33. Radio End Device Node Model........ 4-34
Figure 4-34: fwd Module Process Model........ 4-36
Figure 4-35: SE Module Interfaces........ 4-39
Figure 4-36: Radio SE Process Model........ 4-40
Figure 4-37: Gen_Call State........ 4-41
Figure 4-38: Proc_Pk State........ 4-43
Figure 4-39: Generic Satellite Terminal........ 4-45
Figure 4-40: Antenna Aim Process........ 4-47
Figure 4-41: TSSP Satellite Terminal 4-50
Figure 4-42: Configuration-TSSP Nodal Terminals........ 4-52
Figure 4-43: Each Row Corresponding to deMUX Group 4-52
Figure 4-44: Each Row Corresponding to Input Port Group........ 4-53
Figure 4-45: TSSP Process Model........ 4-55
Figure 4-46: Uplink and Downlink Tables........ 4-62
Figure 4-47: Satellite Switch Process Model 4-63
Figure 4-48: Code 3-Adding Signaling Overhead to Transmission Delay 4-66
Figure 4-49: Functions of OE Process Model 4-69

Figure 4-50: OE Process Model 4-70
Figure 4-51: Promina Configuration Utility Node-Node Model 4-71
Figure 4-52: Promina Configuration Object-Process Model 4-72
Figure 4-53: Promina Configuration Object-Sample Code 4-72
Figure 4-54: Sample Node Model 4-73
Figure 4-55: Selecting "Computer" for equipment_type 4-74
Figure 4-56: Adding se_tcp and se_udp 4-74
Figure 4-57: Adding *net_id* Extended Attribute 4-75
Figure 4-58: Equipment type attribute location 4-77
Figure 4-59: Interface Class and Machine type attribute locations 4-78
Figure 4-60: Interface Class attribute 4-78
Figure 4-61: Machine type attribute 4-79
Figure 5-1: M&S Overall Problem Solving Process 5-3
Figure 5-2: Initiate a static test 5-6
Figure 5-3: Execute a static test for the nw_ethernet_wkstn device 5-7
Figure 5-4: Select component class for static test 5-7
Figure 5-5: Select model options for static test 5-7
Figure 5-6: Select protocols for static test 5-8
Figure 5-7: Select report file name and confirm answers for static test 5-8
Figure 5-8: Summary and completion message for static test 5-9
Figure 5-9: Static test report 5-10
Figure 5-10: Static test report 2 5-11
Figure 5-11: Static test report 3 5-12
Figure 5-12: Static test report 4 5-13
Figure 5-13: Static test report 5 5-14
Figure D-1: Packet Format Files D-1
Figure D-2: Open Packet File D-2
Figure D-3: Packet Format Layout D-2
Figure D-4: Packet Format Attribute Editing D-3
Figure F-1: ICI Format Files F-1
Figure F-2: Open ICI Format F-2
Figure F-3: ICI Format Attributes F-2
Figure L-1: API Files L-1
Figure L-2: Open API File L-2
Figure L-3: oe_stat_support API L-3
Figure N-1: List of Node Models N-1
Figure N-2: Open NETWARS Model N-2
Figure O-1: NETWARS Documentation Set O-1
Figure U-1: Self-Description Port Objects U-1
Figure V-1: Node Model Contents V-2
Figure V-2: Custom Device Attribute Values in OPFAC Soldier 1 .... V-2

LIST OF TABLES

Table 2-1: Model Types and Descriptions 2-9
Table 2-2: Model Development Activities 2-17
Table 3-1: Attributes for OE Node 3-20
Table 3-2: Statistics Collected by OE Node 3-24
Table 3-3: Properties to Determine CP Layer 3-26
Table 3-4: NETWARS Attributes for End-System Device 3-29
Table 3-5: Higher Layer Modules for End-System Device 3-30
Table 3-6: Lower Layer Modules for End-System Device 3-30
Table 3-7: Interface Modules for End-System Device 3-31
Table 3-8: Statistics Information Transferred by End-System Device to OE 3-40
Table 3-9: NETWARS Standard SE Process Models 3-41
Table 3-10: Attributes for Layer 1 Networking Equipment 3-42
Table 3-11: Attributes for Layer 2 Networking Equipment 3-45
Table 3-12: Modules Needed for Various Layer 2 Protocols 3-45
Table 3-13: Modules Needed by Multi-Service Switch 3-46
Table 3-14: Attributes for Layer 3 Networking Equipment 3-49
Table 3-15: Higher Layer Modules for Layer 3 Networking Equipment 3-49
Table 3-16: Required Modules for Various Interface Technologies 3-50
Table 3-17: Interface Modules for Layer 3 Networking Equipment 3-51
Table 3-18: Required Attributes-Circuit-Switched End-System Device 3-56
Table 3-19: Required Attributes-Circuit-Switched Layer 2 Networking Equipment 3-56
Table 3-20: Additional Attributes for Radio Devices 3-60
Table 3-21: Pipeline Stage Attributes on Radio Transmitter 3-61
Table 3-22: Pipeline Stage Attributes on Radio Receiver 3-61
Table 3-23: Restrictions in Building Radio Devices 3-63
Table 3-24: Required Satellite Device Attributes for Moving Orbits 3-65
Table 3-25: Radio Transceiver Pipeline Stages 3-65
Table 3-26: Required Attributes on Link Model 3-68
Table 3-27: Required Attributes for Utility Nodes 3-72
Table 3-28: Optional Attributes for Utility Nodes 3-72
Table 4-1: Available IP Common Route Table API Functions 4-10
Table 4-2: Event Description Table 4-2
Table 4-3: Event Communication Mechanisms 4-2
Table 4-4: State Description Table 4-3
Table 4-5: Event Feasibility Table 4-3
Table 4-6: Event Response Table 4-4
Table 4-7: End-System-Model Attributes 4-14
Table 4-8. Circuit-Switched End-System Device-Model Attributes 4-28
Table 4-9. Radio End-System Device-Model Attributes 4-32
Table 4-10: Event Response Table for “fwd” Process 4-36
Table 4-11: Event Response Table for Radio SE Module 4-39
Table 4-12: Event Response Table for Radio SE Module 4-53
Table 4-13: Events of TSSP Process Model 4-55
Table 4-14: Events of Satellite Switch Process Model 4-63

Table 4-15: Utility Node-Model Attributes 4-71
Table C-1: Attributes for Enumerated Data Types C-1
Table D-1: Packet Formats D-4
Table E-1: Interfaces and Packet Formats E-1
Table F-1: Interfaces and Packet Formats F-3
Table G-1: Constants G-1
Table G-2: Typed File Attribute G-2
Table H-1: Other File Formats H-1
Table I-1: MOPs Reported by OE I-1
Table I-2: Statistics Groups I-2
Table I-3: Modules That Write OV Statistics I-3
Table J-1: Wired Interface Specifications J-2
Table J-2: Radio Device Interface Specifications J-2
Table J-3: Process Models J-2
Table J-4: External Files Needed J-3
Table L-1: Example of API Function Table L-4
Table L-2: NETWARS APIs and Locations L-4
Table N-1: List of NETWARS Models (Alphabetic) N-3
Table R-1: FAQs R-1
Table U-1: Packet Formats to Interface Types U-2
Table U-2: Supporting Technologies per Port Category U-3
Table X-1: NETWARS Model Development Guide Checklist X-1

1 EXECUTIVE OVERVIEW

1.1 PURPOSE OF THIS DOCUMENT

The purpose of the *NETWARS Model Development Guide* is to provide modeling guidelines and standards for creating communications device and traffic models that are interoperable with the Network Warfare Simulation (NETWARS) System and model suite. The *NETWARS Model Development Guide* provides the standards for creating NETWARS communication device and traffic models and provides the instructions for modifying existing OPNET commercial off-the-shelf (COTS) models to adhere to these standards.

This document provides engineers with the information necessary to develop device and traffic models that interoperate with existing NETWARS and OPNET COTS models within the NETWARS modeling framework. Any device model written to these standards will integrate seamlessly with the existing model libraries and will be able to take advantage of the benefits that the NETWARS modeling environment has to offer.

1.2 BENEFITS OF MAKING A NETWARS-COMPLIANT MODEL

NETWARS is a communications system simulation tool. Its primary purpose is to evaluate strategic, operational, and tactical communications networks before they are developed, deployed, or modified in order to provide early feedback to decision makers. NETWARS leverages COTS software that models commercial communications networks and adds military-specific device, protocol, and application models to provide a complete environment for modeling military communications networks.

The following sections detail the benefits that this common simulation framework provides over traditional, stovepipe methods.

1.2.1 Leveraging a Standard Modeling Framework

Many modeling efforts throughout the Department of Defense (DoD) have been undertaken in a standalone manner, with little attempt being made to reuse models or integrate with existing work. Part of the reason for this is a lack of standardization within the modeling community, which makes it difficult to reuse existing component parts. The use of a common simulation framework such as NETWARS imparts some standardization to these modeling efforts and promotes model reuse.

One of the benefits of a common framework is the guarantee that all models built to that specification will work together in an integrated fashion. This increases efficiency and drives down costs in multiple ways:

- **Eliminates Redundant Modeling Efforts.** Engineers embarking on a new modeling project are able to reuse existing device models, knowing that they are interoperable with new models built to the same specification. This reduces or eliminates the need to produce multiple models of the same devices to work in varying simulation environments, thus reducing program cost and overall cost to the Government.

- **Provides a Baseline for Comparative Analysis.** A repeatable set of inputs and constraints is central to an effective modeling exercise. By using a standardized set of models, engineers can control the variables that go into a simulation and ensure that any measured differences in results are due to intentional changes in inputs. This ensures valid comparisons of devices or other variables, which is especially valuable when performing comparisons of new technologies from multiple vendors.

1.2.2 Use of Full NETWARS Functionality

Models built according to the guidelines outlined in this *NETWARS Model Development Guide* are interoperable not only with other models developed using these standards but also with the majority of the OPNET COTS device models. In this way, the models are able to take advantage of many years of commercial development and model testing by leveraging the OPNET COTS Standard and Specialized model library. This library includes intrinsic capabilities for common communication modeling issues such as traffic generation, dynamic routing, and connection establishment. The library also contains a wealth of standard protocol models such as Ethernet, Asynchronous Transfer Mode (ATM), frame relay, Fiber Distributed Data Interface (FDDI), token ring, Digital Subscriber Line (DSL), Transmission Control Protocol (TCP)/Internet Protocol (IP), Routing Information Protocol (RIP), Open Shortest Pathway Forwarding (OSPF), Extended Interior Gateway Routing Protocol (EIGRP), Interior Gateway Routing Protocol (IGRP), Border Gateway Protocol (BGP), File Transfer Protocol (FTP), and Hypertext Transport Protocol (HTTP); a host of wireless protocols such as Wireless Fidelity (WiFi) and Worldwide Interoperability for Microwave Access (WiMax); and a family of Mobile Ad Hoc Network (MANET) protocols such as Optimized Link State Routing (OLSR), Ad Hoc On-Demand Distance Vector (AODV), and Temporally Oriented Routing Algorithm (TORA).

In addition, NETWARS provides access to customized capabilities that do not exist in COTS products. These capabilities include a large military-specific device library, customized reporting, and specialized traffic-handling techniques. Some of the available models are shown in the list below. A full, up-to-date list can be found in Appendix N.

Device models available in NETWARS include but are not limited to the following:

- Prominas (multiple configurations)
- Tactical radio systems (Single-Channel Ground and Airborne Radio System [SINCGARS], Enhanced Position Location Reporting System [EPLRS], Link-11, Link-16, etc.)
- Encryptors (KIV and KG-series)
- Satellites and earth terminals (AN/TSC series, Standardized Tactical Entry Point (STEP), Teleport, Global Broadcast Service (GBS)
- Tactical voice and circuit switches (AN/TTC series, Switch Multiplexer Unit (SMU), Digital Non-Secure Voice Terminal (DNVT), Secure Telephone Units III (STU-III).

1.3 MODELING BASICS

NETWARS is a communications system simulation tool made up of two primary simulation technologies — Discrete Event Simulation (DES) and Capacity Planner (CP). CP is a

customized analytic approach, implemented specifically for NETWARS. By convention, models developed for the NETWARS environment support both modeling technologies.

DES provides an explicit, packet-by-packet simulation of network traffic for the system being modeled. It is extremely detailed and can provide results at a high level, such as time-varying link utilizations, all the way down to very granular measurements such as queue lengths on individual routers. Additionally, because NETWARS ships with the full source code to both the COTS and NETWARS model libraries, model developers can extend the models to add their own statistics or other custom behaviors.

CP provides a broader look into network behavior. It is primarily used to study steady-state network behavior, and as a result is not suitable for studies such as protocol convergence times. However, due to the nature of the modeling technology it uses, it can run much more quickly than DES. For the appropriate analyses, it will provide results similar to those achieved by DES but at a fraction of the run time.

1.3.1 Defining the Purpose

The first and most critical issue to be addressed when undertaking a modeling project is identifying the reason(s) behind the use of the model. Any modeling project that begins with the thought "I will build a model first and figure out what I want to use it for later" is destined to fail. The best way to determine the purpose of the model is by asking "What *specific* question(s) do I want this model to answer for me?" Following are examples of specific, purpose-driven questions:

- What will be the impact on end-to-end message delays when I replace my existing Media Access Control (MAC) layer with a new implementation?
- Will the new routing protocol "X" be interoperable with other protocols in use on my network? Will I be able to redistribute routes between these networks?

Once these questions have been answered, the features of the device/system to be modeled that are pertinent to the study can be identified. This will then allow the identification of features or behaviors of the device that need to be built into the model.

1.3.2 Determining Model Requirements

To develop a model of a communications device, system, or application, there must be a working knowledge of the features that device or system supports. In the case of a communications device, this includes supported protocols, performance specifications, and any known limitations about, or criteria for, its interactions with other devices. For example, a radio that needs to be part of a slot selection mechanism of a network comprised of one or more radios will have additional interoperability requirements.

Some of this material is readily available in vendor specification sheets or documents issued by standards bodies such as Institute of Electrical and Electronics Engineers (IEEE). Another useful source of material is actual performance data from a Testing and Evaluation (T&E) or production environment. The use of empirical data to validate the behavior of the model can be invaluable.

For example, routing convergence data from a live device can be used to validate a routing protocol whose model is being developed.

It is equally important, however, that the relevance of these behaviors is known. For example, many devices send out periodic messaging information (data packets) to communicate with the rest of the network. This data does not materially impact the device's behavior, and as such, if the amount of this traffic is deemed to be small, it may be ignored or "abstracted away" in the context of the model, simplifying the modeling effort with no significant loss of accuracy.

Similarly, it is often not necessary to know the inner workings of a cryptographic or other processing algorithm, for example, to build a behavioral model of such a device. If the purpose of the study is to measure network capacities, then modeling the overhead capacity incurred as a result of encryption is sufficient. The exact encryption algorithm itself does not need to be modeled.

1.3.3 Surveying Existing Models

Once the model requirements have been identified, the next task is to determine whether an existing model possesses some or all of the needed capabilities. Depending on the output of previous modeling projects, a model may exist that has the necessary functionality and, through configuration and without code modification, can be made to satisfy the specific requirements. This is known as *model surrogation*. Model surrogation is an area where the common modeling framework and modeling standardization proves its worth. A community-wide library of models that function in well-defined, interoperable ways can greatly reduce time and costs associated with model development.

Even when a pre-existing model does not serve all of the needs of a new project, in many cases it can be used as a starting point for a new model. The NETWARS environment supports *model derivation*, which is the process of using an existing model as a baseline set of functionalities and adding/modifying just those that are new or different from the baseline set. In this way, improvements to the base (COTS or custom) model will be inherited by the derived model, reducing configuration management (CM) costs.

Finally, even when model derivation is not an appropriate solution, it is normally advantageous to use existing models as a starting point. Models of a similar class (e.g., transport devices, end devices, routers, switches) often provide similar functionality that can be modified through code enhancements to meet the specified need.

All of these examples of model and code re-use are only possible when a set of standards is defined and followed. This document defines that set of standards for the NETWARS environment and helps to determine when each of the above approaches is suitable for a specific project.

1.3.4 Developing the Model

Sometimes there is no alternative but to develop a new model. In such cases, this *NETWARS Model Development Guide* takes on greater importance. There are a number of things that differentiate NETWARS models from OPNET Standard models. A few of the primary

differences are listed below; the rest of this document is devoted to explanations of how to ensure that these differences are accounted for and implemented in such a way that the resulting model is truly interoperable with other models within the NETWARS framework. Full explanations of these differences, and how to interact with them, are provided in Section 3.

Primary differences between NETWARS models and OPNET Standard models include the following:

- **IER Support.** NETWARS provides support for handling Information Exchange Requirements (IER), the doctrinally approved specification of traffic load for communications scenarios. Those devices that are planned as sources or sinks of traffic must be capable of generating and receiving these constructs.
- **CP Support.** NETWARS models must operate in both the DES and CP environment. This analytical simulation technology is custom built to handle the NETWARS circuit-switched modeling construct and to allow capacity planning workflows that include wireless devices.
- **Classification.** NETWARS models support the notion of classification, which enables military network planners to build models of different security enclaves.
- **Interaction with NETWARS Model Suites.** Newly developed NETWARS models must also interact smoothly with the existing device models and technology frameworks that reside within NETWARS. These include Prominas and other circuit-switched devices, broadcast networks, Satellite Communications (SATCOM) devices and terminals, and message-based systems such as Link-16.

NETWARS itself does not provide a model-authoring framework. Models for use in NETWARS are developed using the Modeler development environment, a COTS software package produced by OPNET. This software is not available through the NETWARS program office; to acquire it one must contact OPNET. Prior to beginning NETWARS model development, it is important that the developer is familiar with the following materials:

- OPNET Modeler
- C/C++ development language
- This *NETWARS Model Development Guide*.

There are many resources available to help learn about OPNET Modeler and C/C++. In particular, the OPNET Support Center (http://www.opnet.com/support/home1.html) is an excellent place to obtain a background in using the Modeler framework for model development. Look especially at the "Methodologies and Case Studies" link for more information.

1.4 HOW TO USE THIS DOCUMENT

The remainder of this document covers various aspects of NETWARS model standards and interoperability concerns. Code examples are also presented to emphasize the practical application of the standards described. It may be read as a narrative for an introduction to these topics or used as a reference guide throughout the design and development process.

The sections and their purposes are listed below:

- **Section 1: Executive Overview (this section).** This section provides an executive-level overview of NETWARS model development and the *NETWARS Model Development Guide*.
- **Section 2: Technical Overview.** This section provides an overview of a model development process, including information for the Technical Manager to oversee a model development effort.
- **Section 3: Model Development.** This section provides the technical details of making a model NETWARS compliant.
- **Section 4: Model Development Examples.** This section provides additional examples of model development that go into more detail or cover additional topics.
- **Section 5: Model Validation and Verification.** This section provides detailed technical specifications about model verification and validation (V&V) for all types of NETWARS models.
- **Appendices.** The appendices provide associated references, such as NETWARS Packet Formats, Frequently Asked Questions (FAQ), and a Model Checklist to support model development.

This document should be read by program managers, technical managers, model developers, subject matter experts (SME), and quality assurance engineers (QAE) involved in a modeling project. Recommended sections for each of these audiences are listed below:

- **Program Managers.** Sections 1 and 2
- **Technical Managers.** Sections 1 and 2 and Subsection 3.1 and 3.2
- **Model Developers.** Sections 1 and 2, followed by Subsections 3.1, 3.2, 3.3, and 3.4. This should be followed by Section 5, going back to cover the portions in Sections 3 and 4 that are relevant to the type of device being developed. Finally the developer should return to Section 5 to cover the portions that are relevant to the device being developed.
- **SMEs/QAEs.** Sections 1, 2, and 5 and Subsection 3.2. The purpose of the document is to allow a SME to help with the design and verification of a model.

This document is based on NETWARS 2006-02

2 TECHNICAL OVERVIEW

To understand the details of developing communications device models, one should be familiar with NETWARS and modeling communications systems.

This section is an overview showing what capabilities exist within a device model and how reuse is possible in NETWARS model development. It is not meant to substitute as an instruction manual for OPNET Modeler, which already has significant online documentation and technical support available through OPNET, nor is the *NETWARS Model Development Guide* meant to replace this documentation or to teach modeling in general. Rather, it is intended to provide additional information and guidance to enable the model developer to create models capable of proper interaction with the rest of the NETWARS model library. Such models are termed NETWARS-compliant models.

This section summarizes the following topics:

- The purpose and steps of modeling
- NETWARS software and communications network modeling
- Types of NETWARS models and the OPNET model hierarchy
- Methods for creating NETWARS device models
- The model development process.

2.1 INTRODUCTION TO NETWARS MODELS

2.1.1 Goals of Model Development

The entire modeling enterprise is based on one fundamental assumption, which is much like Newtonian determinism. We must assume that the important processes governing the system to be modeled are repeatable and, more important, obey the laws of nature. A system has inputs (or preconditions) and a process that follows some rules and produces outputs (the post conditions). This high-level view allows engineers to model a system (or process) and predict its performance (see Figure 2-1).

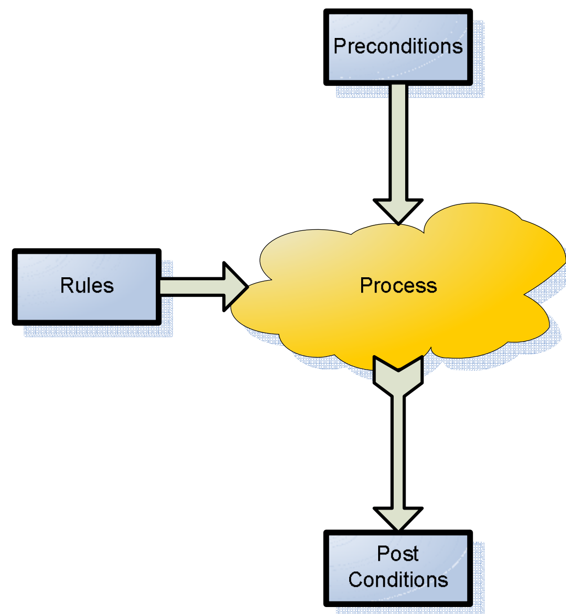


Figure 2-1: Repeatable Process

The modeling discipline involves capturing the rules of a repeatable process, simulating the process, and performing experiments on the simulated system. For example, to simulate the movement of the planets around the Sun, the rules are Newton's laws of motion and gravity. To predict the future position of the planets, a study analyst captures the inputs to the system and runs a simulation. In this case, the inputs are the mass, velocity, and current position of the planets and the Sun. The simulation will then process the inputs according to the rules and produce the outputs (see Figure 2-2).

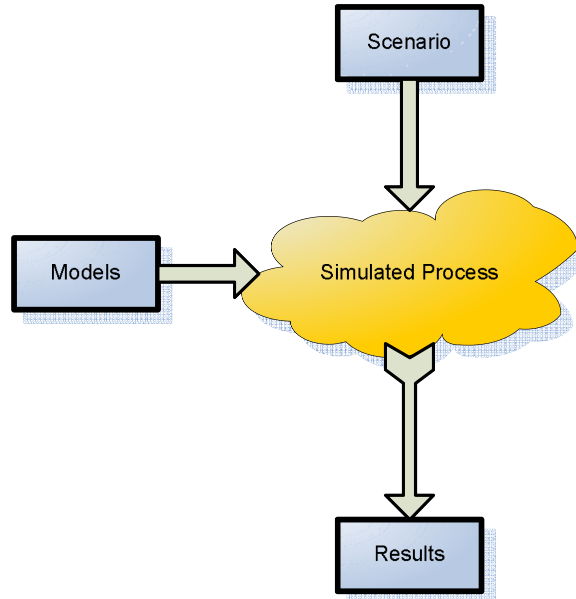


Figure 2-2: Model Repeatable Process

Before reliance can be placed on the results of a model, the model must be validated. To validate a model, the inputs and outputs of the simulation are compared to data collected from real-world observations. Validation is a scientific experiment testing the hypothesis that the model faithfully captures the salient characteristics of the real-world system. Among other things, the experiment measures the accuracy of the model. By measuring the outputs of a real system and comparing them to the outputs of a simulated system, a model tester can determine whether the model can answer the questions it was intended to answer and for what range of inputs the model is valid. Without this validation step, results from a simulation should be interpreted with skepticism.

Modeling and simulation are conceptually simple, but the practice of creating models that correctly answer real-world questions is difficult. This section provides guidance on building NETWARS-compliant communications device models. It describes a process to produce and validate these device models so they can be integrated into the NETWARS simulation environment.

2.1.2 NETWARS Application Architecture

NETWARS is the DoD Joint Communications Modeling and Simulation tool. The NETWARS simulation environment is a government off-the-shelf (GOTS) solution based on OPNET Technologies commercial technology. NETWARS adds five major functions to the OPNET COTS product:

- Military-specific models (tactical radios, Prominas, SATCOM)
- A simple-to-use capacity planning engine
- A simple-to-use analytic simulation engine
- Usability enhancements (wizards, reports, PowerPoint export)
- Collaborative planning workflow for Joint Command, Control, Communications, Computers, and Intelligence (C4I) planning.

The majority of NETWARS users are analysts. NETWARS provides a drag-and-drop graphical user interface (GUI) to assemble a scenario. The scenario is then input to a simulation. After creating a scenario, a NETWARS user can press a button to simulate the scenario. The results of the simulation can be viewed within NETWARS.

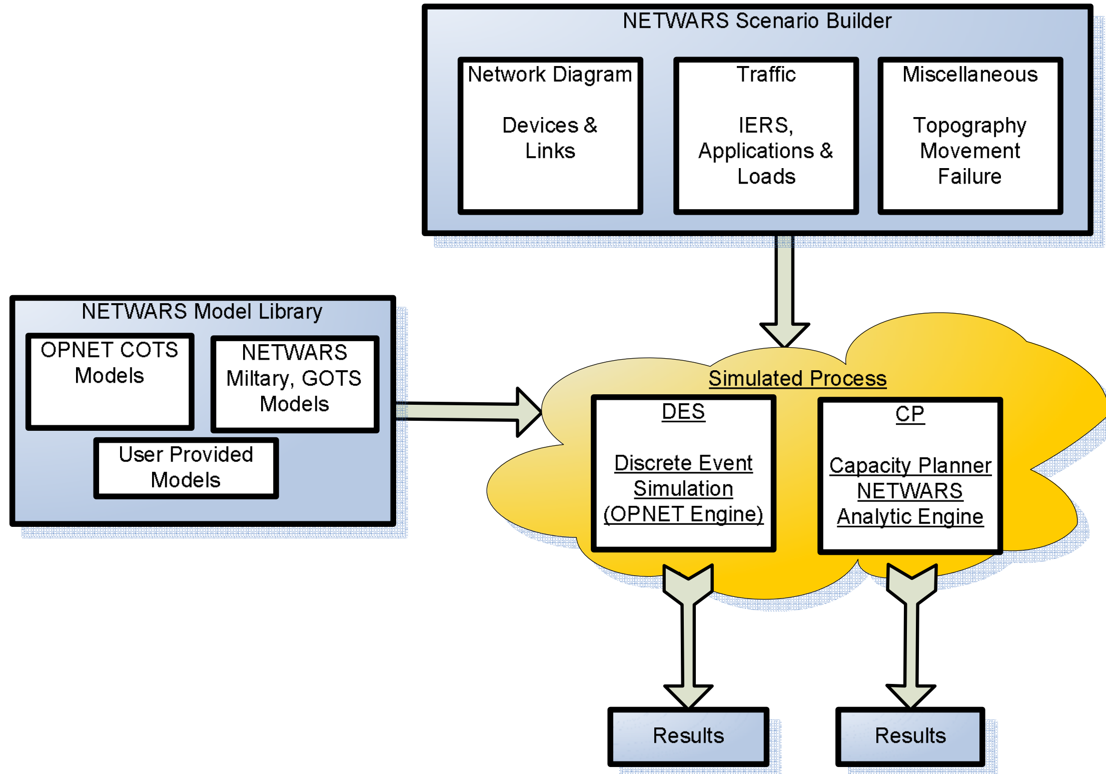


Figure 2-3: NETWARS Architecture

Figure 2-3 illustrates the various NETWARS components and how they fit into the modeling and simulation paradigm. The major components of the NETWARS architecture include the following:

- Scenario Builder
- CP
- DES Engine
- NETWARS model library, including:
  - Device models
  - Process models and other modules
  - Pipeline stages
  - Traffic models.

2.1.2.1 Scenario Builder

The NETWARS Scenario Builder is the most recognizable part of NETWARS. When most users think of NETWARS, they think of the Scenario Builder. Within the Scenario Builder interface, users drag models from the pallet and place them on the workspace. Links are then made

between the devices, and finally traffic is added to the scenario. Figure 2-4 depicts a sample NETWARS scenario.

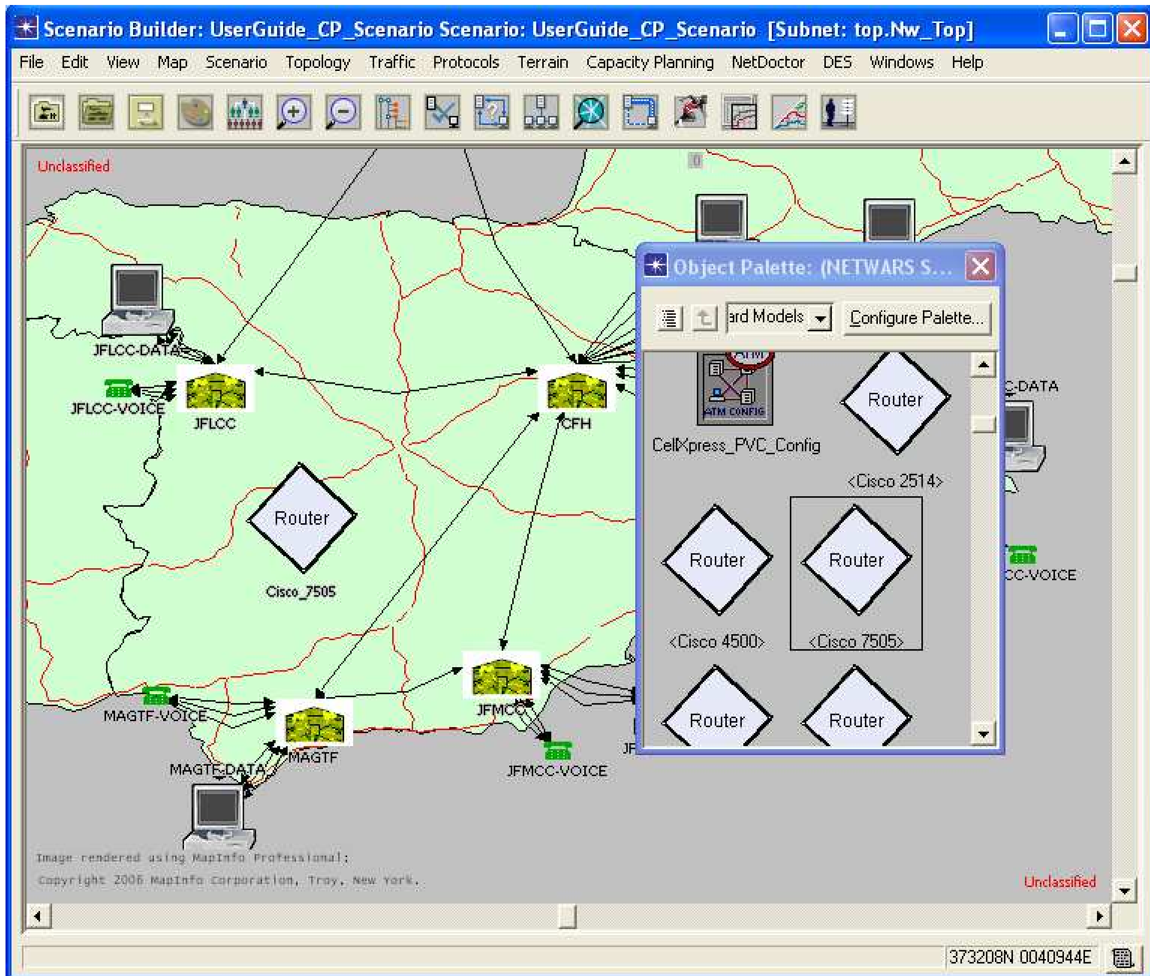


Figure 2-4: NETWARS Scenario-Network-Level Model

The Scenario Builder interface allows users to create a scenario using existing device models. By clicking one of the toolbar buttons, the network can be simulated using either the capacity planner or the DES engine.

This interface also allows editing device attributes. A good example of this is configuring a router. The behavior of a router is highly dependent on its configuration. Figure 2-5 shows some of the detail incorporated into one of the standard NETWARS routers. The list of attributes exposed is defined by the model developer, but the NETWARS user is able to change these values to configure the device for simulation.

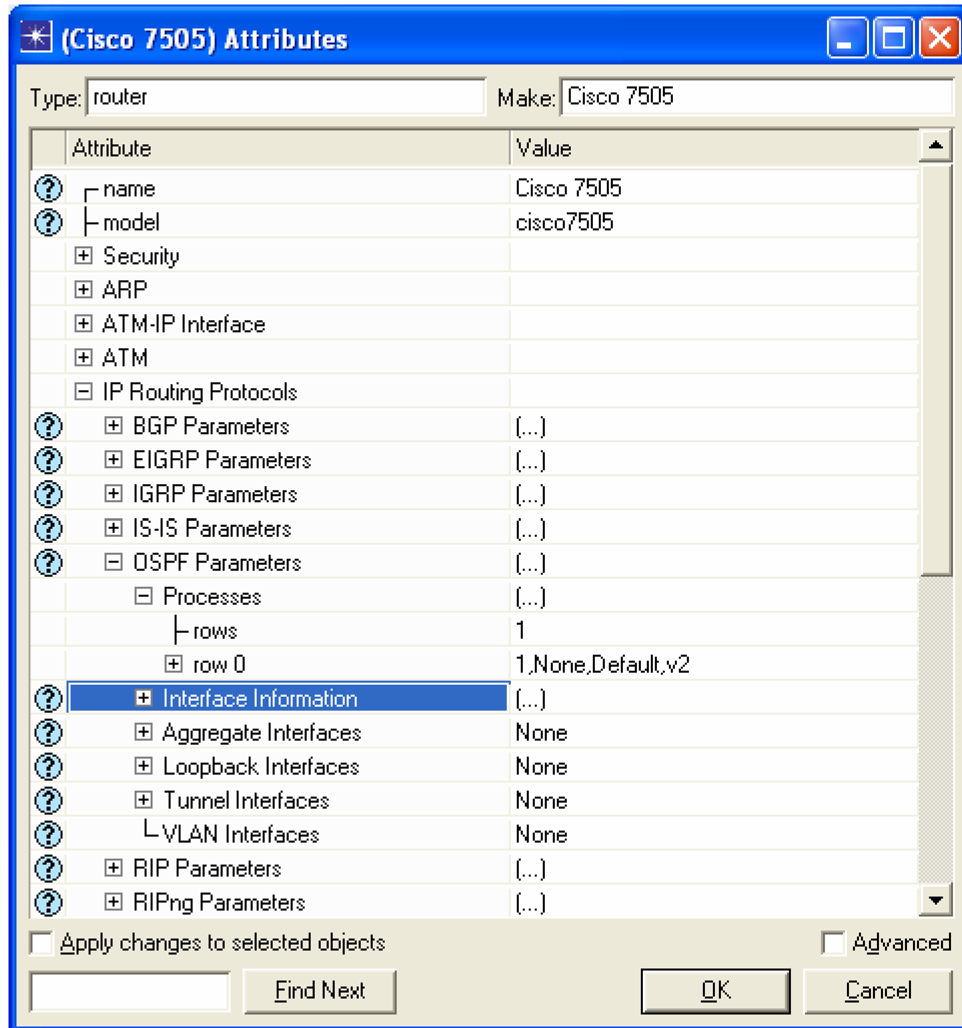


Figure 2-5: Editing Device Attributes

2.1.2.2 Capacity Planner

CP is a NETWARS analytic simulation engine. It routes traffic and calculates link and circuit utilizations. CP is designed to run quickly and be easy to use, and it usually requires little effort to make models work with CP. CP uses only a handful of device attributes and properties. Subsection 3.3 describes in detail how to make models work with CP.

2.1.2.3 Discrete Event Simulation

The DES engine is COTS technology available from OPNET. OPNET Modeler and IT Guru use the same DES engine. DES involves modeling all the individual events in the communications network. This includes every TCP/IP packet sent, each radio packet sent, and numerous signaling packets for voice communications. Although the DES engine is highly optimized, DES takes much longer than CP to simulate the same network. The tradeoff for the longer running times is that a DES simulation will generate more accurate results.

In addition, the results can include low-level measurements, such as end-to-end delay (minimum, maximum, and average), bit error rate, packets sent for each interface on a router, and number of packets dropped. Figure 2-6 shows some of the statistics available for a NETWARS router.

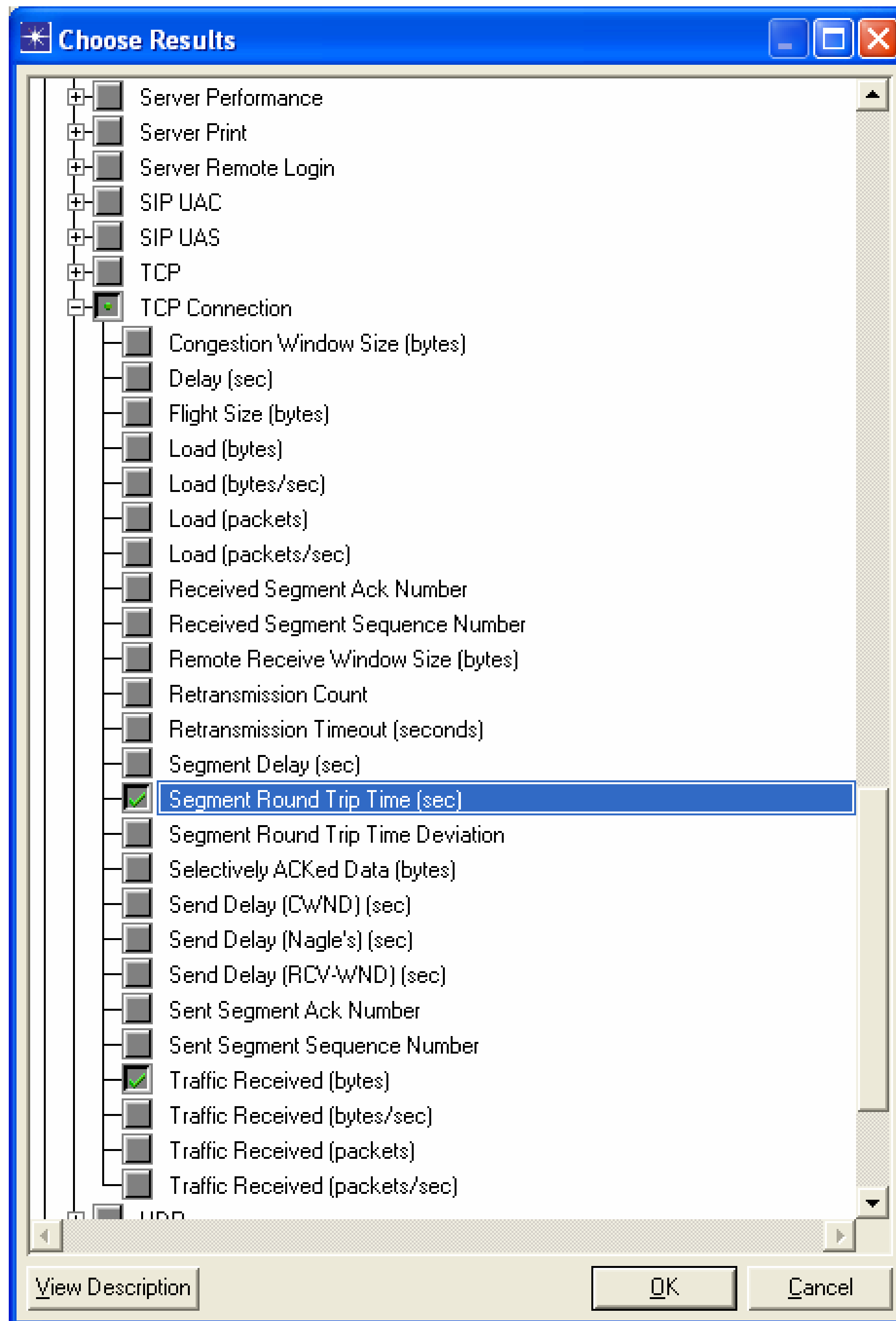


Figure 2-6: Statistics Available in DES

With a user-selectable level of statistics granularity, NETWARS can provide answers to very detailed questions. However, it is important to remember that bad inputs can lead to bad outputs.

Users must validate their scenarios and configurations. Model developers are also expected to validate their own models.

2.1.2.4 NETWARS Model Library

NETWARS is supplied with a wide selection of military and commercial device models. This includes the full OPNET Model Library of commercial network devices and the NETWARS military model library. The military models include:

- Tactical radios
- Encryptors (bulk encryptors and Inline Network Encryptors (INE))
- Multiplexers (including Federal Communication Commission (FCC)-100 and Promina)
- Military phone systems
- Satellite terminals
- Models to process DoD Architecture Framework (DoDAF) traffic: IER

These models include network routing behavior, priority preemption, Radio Frequency (RF) attenuation and propagation effects, and IP quality of service (QoS). Subsection 2.1.3 provides more information on the composition of a NETWARS model.

2.1.3 NETWARS/OPNET Model Hierarchy

NETWARS is built upon OPNET COTS technology, and the DES engine used by NETWARS is the highly optimized OPNET COTS DES engine. This DES engine uses models stored in an OPNET format, and creating new models usually involves using the OPNET Modeler product.

When discussing models in NETWARS, the terminology becomes important because there are many types of models. This section briefly describes the six basic types of models, shown in Figure 2-7. These model types are identical to those used in the OPNET Modeler product. For clarity, some OPNET terminology has been adopted with the exception that NETWARS uses “device” instead of “node”. Most important, NETWARS employs Operational Facility (OPFAC) and Organization (Org) military ideas to create network scenarios.

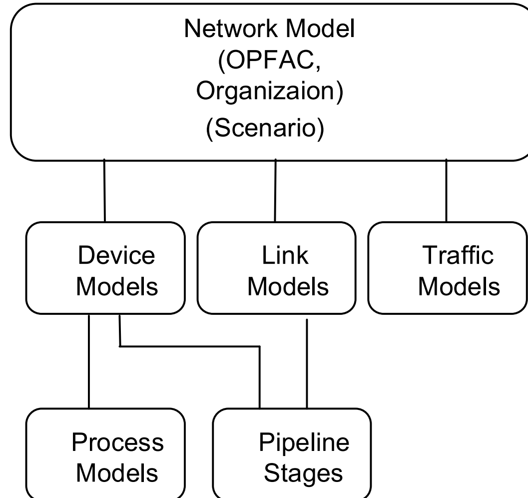


Figure 2-7: NETWARS/OPNET Model Hierarchy

This diagram can be read as follows:

- A scenario is built with OPFACs and Org using device models, link models, and traffic models.
- Device models are built using modules, which include process models, transmitters, receivers and antennas, and associated pipeline stages.

This hierarchy allows modelers to create building blocks, such as process models, OPFACs, and Orgs, that can be reused, reducing the cost of model development. A user who has OPNET Modeler can see the source code for nearly all of OPNET's COTS models and for all the NETWARS models. The user can copy and modify this code to make the model development tasks easier. For more OPFAC and Org information, please see "NETWARS 2006-2 User Manual".

Table 2-1: Model Types and Descriptions

Model	Description
Scenario	A schematic of a network, including devices, links and traffic, terrain, failure scripts, and trajectories for the movement of mobile devices. Scenarios are built with NETWARS by NETWARS users, not model developers.
Organization	A collection of OPFACs, devices, links, and traffic.
OPFAC	A collection of devices, links, and traffic.
Device models	Encapsulate the communications behavior of a physical device.
Process models	A collection of state machines that often model specific network protocols or layers in the Open Systems Interconnection (OSI) protocol stack. The behavior of the process model state machines is implemented in C or C++.
Pipeline stages	Model the communications effect of the physical layer. For wired connections this is usually minor, but for wireless communications the pipeline stages model the effects of radio propagation.
Link models	Model wired connections. These can introduce delay and possess bandwidth constraints.
Traffic models	Model the traffic characteristics/patterns of a use case or scenario.

Some of the models from Table 2-1 are described in more detail in the following subsections.

2.1.3.1 Device Models

Device models, along with link models and utility nodes, are the fundamental building blocks for NETWARS scenarios. Device models embody the conceptual models that emulate real-world devices. Device models are called node models within OPNET Modeler because they represent a node in the network. Device models have two major functions:

- Define the external interfaces of the model, specifically how the user and the Scenario Builder will interact with the model.
- Define the modeling behavior of the device by assembling and connecting appropriate modules, which include process models, antennas, transmitters, and receivers.

Figure 2-8 shows the node editor. The model open in this editor is NETWARS' Cisco 2514 router. This router is based on OPNET's COTS Cisco 2514 model, with minor changes to make it compliant with NETWARS.

The Cisco 2514 is a simple router with two Ethernet ports and two serial ports. The Ethernet ports are listed as hub_rx_3_0 (receive) and hub_tx_3_0 (transmit), and hub_rx_2_0 (receive) and hub_tx_2_0 (transmit). These ports flow into Ethernet MAC process models, mac_3 and mac_2. Further up in the model there are OPNET standard process models for IP, TCP, UDP, RIP, OSPF, IGRP, EIGRP, and BGP. These protocols (and many others, including IPv6 and Multiprotocol Label Switching [MPLS]) come with NETWARS. They do not have to be coded for each device, but simply laid out and connected in the node editor.

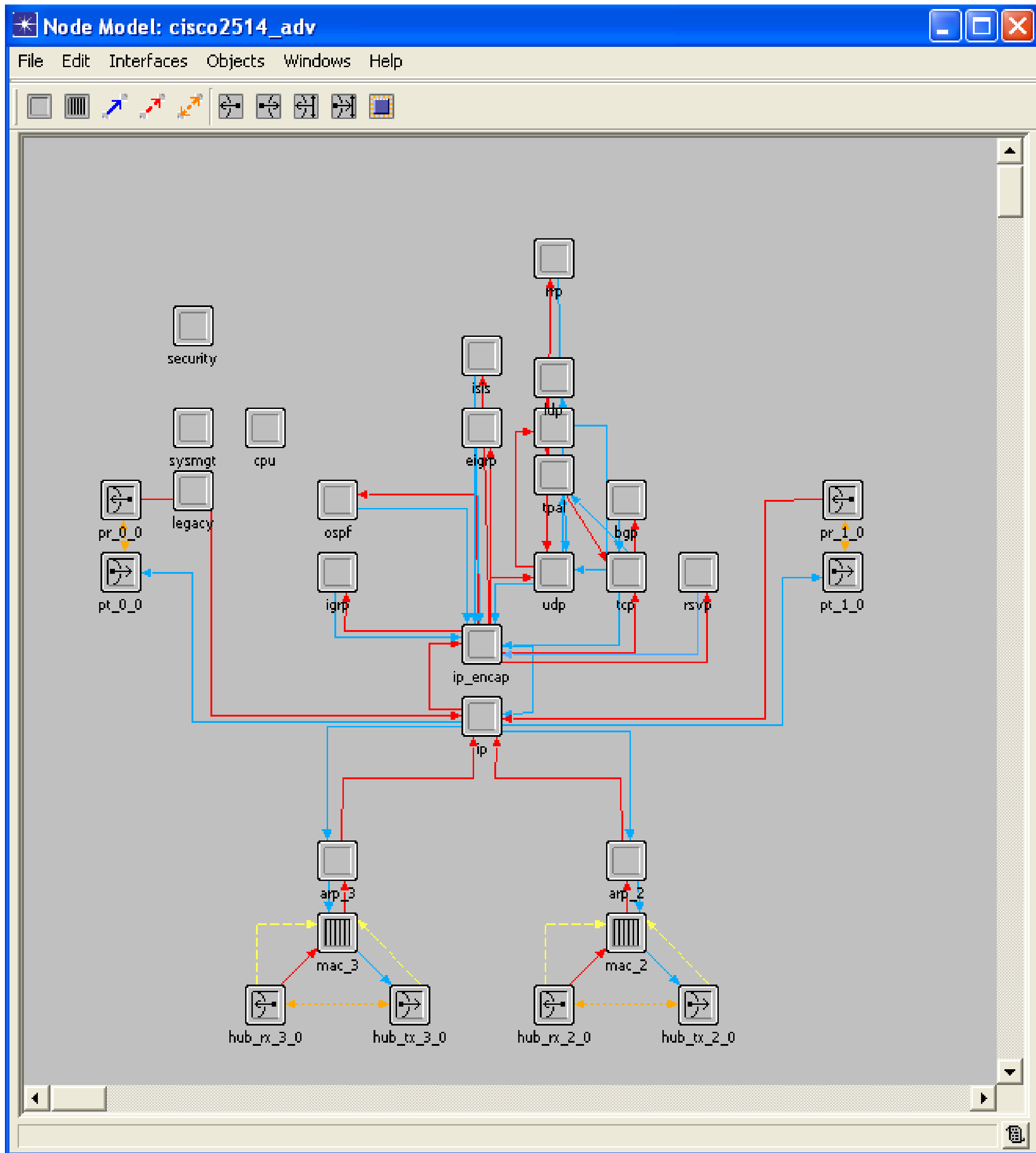


Figure 2-8: Editing NETWARS Cisco 2514 Router Model

2.1.3.2 Process Models (.pr.c)

Device models are created from sub-models called modules. The most important of these are process models, including a special type of process model called a queue model. Several types of modules are shown in the sample device model in Figure 2-9, which depicts the SINCGARS device model:

- pt_0 is a point-to-point transmitter.
- pr_0 is a point-to-point receiver.
- Antenna is an antenna.

- tx_0 is a radio transmitter.
- rx_0 is a radio receiver.
- The remaining modules are process models.

Not shown are the queue model (which can be found in Figure 2-8, as mac_2 and mac_3), bus transmitter, bus receiver, and external system module.

Also seen in the diagram are streams, represented by solid arrows, which facilitate communication between modules; a statistic wire, represented by a broken arrow; and an association, depicted as a dotted double-headed arrow.

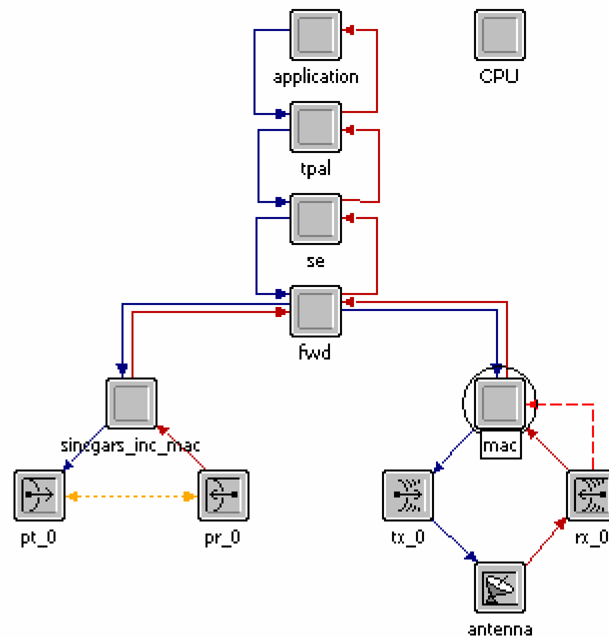


Figure 2-9: Process Models Within SINCGARS Device Model

Process models (including queue models) are created and edited using the OPNET Process Model Editor, which is a part of OPNET Modeler. Figure 2-10, for example, shows the process model being edited. The name of the instance in the device model is "mac," but the name of the process model itself is "sincgars_mac."

The highest level view of a process model is the state machine. (It is assumed that readers of this document are familiar with the concept of a state machine, so this discussion is limited to an overview of the OPNET framework for state machines.) States are represented by colored disks. There must be one initial state, which is indicated by a big black arrow. There are two types of states, forced and unforced. Forced states are transient and are exited immediately after entry. Once a machine enters an unforced state, it remains there until the next event.

State transitions are represented by black arrows. Solid black arrows indicate unconditional transitions. Dashed arrows indicate conditional transitions. The condition is shown in parentheses. In the example, the condition is the name of a C pre-processor macro.

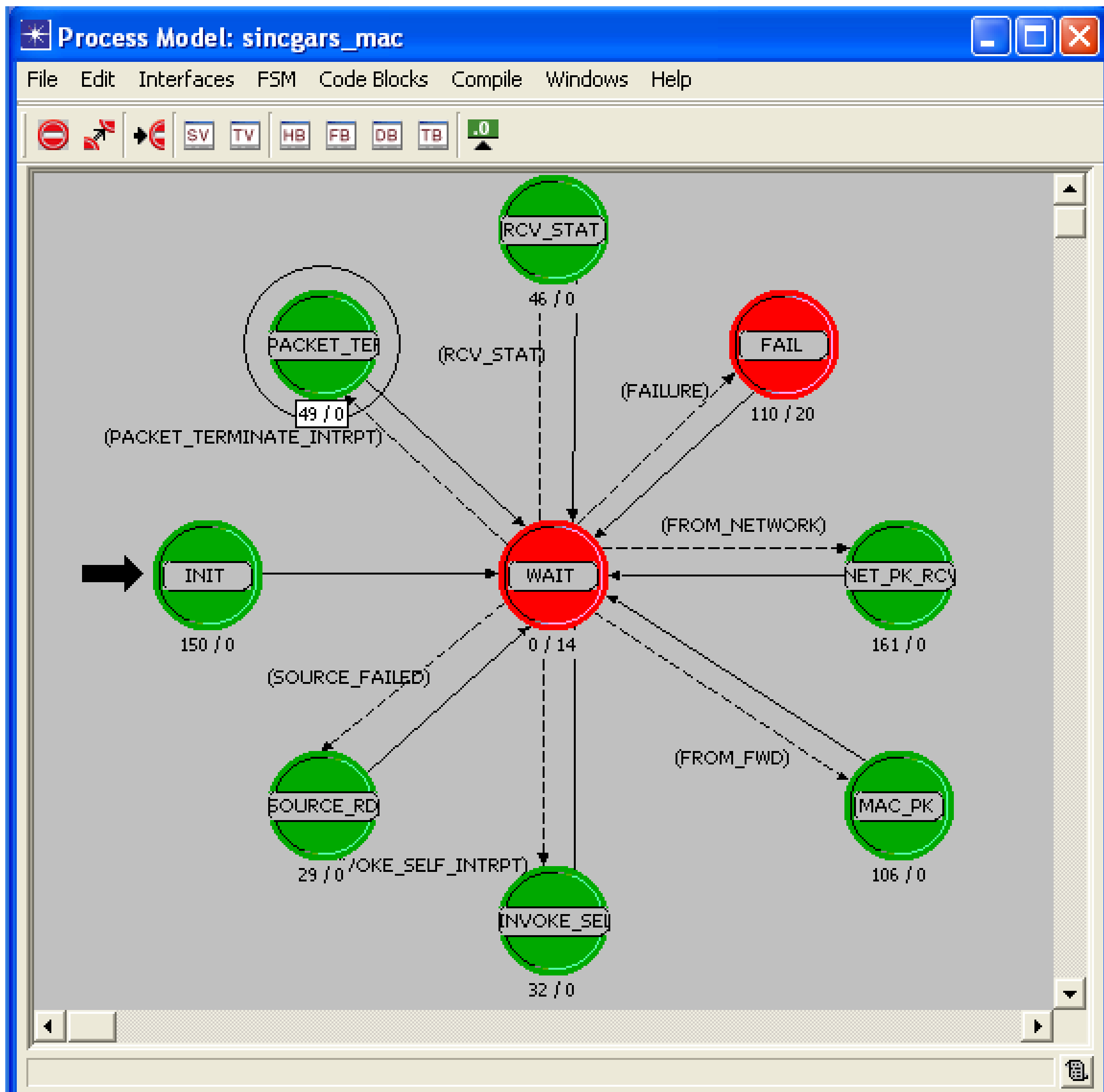


Figure 2-10: Process Model Editor

There are three places where executable code can be invoked:

- Upon entry to a state, called the Enter Execs
- Upon exit from a state, called the Exit Execs
- During state transition, set as the *executive* attribute of the transition

If a state transition executive has been set, then it will be displayed following the transition condition, preceded by a virgule. There are none shown in Figure 2-10. .

An optional feature of the Process Model Editor is that the number of lines of code in the Enter Execs and the Exit Execs can be shown beneath each state. The PACKET_TERMINATE state, for example, contains 49 lines. None of the forced states has an Exit Execs. The WAIT state has 14 lines in the Exit Execs. This is used to query the simulation kernel for information to

determine the type of event that woke up the state machine. The Enter Execs and the Exit Execs can be edited with a text editor, as shown in Figure 2-11.

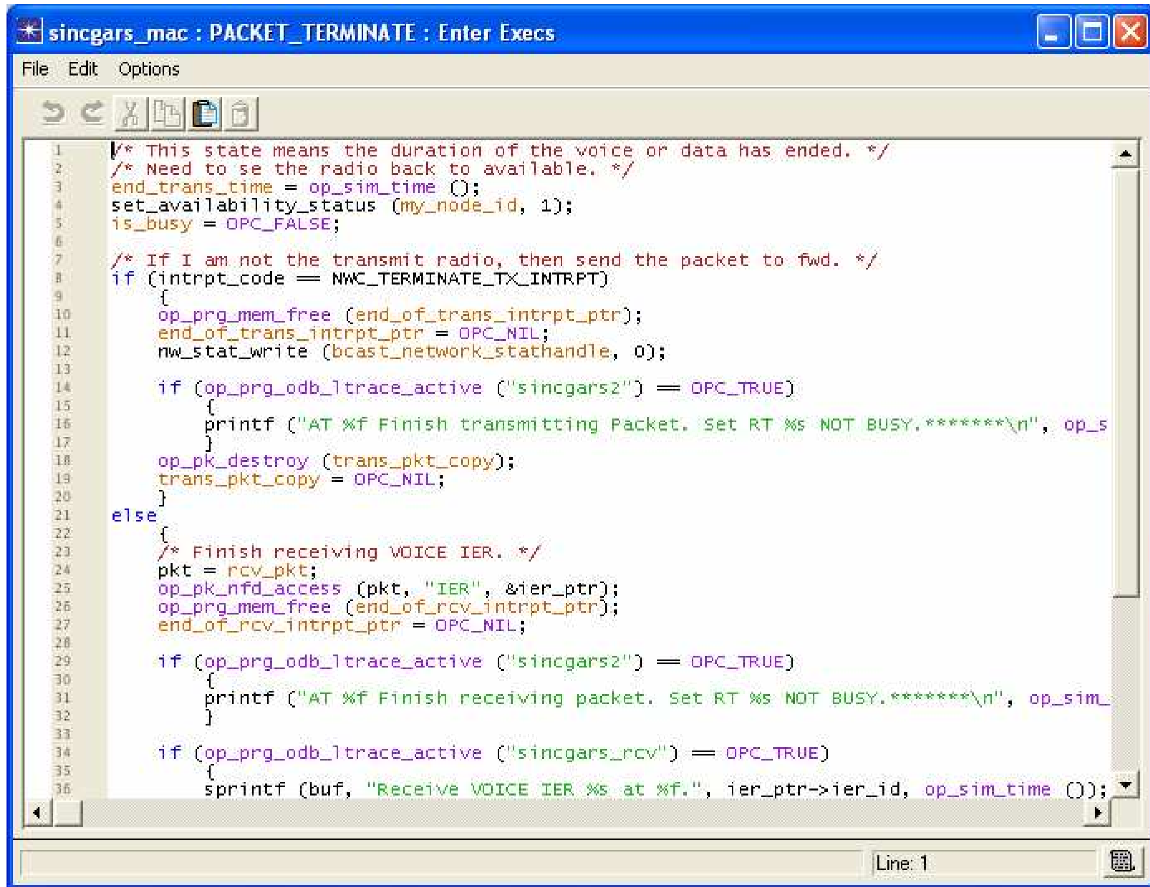


Figure 2-11: Editing C Code in Process Model Editor

2.1.3.3 Pipeline Stages

The physical layer is modeled by pipeline stages, which emulate physical processes. Link models and radio models rely on pipeline stages to implement modular computations and make decisions relating to the transfer of packets between transmitters and receivers. Each pipeline stage is a C language procedure within one C file with the suffix **.ps.c**. There may be seven stages (including the receiver group logic, Stage 0) for a radio transmitter, and for a radio receiver, eight stages. Refer to OPNETWORK Session 1530, Modeling Custom Wireless Effects (see Figure 2-12).

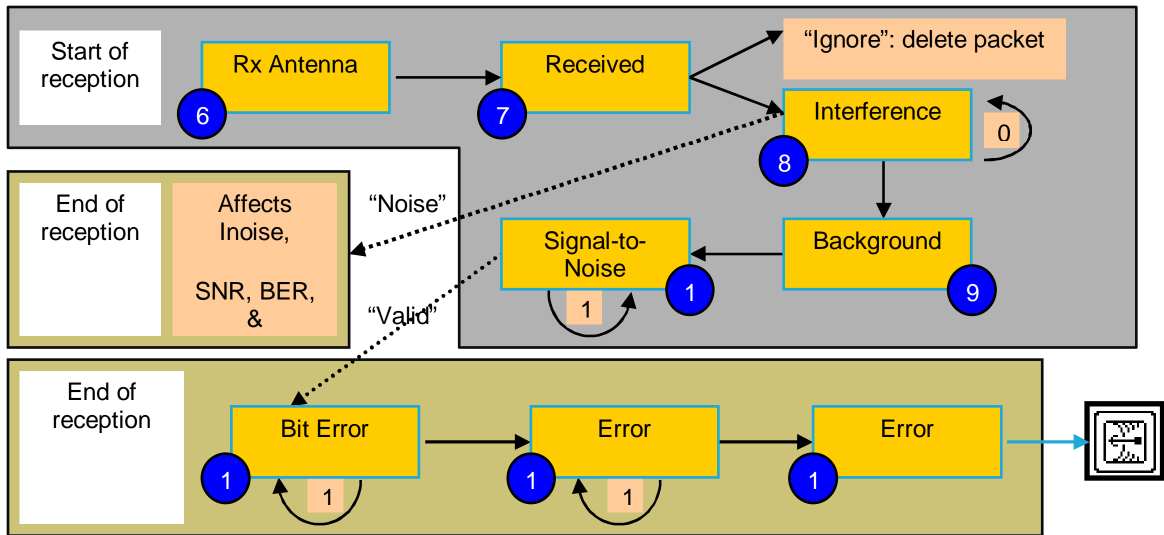


Figure 2-12: Receive Pipeline Stages

This is an illustration from OPNETWORK Session 1530, Modeling Custom Wireless Effects.

2.1.3.4 Link Models

Link models simulate the characteristics of transmission media, such as coaxial cable or fiber-optic cable. Links are used to wire together the device models in a scenario. Important attributes are: whether the link is simplex or duplex; the speed (which may be selected by mnemonics such as OC3 or T1); and the delay (which may be a constant value or based upon speed times distance). There are currently no additional NETWARS requirements for modeling links.

2.1.3.5 Traffic Models

NETWARS makes use of all the traffic models available in OPNET Modeler. These include explicit traffic (modeled by OPNET application models), traffic flows (background routed traffic), captured traffic Application Characterization Environment (ACE), and link loads (background loads on links). In addition, NETWARS provides an IER model, which can model the various types of traffic that IERs specify.

Traffic modeling is performed by study analysts, and more information can be found in the following sections.

2.2 MODEL DEVELOPMENT LIFE CYCLE

Figure 2-13 shows the high-level NETWARS model development life cycle. The life cycle contains seven key activities: identify the model need, define model requirements, design the model architecture, implement the model, develop the test plan and test scripts, validate and verify the model, and document the model. The following sub-sections provide an overview of activities and associated roles and responsibilities of the involved parties.

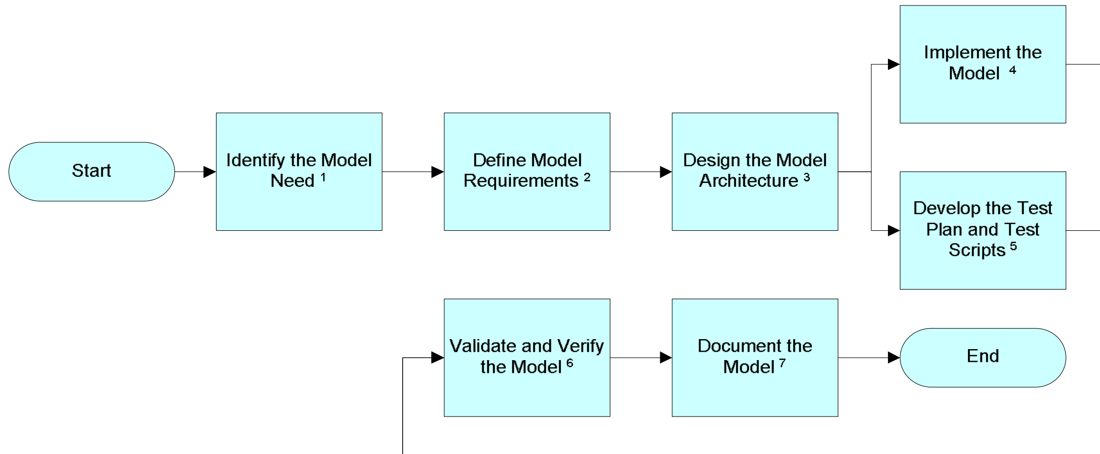


Figure 2-13: NETWARS Model Development Life Cycle

2.2.1 Model Development Roles and Responsibilities

A general model development life cycle contains a program manager, a technical manager, a model developer, SMEs, and a QAE. Their roles and responsibilities are as follows:

- **Program Manager.** The program manager has financial responsibility and visibility to concerns outside the development process. The program manager will take input from all the other individuals, but is responsible for getting the correct model developed at the correct cost.
- **Technical Manager.** The technical manager is responsible for the technical decisions, such as identifying participants, resources, standards, tools, and objectives. The technical manager also provides technical oversight of the development process, and this individual's primary role is to match the requirements and business constraints with the technical constraints.
- **Model Developer.** This individual is a technical expert in coding models with specifications.
- **SMEs.** There are two SMEs involved with the model development life cycle: an operational SME who understands how the equipment is used in the field and a technical SME who understands how the equipment works internally. Both are needed. The SMEs are heavily involved in specifying requirements and validating the model architecture.

- **QAE.** This individual insures certain steps are properly validated. They are responsible for developing and executing test scripts from the test plans.

2.2.2 Model Development Activities

Each life-cycle activity in the life-cycle flow depicted in Figure 2-13 is described in terms of actions, roles, and outputs in the corresponding step in Table 2-2. The Roles column lists the owner of the activities for each step. The Outputs column lists the applicable outputs of each step. The list is not meant to be exhaustive; users should tailor their actions and outputs for their needs.

Table 2-2: Model Development Activities

Step	Action	Roles	Outputs
1.	Identify the model need. The program manager should work with the technical manager to determine the reasons and the facts needed to develop the model. He or she must also identify and allocate resources and responsibilities for supporting the entire model development life cycle.	• Program manager • Technical manager	• Model need • Resources plan
2.	Define model requirements. The program manager should involve relevant parties in the development life cycle. The program manager and technical manager should also clearly identify the model need, the individual responsibilities, and the expected outcomes to the team. The SMEs and QAE should provide information to help the team analyze the model needs and determine the requirements.	• Program manager • Technical manager • Model developer • SMEs • QAE	• Model requirements
3.	Design the model architecture. The development team is responsible for designing a model architecture that can fulfill the requirements.	• Technical manager • Model developer • SMEs	• Model architecture
4.	Implement the model. The model developer should follow the model architecture to implement the model.	• Model developer	• Model
5.	Develop the test plan and test scripts. The QAE should apply the defined requirements and model architecture to develop the model test plan and test scripts.	• QAE	• Model test plan • Model test scripts
6.	V&V the model. The QAE should work with the model developer and SMEs to V&V the model. In addition, the QAE should document the results in the V&V Report.	• Model developer • SMEs • QAE	• V&V Report • Final model
7.	Document the model. The model developer is responsible for documenting the usage of the model in the model user guide.	• Model developer	• Model user guide

3 NETWARS MODEL DEVELOPMENT

This section provides the guidance and requirements for creating traffic and communications device models compliant with the NETWARS modeling architecture and which can interoperate with models in the NETWARS standard library. This section is divided into two parts: the first part will focus on the traffic model development, and will introduce the different types of traffic models that can be shared in NETWARS environment. The second part will emphasize the device and process model development, and will discuss the details of developing communications device and process models. These models can be grouped into three categories: the first category provides guidance to kick off the development process; the second category introduces the common NETWARS model development considerations to the developer; and the third category applies to specific classes of NETWARS model development. Each of these specific class subsections explains how to build a NETWARS component model, and includes the following:

- Defines a NETWARS component class model
- Defines minimum attribute compliance
- Identifies required modules for a device of that class
- Identifies device model initialization steps
- Describes component class interoperability with other NETWARS and COTS classes
- Describes the NETWARS and COTS failure and recovery
- Describes device model measures of performance (MOP) and how to collect statistics
- Describes the NETWARS model documentation standards
- Describes the device model construction process

Phase converters and long-haul modems are not covered in this version of the *NETWARS Model Development Guide*; however, they can be modeled as link models with appropriate latency.

3.1 TRAFFIC MODEL DEVELOPMENT PROCESS

Traffic modeling is performed by individual analysts and developers, and can be a very time consuming effort. It is beneficial, therefore, to be able to share developed traffic models across the NETWARS community. In general, two major approaches can be utilized to deploy and share defined traffic models into a NETWARS scenario: IER and ACE traffic models. The following sections will introduce the differences and highlight the areas of focus for developing traffic models.

3.1.1 Development Approach

The first step in developing traffic models is to gather supporting information to determine the best approach to support the objective. Supporting information includes, but is not limited to, instrumentation data, application design documents, and operational activity logs. In general, ACE models can be created by directly importing packet capture from sniffer, ethereal, NETVCR and other instrumentation equipment. Occasionally, the developer can also create IER text files directly from those captures. The use of IER and ACE can be determined by the availability of resources including time, funding, and manpower.

The following sub-sections will provide highlights on each of the traffic model types, such as ACE, ACE whiteboard, and IER. It is important to note that ACE and ACE Whiteboard will require an external license that can be purchased from OPNET Technology Inc. It is not included with NETWARS.

3.1.2 ACE Traffic Model

NETWARS provides the same functionality as other OPNET products for utilizing the ACE traffic model (atc.m) to create traffic profiles and generate traffic load. ACE traffic models are created directly from captured network packet data through the ACE import function. In addition, the ACE application provides capabilities to import network packet data from a wide variety of popular network monitoring and instrumentation tools and software, including:

- OPNET .appcapture
- Network Associates' Sniffer
- Industry-standard Binary Ethernet format (.enc, .cap)
- Other analyzers (such as NetScout) can also generate .enc files
- Binary Token Ring format (.trc)
- Binary FDDI format (.fdc)
- Free utilities, such as TCPdump and Windump
- Comma Separated Value file (.csv)

For detailed information on the procedures for importing captured data to ACE, please see the "Introduction to the ACE Editors" and OPNET Modeler ACE Overview online documentation.

NETWARS can deploy the ACE traffic models directly into a network scenario to conduct simulations and performance analyses, providing a mechanism to share traffic models with other developers in the community. With the use of ACE to create the traffic models, users can re-use the same traffic models in different network scenarios to support simulations without additional effort. Refer to "Introduction to the ACE Editors" for detail procedures on using ACE and ACE Whiteboard to create the traffic model (atc.m). Note that an external ACE module license is required to operate ACE to import packet traces and create the corresponding traffic models.

3.1.3 ACE Whiteboard Traffic Model

ACE Whiteboard is another external OPNET module that can be used to create traffic models. Similar to ACE, NETWARS provides the functionality to deploy ACE Whiteboard traffic models into a model scenario. ACE Whiteboard provides the ability to modify existing ACE and ACE Whiteboard models. Furthermore, the primary advantage of using ACE Whiteboard is that developers can use ACE Whiteboard to create traffic models from scratch.

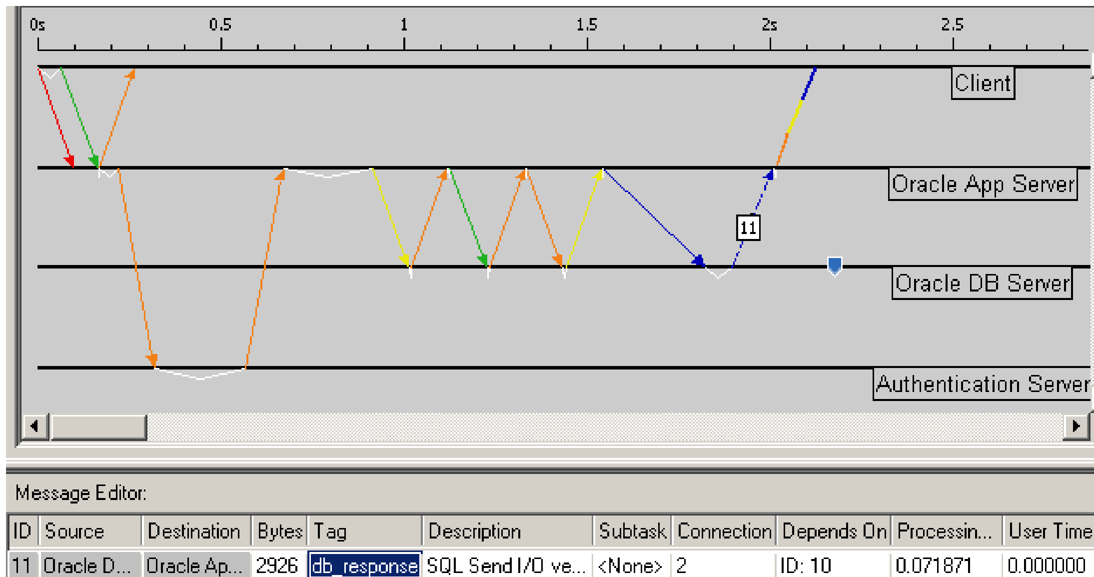


Figure 3-1: ACE Whiteboard Screen Capture

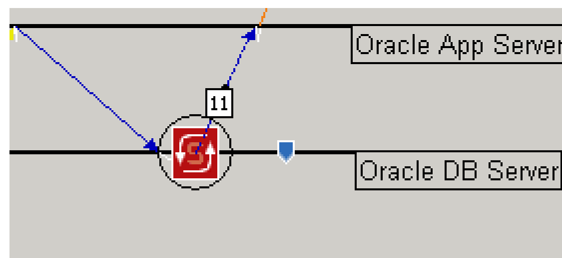


Figure 3-2: ACE Whiteboard Python Logic

As Figure 3-1 and Figure 3-2 show, a developer is needed to define the associated tiers, required transactions, transaction size, dependency, user time, and processing time for the traffic model. Another important feature of ACE Whiteboard is the use of Python to create logic scripts to model the dynamic behaviors of an application. It should be noted that ACE Whiteboard requires an additional module license for use in NETWARS. For detailed information on the usage of ACE Whiteboard, please see the “Introduction to the ACE Editors” and OPNET Modeler ACE Whiteboard Overview online documentation.

Developers should conduct information gathering to support traffic model development in the ACE Whiteboard. The information includes, but is not limited to:

- All required tiers or end-devices, such as database servers, web servers, and workstations.
- Required messages and transactions associated to each defined tier.
- Transaction characteristics, such as message size, dependency, processing time, etc.

ACE Whiteboard provides powerful capability and flexibly to create traffic models from scratch, yet requires the highest level of effort.

3.1.4 ACE and ACE Whiteboard Traffic Model Concerns

In order to integrate and use the ACE traffic models in NETWARS, there are several important concerns that include:

- **Number of tiers** – Traffic model developers should make sure the ACE models contain the correct number of tiers to support the entire application transaction. Tiers refer to different end-user network devices, such as servers, computers, workstations, and phones.
- **NETWARS scenarios** – NETWARS scenarios should contain all the associated nodes corresponding to each individual tier in the planned traffic models.
- **Understanding the need of discrete event traffic and background traffic** – ACE and ACE Whiteboard traffic models can be deployed as either discrete or background traffic load. Developers and users should deploy the traffic models corresponding to their needs.
- **Validate and Verify traffic models** – Developers should V&V the traffic models before using them to conduct further simulations. During the V&V process, developers can make use of the NETWARS built-in functionalities to execute simulations and capture the packet traces. The capture results are compared against the original packet capture to V&V the traffic models. In general, the developers can use the following values to conduct the comparison:
    - Transaction dependency
    - Number of transactions
    - Dependency delay
    - Message size.

3.1.5 IER Text File

One of the critical features of NETWARS is the use of IER and thread to define traffic flow. Traffic models based on IER text files can be created directly by using operational missions and existing IER databases. In addition, the traffic models can be easily modified and created through the use of text editors or Microsoft Excel spreadsheets. If an Excel spreadsheet is used, users can export the IER test file into plain text format with each column delimited by a tab character.

Figure 3-3 shows an example of the IER text file. The file contains the required general attributes for each IER, and each line represents one IER. The general attributes include:

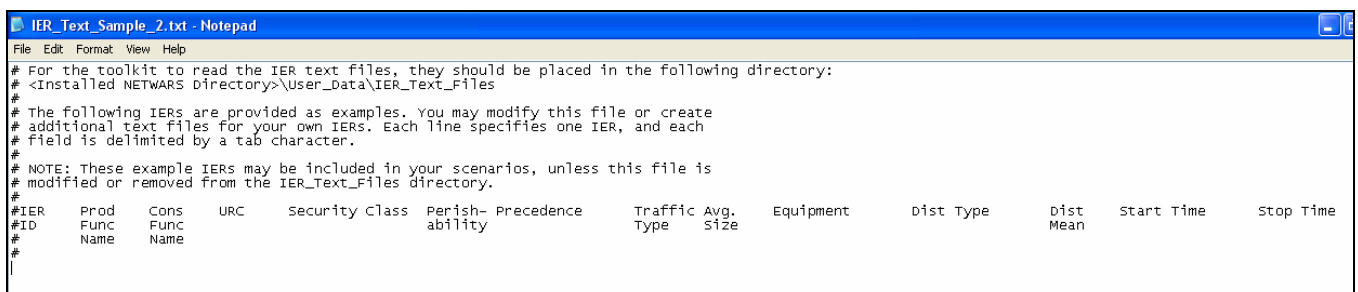


Figure 3-3: IER Text File Sample

- **IER ID –** Identifies an IER in the database. IDs for IERs that are created by a user start with the prefix USER, as to not conflict with IER IDs in the database. Background IERs start with the prefix BKGD.
- **Producer Functional Name –** Identifies OPFAC Producer Functional Name of the IER.
- **Consumer Functional Name –** Identifies OPFAC consumer of the IER.
- **URC –** Identifies the relationship between the Producer OPFAC and Consumer OPFAC. Please see “NETWARS User Manuel” for detailed information.
- **Classification –** Specifies the security classification of an IER. The security classification of an IER is one criterion that determines the system element through which the IER is transmitted.
- **Perishability –** Specifies the time in seconds during which the IER is alive.
- **Priority (Precedence) –** Determines the number of transmission retries and the wait time between successive retries.
- **Traffic Type –** Specifies the type of IER traffic, such as data and voice.
- **Average Size –** Indicates the average size of the IER in bytes.
- **Equipment –** Specifies the system element, such as computer and radio, over which the IER can be transmitted.
- **Distribution Type –** Indicates the inter-arrival distribution for the IERs.
- **Interarrival (Distribution Mean) –** Represents the time, in seconds, between IER firings.
- **Start Time –** Identifies the time, in seconds, in which the IER will begin firing after a simulation begins.
- **Stop Time –** Identifies the time, in seconds, in which the IER will stop firing after the simulation begins.
- **Producer Device** and **Consumer Device –** Indicates the devices transmitting and receiving the IER.
- **Transport Protocol –** Identifies the protocol used for transporting the IER.

Other than the above general attributes, the link-16 IER (J-Series message) needs an additional attribute for entering the Network Participation Group (NPG) number of the IER. Detailed information can be found in the “Link-16 Model User Guide.”

The IER Text File should be placed in the “<install drive>:\NETWARS\User_Data\IER_Text_ Files” directory. It is important to note that users may install NETWARS in different folders.

3.1.6 Traffic Model Deployment

The purpose of this section is to highlight the important issues while deploying the traffic models into NETWARS scenarios, and therefore does not contain step-by-step procedures. The following sub-sections will provide the information and reference necessary for developers to deploy the traffic models into NETWARS scenarios.

3.1.6.1 ACE and ACE Whiteboard Traffic Model

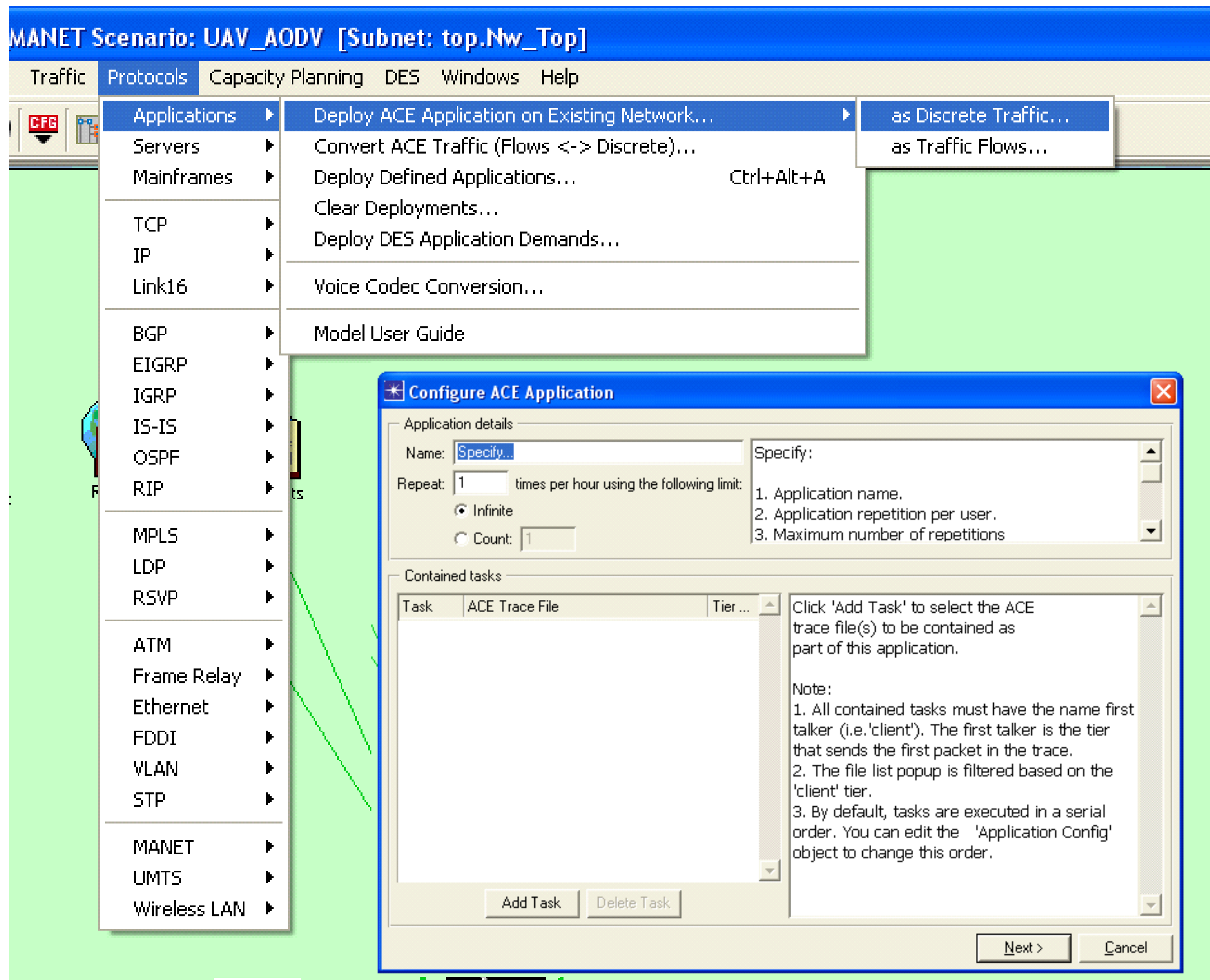


Figure 3-4: ACE and ACE Whiteboard Traffic Model Deployment GUI

As shown in Figure 3-4, NETWARS employs the same technology from OPNET Modeler and IT Guru to import ACE and ACE Whiteboard traffic models into simulation scenarios. The function bar of NETWARS supports the importation of traffic models to applications as discrete event or traffic flow formats. After the models have been imported, the users can deploy the applications to corresponding nodes in the scenarios through the use of the "Deploy Defined Applications" function. Please see "Introduction to the ACE Editors" for further information.

3.1.6.2 IER Text File

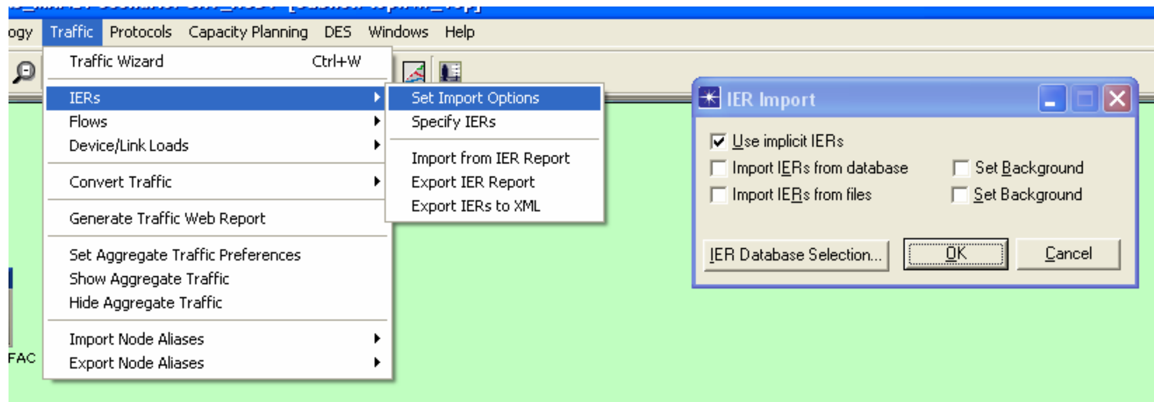


Figure 3-5: IER Import Manual

NETWARS provides a function to import IER directly from text files, as demonstrated in Figure 3-5. By selecting “Import IERs from files,” NETWARS will generate IER traffic from IER text files during simulations. If “Set Background” is checked, the IER traffic will act as background traffic during simulations. In order to import the IER from the text files, users need to store the IER text files in a specific directory (i.e., <Installed NETWARS Directory>\User_Data\IER_Text_Files). NETWARS will read all the files in this directory, and the IER with corresponding functional name will be generated during the simulations. Therefore, users need to make sure the directory only contains the IER text files that will be required to support the simulations. Other IER text files can be stored in another directory created by the users. Please read “NETWARS User Guide” and “NETWARS Code of Best Practice” for further information.

3.2 COMMUNICATIONS DEVICE AND PROCESS MODEL DEVELOPMENT PROCESS

The development process is the second of three phases in the NETWARS communication device model life cycle. At this point, the developer has a set of model development requirements that can be used to define the development approach. Figure 3-6 shows the high-level development process that consists of three individual development approaches which guide the developer to kick off the model implementation with appropriate procedures.

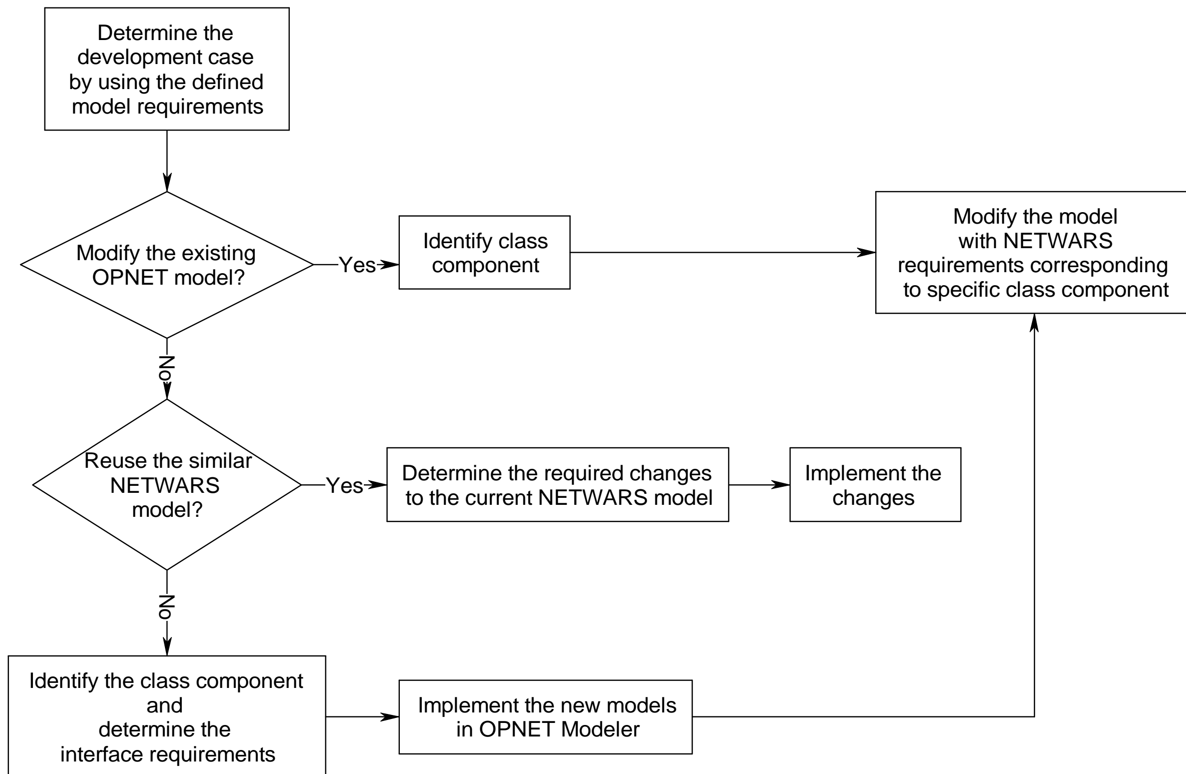


Figure 3-6: High-Level Model Development Process

3.2.1 Development Approaches

In order to determine the most efficient way to implement the model, the developer needs to match the development effort to appropriate development approaches, such as:

- Modifying the existing OPNET model to be NETWARS compatible
- Surrogating from the existing NETWARS model
- Developing a new model

The following subsections introduce the key considerations of each specific development case.

3.2.2 Modifying the Existing OPNET Model to Be NETWARS Compatible

In this case, the scope is to convert an existing OPNET model into a NETWARS model. The goal of this subsection is to provide the basic approach and key focuses for the developer to kick off the modification process. They are as follows:

- Identify the component class of the device and the OPNET version that was used to implement the model.
- If the model is implemented in an older version, then it must be upgraded and matched to the version of NETWARS.
- If the device used a COTS traffic model, then it will work as-is in NETWARS using DES only.
- If the device "wants" to use the NETWARS IER traffic specification infrastructure, then ensure it has required attributes (specific for each component class).
- If the device is an end device, it needs the addition of the relevant "se" module.
- For interoperability with specific NETWARS component class devices, refer to Section 3, which has a compliance subsection for each component class.
- To get proper device functionality in CP/logical views, make sure the device has the required attributes (specific for each component class). Scenario Builder may still require CP routing/logical view code enhancements to support full CP/logical view functionality.
- The link deployment wizard will ONLY work if it has relevant self-description (and a matching link name in the LinkTypeMap.gdf file).
- If it has complex attribute specification, then Scenario Builder may require a wizard-like functionality to ease the device deployment.

3.2.3 Surrogating From the Existing NETWARS Model

In this step, the developer re-uses a similar NETWARS model as the foundation to construct the new model. The key considerations while surrogating from the existing NETWARS model include the following:

- If surrogating ONLY involves attribute default changes, then NO modification would be required.
- If surrogating involves new attribute addition or changing the behavior of contained modules, then it may need device model functionality enhancements.
  - In DES, process models/external files/pipeline stages need to be enhanced.
  - In CP, CP routing changes need to be determined.
- If surrogating involves changing physical layer characteristics (like changing radio transceiver frequency, power, etc.), then NO modification would be required.
- If surrogating involves adding new interfaces (ports), then relevant self-descriptions for the new interfaces (ports) need to be added.

3.2.4 Developing a New Model

In this case, the developer is required to construct a new model from scratch.

- Identify the component class of the device and its interface requirements.

- If the characteristics of the device include protocols and technologies available in the OPNET COTS offering, then use device creator to create a new model with required interfaces and technologies.
- If device creator cannot be used, then build the new model in OPNET Modeler according to device specification (building process models/external files/pipeline stages).
- Perform all the steps in the "Modifying the Existing OPNET Model to Be NETWARS Compatible" subsection.

3.3 MODEL INTEROPERABILITY ISSUES

Before development of any device models in the NETWARS environment; the developer needs to pay attention to the interoperability issues that are associated with the interactions between different device models. This subsection in particular discusses the interoperability concerns that users must have before starting the model design/implementation. Based on the objective of the model development and the final modeling environment in which users will deploy their models, interoperability can be separated into four main categories:

- Compatibility issues
- Interfacing issues
- Self-description issues
- Versioning issues.

The following provides some of the common concerns and issues among those four categories that a developer will face. In addition, examples are used to address the detail of those concerns.

3.3.1 Compatibility Issues

Compatibility issues include functionality, protocols, and IP auto-addressing issues. The following subsections discuss these in detail.

3.3.1.1 Functionality Issues

A particular device model's intended behavior determines some of its compatibility with respect to other models. The model developer should give due attention to interoperability, starting at the high-level design of the device. At this point the developer also needs to give attention to the high-level function of the models with which it will interface.

For example, when building a radio device model that has the ability to generate IER traffic, the user needs to know the functions of the operational element (OE) at a high level. (The OE coordinates sending and receiving IER-based traffic.) This reduces or ideally eliminates work duplication and code overlap between the radio and the OE. In this example the user should know the following:[1]

- The radio does not need to write IER statistics.
- The radio does not need to read the IER information.
- The radio does not need to schedule IERs.
- The availability of the radio for transmission and/or relay will be dependent on the OE implementation.

This example merely covers, at a high level, interfacing the radio with the OE. During the high-level design, the developer needs to make a list of devices (per layer) that will interface directly (wired/wireless connection) or indirectly (using other communication mechanisms). Usually, model specifications clarify device functions, but this quick check should be performed to discover any functionality-related overlaps in advance.

[1] Assuming that the behavior of the OE is similar to the one present in the NETWARS standard model library.

3.3.1.2 Protocol-Related Issues

In addition to functionality, the developer should make sure that the model under development interfaces with the correct protocols and/or technologies. For example, the current NETWARS model *nw_ethernet_wkstn.nd.m* has two specialized interfaces—one that supports TCP transport protocol and one that supports User Datagram Protocol (UDP). It has a separate implementation of the system element (SE) for either of these protocols.

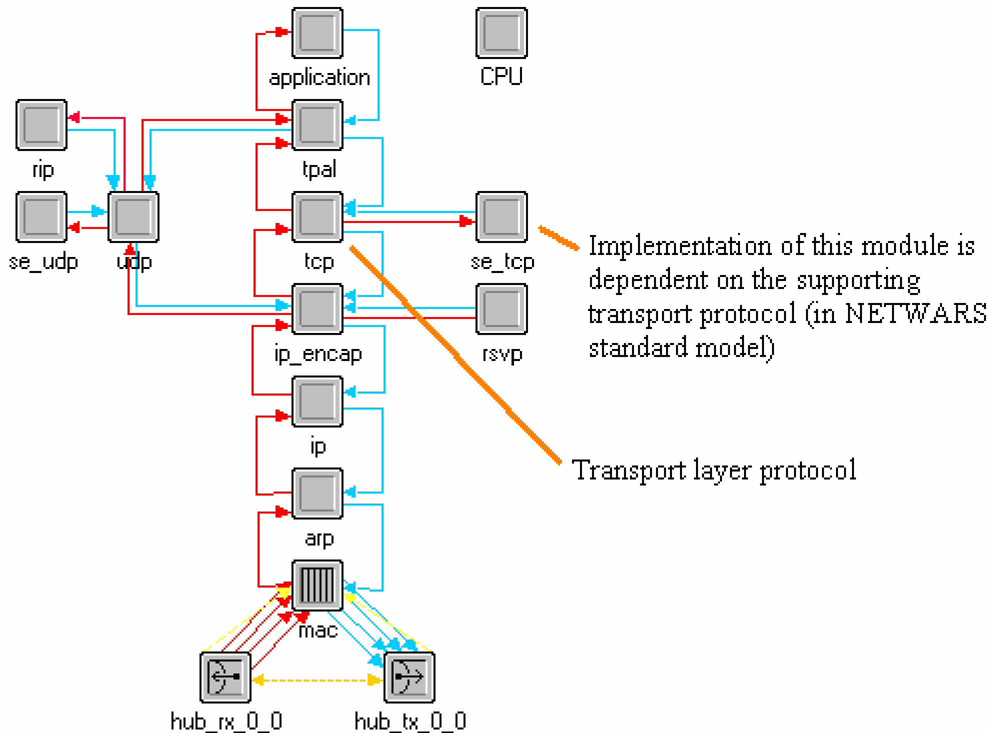


Figure 3-7: Protocol Dependency (e.g., Ethernet Computer Model)

Based on the supporting protocol layer stack, the developer needs to do some custom model development. Also, in some cases protocols (upper- or lower-layer protocols) have interdependency upon one another, and the developer must consider this while performing the high-level design for the device model.

3.3.1.3 IP Auto-Addressing Enhancements

Every IP interface that has a link connected to it needs to have an IP address. If the network is huge, then assigning addresses manually to every interface becomes cumbersome. To make it easy for the user, OPNET Standard (COTS) models have a feature called “IP Auto-Addressing.” By default, device model instances have auto-addressing enabled in a network, and the first IP process to initiate in the simulation automatically assigns IP addresses to the interfaces that have their value set to “Auto Assigned.” To accommodate new models developed, model developers need to enhance this COTS utility, typically (but not only) for Layer 2 custom models. Currently, support exists for the NETWARS standard models such as Promina, circuit switches, satellite terminals, and the like.

3.3.2 Interfacing Issues

One of the key steps in development involves taking into account the model integration issues (in the case of a single model, integration of different modules/processes[2]). The model developer needs to realize that not all of the model development progresses in seclusion (i.e., the various modules of a device model need to interface with each other, even during development). Recognizing the integration issues sooner rather than later benefits the model integration process. Initial designs for model development should address this. The various components of this category are information-sharing and communication aspects.

3.3.2.1 Information Sharing

Through the interfaces, information can be shared between the two process models that belong to the same module, different modules of the same device model, or two completely different device models. This can be done in a variety of ways, some of which are discussed in the following subsections.

3.3.2.2 Process Registry

The OPNET simulation kernel allows any number of OPNET process instances to register themselves in a global (i.e., accessible to any process in the scenario) process registry. The processes register themselves with the required attributes only once during simulation (typically upon creation); however, processes can add new attributes/descriptors whenever required. Other processes can later access these attributes during the simulation's execution. Model developers should consider what information, in the form of process registry attributes, processes should publish via the process registry upon their creation or modification. It is necessary that the new processes written realize what information (attributes) published by previous processes could be of use.

An example of process registry[3] use can be seen in the NETWARS satellite models, where the satellite space segment registers its attributes in the process registry and then the earth terminals discover (retrieve) this information during their initialization.

3.3.2.3 Module-Wide Memory

Module memory is the most permanent and widely scoped memory provided in OPNET modeling (except for global variables). A single block of memory can be installed for a module by any process that is owned by that module. Installation is performed by calling the Kernel Process (KP) *op_pro_modmem_install()* and passing the address of the memory block. Any process owned by the module can then obtain the installed address by calling the KP *op_pro_modmem_access()*. The structure and contents of the memory block are entirely the responsibility of the model developer, as is memory de-allocation of previously installed blocks when a new installation occurs. Initially the address *OPC_NIL* is installed to indicate the absence of any module memory.

---

[2] Processes are instances of a process model. For example, *ip_dispatch.pr.m* is a process model that can be instantiated a number of times in a simulation of a network that contains many routers and workstations.

[3] Refer to the OPNET Product documentation for details on the process registry and its use.

Again if the developer is adding the new process models to an existing module in the node model, this would be a place to look for some already initialized information.

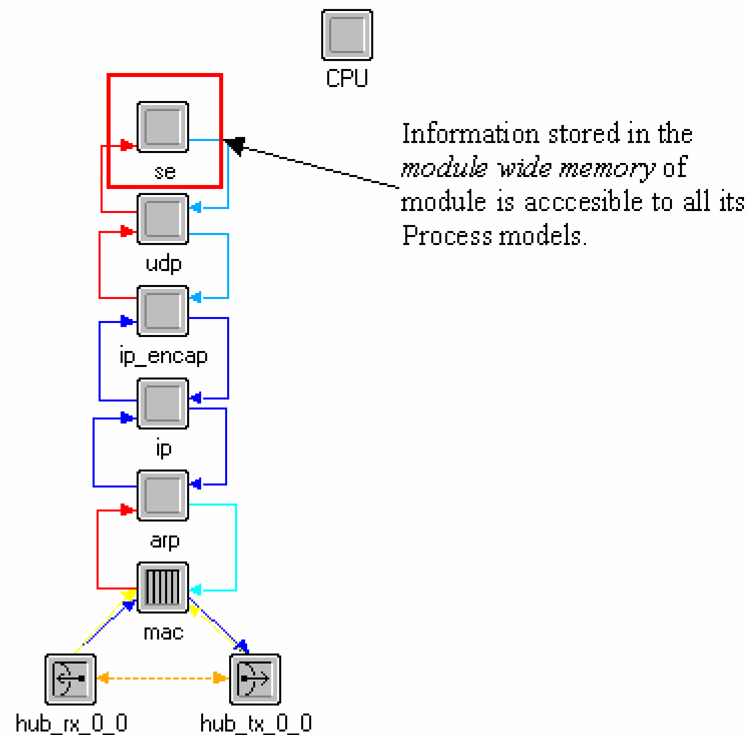


Figure 3-8: Module-Wide Memory (e.g., Ethernet Computer Model)

3.3.2.4 State Variables

State variables are analogous to the global file and are associated with each process model. Other processes can access these variables through the use of the KP *op_ima_obj_svar_get()*.

3.3.2.5 Global Variables

Global variables are the regular global variables declared in the header block of one process and can be used by other processes. Use of these variables should be minimal. The developer should declare the variable in the header block of one process and declare the variable as an extern in the header block of all other process models. Note that declaring a variable in the header block also makes it global to all instances of the process in which it is declared, as opposed to state variables where the information remains local to the process instance.

Following is an example of using a global variable:

> If the global variable is named foo_var and is of type int, declare the variable in foo.h:
> extern int foo_var;
>
> Then define it in foo.ex.c (or alternatively foo.pr.m):
> #include <foo.h>

```
int foo_var;

Now to access or set it in bar.pr.m:
#include <foo.h>
foo_var = 10;
```

3.3.3 Communication Aspects

This subsection introduce the key aspects of communication, such as packet formats, transceivers, process models, link models, the link type map file (i.e., LinkTypeMap.gdf), packet encapsulation, interrupt types, and interface control information (ICI).

3.3.3.1 Packet Formats

Packets are the units of transfer of information in a data network. In OPNET/NETWARS terminology, there are two basic types of packets: formatted and unformatted. The formatted packets are the most commonly used mode of data transfer because formats can easily act as a constraint on the transmitter and the receiver of the device model. Packet formats define the internal structure of packets as a set of fields. Refer to Appendix D for a list of packet formats used in NETWARS standard models. The packet format constraints are placed at transceivers, process models, link models, and the LinkTypeMap.gdf file. For example, a Promina device and the associated link that connects two of its Wide Area Network (WAN) ports, *Promina_wan_link*. Because the packet format affects multiple model elements, it can be a significant issue when integrating different device models.

3.3.3.2 Transceivers

Each pair of transceivers in a device node model has a list of packet formats it can support. In the case of Promina, the packet formats supported by the WAN transmitter and receiver are pro_cx_pk, pro_hello_pk, and pro_wan_pk, which are packet formats to support the Promina Cell Express packets, Promina Hello packets, and Promina data packets from neighboring Prominas.

3.3.3.3 Process Models

This is the place where the packets are actually created, received and/or passed on by the modules above or below using the stream or forced interrupts. A process model can be said to be supporting a packet format if the stream interrupt received by this process model with this stream interrupt is properly handled. In the case of Promina, the process model that handles (processes) the above-mentioned packet formats is pro_wan_port_controller. The packet format supported on a pair of transceivers is decided based on the design of the process models.

3.3.3.4 Link Models

Every link also supports a list of packet formats; if trying to connect a link between two devices and the packet formats supported by the transceivers are not supported by the link model itself, then the connection between the two devices will be invalid. Continuing with the Promina example, the promina_wan_link used to connect the two WAN ports supports promina_hello_pk and promina_wan_pk.

3.3.3.5 Link Type Map File

This is a text file that contains information about the various link types used in the NETWARS environment that is primarily used by the NETWARS Scenario Builder to determine if an external link connected between two devices supports the assigned ports (transceivers). Refer to the *NETWARS Interface Control Document* for details on this file, including its format and content.

3.3.3.6 Packet Encapsulation

Additional information, such as header information, is added to the packets as they are forwarded from one module to the other. One of the common methods is to use packet encapsulation, where the original packet is wrapped in a new packet format and the relevant packet fields are populated (the original packet now being a packet field of the new packet). For example, as a TCP packet goes down the protocol layer stack, it gets encapsulated into an IP datagram, which then gets encapsulated into the data link layer technology packets (e.g., Ethernet), and so on. Later, on the receiving end the same packet gets de-capsulated (i.e., the information is stripped), and the de-capsulated packet is then sent up the protocol stack. The correct encapsulation and de-capsulation processes are necessary at each layer (OPNET module), and one of the interoperability concerns that developers should have is handling it appropriately in their models and forwarding packets of formats as expected by the neighboring modules.

3.3.3.7 Interrupt Types

When a process is invoked by an interrupt, it usually is in a state in which it expects a limited set of interrupts. The first concern of the process is to determine the type of the incoming interrupt, so it can tailor subsequent processing appropriately. The KP *op_intrpt_type()* provides the process with an integer code that represents the type of the current interrupt.

Apart from the packets (stream interrupts) that can be received by a process from other processes, there are other interrupts that can affect the behavior of a model. It is imperative that caution be taken in the handling and scheduling of these interrupts because they are the primary means of communication in a simulation.

Each interrupt type can have many different purposes. For instance, a single process might schedule self-interrupts to model various kinds of processing delays and time-out intervals. To distinguish the purpose of such interrupts, and hence provide the receiving process with context-sensitive processing ability, an integer code is associated with self-, remote, and multicast interrupts. The code of the current incoming interrupt is available from the KP *op_intrpt_code()*.

It is important that the process model under development be ready to handle all the interrupts it is designed to handle. For example, if the process model under question is development of a new SE that supports both TCP and UDP transport protocol, then the application module (SE) generates the traffic based on the information received from the OE. In this case, the SE module should be aware of the communication mechanism that the OE will be using to transfer this information (e.g., remote/stream/forced interrupt) and should be able to handle that particular interrupt in a desired fashion (generate the traffic based on this information).

A less preferred approach is to have a default handling of any interrupts that model is not defined to handle (using the interrupt steering mechanism). This is done by defining a state transition with its *condition* attribute set to “default,” as shown in Figure 3-9. This will apply to interrupts received at the source state of the transition that the process does not know how to handle.

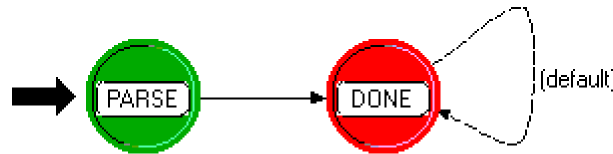


Figure 3-9: Default Interrupt Handling

3.3.3.8 Interface Control Information

An ICI is a structured collection of data that is transferred between processes, as a form of inter-process communication. An ICI becomes associated with an interrupt if a process installs the ICI prior to taking the action that causes the interrupt. Layered protocol interfacing is the main application of ICIs, but they can also be used to associate information with sophisticated self-interrupts or peer-to-peer remote interrupts.

Because ICIs are associated with interrupts, handling the information in the ICIs is as important as handling the interrupts themselves. In case of the current NETWARS standard models, the communication between the OE and the SE is established via a remote interrupt. There is an ICI associated with this remote interrupt that has the information about the IER that this SE needs to generate. The KP *op_intrpt_ici()* is used to get the ICI associated with the recent interrupt and *op_ici_format()* to get the format of the associated ICI.

Another example of the use of ICIs is in the *oe_threads* process model (of the OE). In this process model, all the thread instances are scheduled at the start of the simulation, and the ICIs are associated with self-interrupts. These ICIs contain the actual information regarding the thread that needs to be fired. Once the process receives these self-interrupts it retrieves the ICI information and then actually fires the thread segments. The KP *op_ici_create()* is used to create an ICI and *op_ici_install()* to install it with the interrupt.

The most important interfacing issue that can be associated with ICIs is their formats. The interfacing process needs to know what ICI format to expect and what information is available in that ICI format (ICI files are stored as *.ic.m). Refer to Appendix E for the list of ICIs currently used in the NETWARS standard models.

3.3.4 Self-Description Issues

Every model produced in NETWARS holds some information regarding how it can interface with other model types. NETWARS refers to this part of the model definition as the self-description. This subsection plays a key role in defining device interoperability and provides guidelines for how to define the self-description of the custom model.

The self-description information for each model will vary depending on the class component of the model (e.g., a network layer device versus a datalink layer device), supporting technologies, and so on. The port information is one of the most common pieces of information that is looked for within the self-description. Following discussions point out how this information is specified for the NETWARS models. If the custom models do not support the same packet format information as NETWARS models, then self-description information based on the developed models will have to be developed.

3.3.4.1 Port and Port Groups

The NETWARS Link Deployment Wizard depends on the information present in devices' and links' Port Self-Descriptions. The Port Self–Description can be accessed by selecting "Interfaces | Self-Description" from within OPNET Modeler's Node Model Editor or Link Model Editor. For all the NETWARS models, each port category must have a self-description port object. For example, MRC-142 (NETWARS standard device model) has the following ports:

- **Point-to-Point Ports**. ptp_pt_0, ptp_pt_1
- **Radio Ports.** radio_tx_0, radio_tx_1

Two port objects (ptp_pt_<n> and radio_tx_<n>) will be created with a range from 0 to 1 (see Figure 3-5).

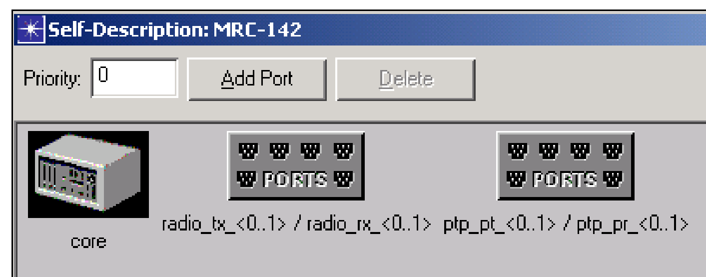


Figure 3-10: Self-Description Port Objects

Each port category needs an "interface type" characteristic defined for it. This interface type defines the technologies that the set of ports supports. Refer to Appendix V for details.

3.3.5 Versioning Issues

To upgrade the models to a new NETWARS standard model library, users need to force-compile all their models with the new header files. NETWARS supports backward compatibility. For example, models developed on Version 11.5 can be applied on Version 12.0, but not vice versa.

3.3.5.1 Force Compilation

This is one of the easiest but very vital steps in development of models that are interoperable. It is necessary to compile all the models with the correct headers. During the development efforts, it is possible that the developer may have had to modify or enhance the current headers in either the NETWARS or OPNET standard model library.

To force-compile the models used in a particular simulation, check the force model recompilation checkbox under “Execution|Advanced|Compilation.”

To force-compile all the models in directories listed in the mod_dirs attribute of the Sim_Domain\op_admin\env_dbX.Y file, the user needs to open an OPNET console. Force compilation can be done from this console as follows:

- set opnet_user_home=<Netwars_Install_Dir>\Scenario_Builder
- op_mko –all >comp_info.txt

This will compile all the models and put the compilation information in the comp_info.txt file.

3.4 NETWARS COMPLIANCE REQUIREMENTS

To develop a NETWARS-compliant model, the OE and CP compliance requirements should be noted. NETWARS architecture involves the use of both OE and CP, which are the key differences in using OPNET Modeler. The developer should have a basic knowledge of creating:

- An OE compliance model
- A CP compliance model.

3.4.1 Compliance for OE Nodes

The OE node is the brain behind the OPFAC. It is responsible for traffic generation and node movement. This subsection explains how to build an OE node.

3.4.1.1 Attributes

Table 3-1 lists the minimum set of attributes an OE node must have.

Table 3-1: Attributes for OE Node

Attribute Name	Attribute Type	Description
Name	String	Specifies name of OE—*must be "OE"*
Model	String	Specifies name of model
equipment_type	Enumerated	Identifies device type—*must be "OE"*
opfacCondition	Toggle	Specifies current condition of OE node

In standard NETWARS models, the OE parent process (*oe_mgr.pr.m*) is used to declare external files that are required in the simulation. The file oe_mgr.pr.m in the process editor must be opened to find the list of the external files that have been included. The model developer may choose to declare these files in the OE or any other model. They have been included in the OE because every simulation has at least a single OE in it. Not including all the required external files may result in bind errors while running the simulation.

3.4.1.2 Initialization

The OE initializes files for writing the statistics. The "*initialize()*" function is called in the NETWARS standard models; refer to Appendix M for more information. It opens the files in write mode and writes the header row in all the files. The header row has the names of the fields in the file, separated by tabs. The following are the files that need to be initialized:

- <file_name>.ier_sent
- <file_name>.ier_rcvd
- <file_name>.ier_fail
- <file_name>.ier_block

Following is sample code (see *oe_mgr.pr.c / oe_mgr.pr.m* process model files and the external file netwars_support.ex.c for further reference) for opening the <scenario_name>.ier_sent file in write mode and writing the header row:

```
/* Get the name of the scenario */
op_ima_sim_attr_get (OPC_IMA_STRING, "net_name", scenario_name);

/* Get the file name to which the statistics must */
/* be written to                                  */
sprintf ( file_name, "%s.ier_sent", scenario_name);

/* open file <scenario_name>.ier_sent */
sent_fptr = fopen (file_name, "w");

/* write the header in the ier_sent file */
fprintf (sent_fptr, "IER_ID \t Th_ID \t IER_Src_Pf \t
    IER_Dest_Pf \t IER_Type \t IER_Class \t IER_Size \t
    IER_Start \t IER_Sent \t SE_Over \t Blocks \t
        IER_priority\n");
```

The initialization of the statistic files can be handled by provided NETWARS standard Applications Programming Interface (API) functions. Refer to Appendix L for more details about the functions available.

The OE parses an Extended Markup Language (XML) file in the scenario's folder, titled in the format <project name>-<scenario name>_traffic.xml, to get the traffic information about its parent OPFAC. This includes:

- **IER Information.** The size of the IER, name of the consumer OPFAC, IER ID, IER start/stop time, etc.
- **Threaded IER Information.** The thread start/stop time, thread ID, thread segment information, etc.

The model builder can use a series of NETWARS-provided APIs to gather this information from the XML file. The parsed IER should be stored locally for access during the simulation, such as during IER generation. The OE is also responsible for performing initialization procedures for handling of threaded IERs. The OE is required to build information regarding the threads that the OPFAC is part of. In the standard NETWARS model, the OE builds the Threaded IER Table, which contains information regarding the incoming conditions (*condition_iers*) and associated outgoing events (*reaction_iers*). The OE will use this information for every incoming part of a thread (an IER) and to fire a reaction IER if required.

Additionally, the NETWARS standard OE creates two global tables during initialization: the global information per thread and the information per thread instance. These tables contain information regarding the destination list, source OPFAC, and destination reference count. The destination list contains the list of destination nodes (which have "critical" IERs destined to them) for the thread (because a thread can have multiple destinations), and the destination reference count is the total count of these destinations. Threads can have segments (IERs) marked as critical or non-critical. If all of the IERs marked as critical reach their destinations, the simulation will mark that thread as successful even though any number of the non-critical IERs failed to reach their destination.

The NETWARS standard OE also allows IERs to be specified in various "modes": as being part of a thread only, being independently fired based on inter-arrival times, or being part of both. For a standard NETWARS OE, this is determined based on the start time of the IER—if it is set to

"THREAD," then the OE fires the IER as part of a reaction to some thread condition; otherwise, it is scheduled to fire independently (based on the inter-arrival times) *and* potentially as part of a reaction.

3.4.1.3 Traffic Generation

IERs can be assigned to OPFACs in the Scenario Builder GUI, including import from the IER database, import from text files, or manually creating the IERs. Scenario Builder writes these IERs out to the traffic XML file. The OE parses the traffic XML file and maintains a list of IERs to send by its containing OPFAC.

When it is time for this OPFAC to send an IER, the OE in the producer OPFAC finds a pair of devices in the consumer OPFAC and producer OPFAC. It can select these devices at random according to some definable constraints, or the IER definition can specify them. Then it sends a forced remote interrupt to the SE modules of the selected device in the producer OPFAC with an oe_se ICI. The SE module in the chosen producer device uses the information in the ICI to construct the IER and sends it to the consumer device.

The OE in the producer OPFAC uses the following information to find devices in the producer and consumer that can generate and accept a certain type of IER:

- **IER Classification.** The producer and consumer devices must be able to support the level of security classification required by this IER.
- **IER Traffic Type.** The producer and consumer devices must be able to support the required type of IER traffic.
- **availability_status.** The producer and consumer devices must be able to handle the new call and must not be in a "failed" state or busy with another call/transmission.
- **transport_protocol.** Depending on the transport layer specification in the IER, the producer OE associates the IER to the correct "SE." For example, if data is to be sent over TCP from a workstation, the OE will inform the relevant se module (determined by name se_tcp) to fire this IER.

To handle threaded IERs, the NETWARS standard OE maintains a *pending_reaction* list. This list consists of all the IERs that have been received (for a particular thread instance) and for which a reaction is pending. Upon the receipt of an IER (which belongs to a thread), the thread ID and associated thread instance are determined by the OE. The IER information is then inserted into the *pending_reaction* list.

The OE will match the *condition_iers* of this thread with the *pending_reaction* list. If it finds a match, then the *reaction_iers* list for this particular thread in the OPFAC is accessed and the IERs are fired. The steps for the generation (e.g., device selection, blocking) of the reaction IER are carried out as if the IER were fired independently.

If this OPFAC is one of the destinations for the thread and the incoming IER is critical, then the destination reference count for this thread instance is decremented. If the count is down to zero, then the thread is logged as being received.

3.4.1.4 Handling Background IERs

The OE is also responsible for the initialization procedures for background IER firing. Only IERs that have the traffic type set to "data" can be marked as background IERs. IERs marked as background have the IER ID starting with "BKGD." Background IERs may have explicit end devices specified on the source and destination platforms or may be left as "Auto Assigned." In case of the latter, the OE will perform the device selection procedure as it would for explicit IERs.

After the end devices are established (by one of the two methods stated here) for all background IERs being fired from this OPFAC, the OE builds what is referred to as "profile" information for background IERs to fire from all the end devices in the OPFAC. A profile represents values of packets per second and bits per second versus time. For all periods of constant utilization in the profile, the OE sends remote interrupts to the IP module in the end device to generate tracer packets. Further information regarding background traffic in OPNET can be obtained from "OPNET Online Documentation | Modeling Concepts | Simulation Project Strategies | Scalability Issues—Working with Background Traffic."

In the NETWARS standard OE model, the background traffic report file (for the flows created for the explicit IERs) is produced in the scenario directory. The OE module is required to invoke, through an API call, the IP module to generate the tracer packet. The API functions for interfacing with the IP module can be found in *app_bgutil_support.h* (in the <OPNET DIR>/<OPNET Version>/models/std/include folder in the standard OPNET installation). In particular, the function that can be used to invoke the IP module is *app_bgutil_traf_gen()*. One of the parameters that this API call accepts is the hold time—that is, the amount of time for which the specified background load is valid. The standard IP module automatically generates multiple (according to the simulation attribute, "tracer packets per interval") tracer packets during this hold time.

3.4.1.5 Interfacing with End-System Devices

The SE module in the end-system device interacts with the OE to generate the IER traffic.

The producer OE chooses a pair of producer/consumer end-system devices, based on the *equipment type* attribute on nodes. The producer OE gets the address of the consumer device and sends a remote interrupt to the chosen end-system device in the producer OPFAC, with the code set to *OE_SE_IER_SEND*. With this remote interrupt, the OE also installs an ICI with format *oe_se*. The OE must fill all the fields in this ICI before sending the remote interrupt. The format of this ICI is specified in Appendix E.

The study analyst specifies movement for OPFACs by assigning them trajectories in the Scenario Builder of the Scenario Builder GUI. The trajectory information is converted to *bearing*, *ground speed*, and *ascent* values and written out to the SDF file. The OE reads the values pertaining to its OPFAC from the Simulation Description File (SDF) file. Whenever there is a change in these values, the OE changes the *bearing*, *ground speed*, and *ascent* value attributes of its parent subnet.

Sample code for changing the *bearing*, *ground speed*, and *ascent* value attributes can be found in the oe_status process model.

3.4.1.6 Collecting Statistics

The OE node records the statistics shown in Table 3-2.

Table 3-2: Statistics Collected by OE Node

File Name	When the File Is Updated
<scenario_name>.ier_sent	The OE has found a producer device to transmit an IER.
<scenario_name>.ier_block	The OE tries to send an IER and cannot find an end-system device in the OPFAC that can transmit this IER.
<scenario_name>.ier_fail	The OE fails to send an IER because of an inability to transmit (e.g., out of retries, no SEs of appropriate type, no Decision Table, etc.).
<scenario_name>.ier_rcvd	A device receives an IER; the receiving OE records this statistic.
<scenario_name>.th_sent	The OE in the source OPFAC has found a producer device to transmit a threaded IER.
<scenario_name>.th_rcvd	All the critical IERs of a thread have been received at the destination devices.
<scenario_name>.th_fail	Some of the critical IERs of a thread have not been received at the destination devices.

3.4.1.7 Example: Constructing an OE Node

For an example, refer to Section 4, “OE Node Example.”

3.4.2 Compliance for Models for Non-Discrete Simulation (Capacity Planning)

CP in NETWARS applies analytical techniques to rapidly determine the bandwidth requirements to support specific traffic profiles and patterns. CP graphs are created in layers, and traffic is applied and performs shortest-hop routing in the order illustrated in Figure 3-6. This subsection is of interest when:

- Analytical modeling is being performed using the Deployment Editor/CP/Resource Planner
- Models are required to be built at minimum cost
- A decision regarding the “closest match” to models available in the NETWARS standard suite needs to be made

3.4.2.1 Factors of Interest during Analytical Modeling in Capacity Planning

The following properties of a model are of interest and significance when a model is used in the CP:

- How does the device affect routing of messages in the scenario? Does it perform shortest path routing? Does it treat voice and data messages differently (as far as routing is concerned)? For example, for a particular device, does it route voice messages differently

than data? Does it require circuits to be set up? Which layer does it belong to in the CP routing layer (see Figure 3-6)?

- How does the device affect the size of the message after it processes it? That is, does the message size differ when it receives on an in-port and sends on an out-port?
- What special connectivity restrictions are there for the device? Are there particular ports that connect to particular devices/device types? Do particular ports have specific message type handling capability (e.g., only data, only voice)?

3.4.2.2 Handling CP Routing

CP generates graphs in layers in the order specified in Figure 3-6. Edges belonging to the layer above are abstracted away in the current layer.

By default, all new device models encountered by the CP will be assumed to perform shortest-hop routing without the need for circuits. If circuits are required by the device being modeled, then the use of a surrogate is warranted. Possible surrogates are ATM, Tactical Satellite Signal Processing (TSSP), Promina, Multiplexer, and frame relay devices. Routing is performed in the order illustrated in Figure 3-11. For example, TSSP circuits are built and routed prior to Promina circuits. Properties to determine which layer a device belongs to are listed in Table 3-3.

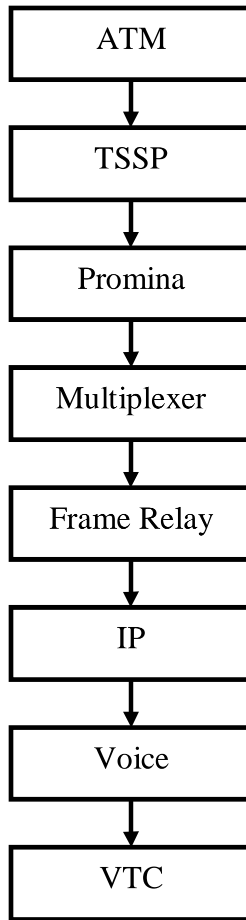


Figure 3-11: CP Layers

Table 3-3: Properties to Determine CP Layer

Layer	Attribute	Attribute Location	Acceptable Value
ATM	equipment type	on device	generic
	interface type machine type	self-description self-description	contains atm: router or switch
	interface type equipment_type	self-description on device	contains atm: Promina
TSSP	equipment type	on device	generic
	nodal mode	self-description	contains TSSP

Layer	Attribute	Attribute Location	Acceptable Value
Promina	equipment type	on device	generic or Promina
Multiplexer	equipment type	on device	generic or Promina or Multiplexer
IP	equipment type	on device	generic or radio or Joint Tactical Information Distribution System (JTIDS) or computer
	machine type	self-description	router or workstation or server or Local Area Network (LAN) or Accelerator 4000 or application proxy
	interface class	self-description	IP
Voice	equipment type	on device	generic or phone or radio or JTIDS or Media Gateway
	interface type equipment type	self-description on device	contains circuit_switched:Voice_LAN or contains circuit_switched:Voice_WAN is not Promina and is not Encryptor and is not Multiplexer
Video Teleconferencing (VTC)	equipment type	on device	generic or VTC Terminal
	interface type equipment type	self-description on device	contains circuit_switched:Voice_LAN or contains circuit_switched:Voice_WAN is not Promina and is not Encryptor and is not Multiplexer

At least one row must be satisfied to place the device in that particular layer. For example, a device belongs to the ATM layer if it is a generic device; or if its interface type contains "atm:" and it is a router or a switch; or if its interface type contains "atm:" and it is a Promina device. Misconfiguration of the attributes in Table 3-3 will cause unroutable demands.

3.4.2.3 Handling Models Modifying Message Sizes

By default, all new device models encountered by the analytical tools will be assumed to have no effect on message size. If this is not the case, for example, if the device adds a certain amount of overhead, then the use of a surrogate is warranted. Possible surrogates are KG-84, KG-194, KG-175, KIV-7, KIV-19, IP_ATM_TACLANE, and NES. Each of the devices has a user-specified overhead attribute that will increase the message size by a certain percentage. There are different connectivity restrictions enforced by these devices, so the specific properties of each should be researched when choosing the "closest match."

3.4.2.4 Handling Specific Port Selection for Alternate Links Selection in the CP

When suggesting alternate links between devices, the CP will consider the following properties of the device:

- **Does the device support the demand's traffic type?** This is determined by examining the device's packet formats and comparing them to a list of all the voice or data packet formats. These two packet format lists are built from the set of voice and data packet formats defined by the link entries in the LinkTypeMap.gdf file. If, for example, an alternate link is being suggested to help the routing of a data demand and the device does not support any of the entries in the data packet formats list, then no link will be created to that device.
- **Does the device have a free port?** If all of the ports on a device already have links connected to them, then no new links will be created for that device.
- **Is there a link that supports the device's packet format?** Once the two endpoint devices and ports are chosen, a common packet format supported by the ports on both devices will be chosen. (If there is no common packet format, then the devices cannot talk to each other and a new pair will be chosen.) An attempt will then be made to create a link that supports the common packet format. No link will be created if there is no entry in the LinkTypeMap.gdf file that supports the common packet format. For example, if the port on device A supports "ckswpkt" and "custompk" and the port on device B supports "phone_switch" and "custompk," an attempt will be made to create a link that supports "custompk." If no such link type is defined in the LinkTypeMap.gdf file, no link will be created.

Any connectivity rules beyond these are handled for a specific set of devices only. These devices are Mobile Subscriber Equipment (MSE), Promina, Promina Cell Express, and Internet Controller (INC). In each case, finding free ports with a common packet format is not sufficient when connecting those devices. Two MSE devices can be connected via their Digital Transmission Group (DTG) ports only. Promina is a similar case, because two Prominas can be connected via WAN ports only, not LAN ports. Two Promina Cell Express nodes cannot be connected directly because they require intermediate ATM devices, and two INCs can be connected via their ip_dgram_v4 ports only. If the new device has these types of restrictions, then the use of a surrogate from the above list is warranted.

3.4.2.5 Self-Description Changes

The CP requires self-description information to build various topology graphs. This is determined based on the interface class and machine type. Also, the packet format information will no longer be retrieved from the devices node model directly, but from the self-descriptions.

An example of some of the information that the CP will use in 2006-2 is as follows:

- "Radio_Wired:EPLRS INC Interface" interface type on EPLRS ports
- "Radio_Wired:Sincgars INC Interface" interface type on SINCGARS ports
- "atm:*" interface type on ATM ports
- "frame relay:*" interface type on frame relay ports
- "Circuit_Switched:*" interface type on voice-capable ports
- "router" machine type on layer 3 crypto devices and any other IP router
- "IP" interface class on router IP ports

3.5 COMPLIANCE FOR END-SYSTEM DEVICES

This subsection expects the reader to be familiar with the concepts of circuit switching. For more details on circuit switching, refer to the Subsection 3.89. End-system devices can act as sources or sinks for traffic. For IERs, the end-system device does not generate IER traffic on its own; it relies on the OE for IER generation. When the time comes to send an IER, the OE sends remote interrupts with the IER information, such as size, consumer OPFAC, etc., to the SE, using the *oe_se* ICI. Every end-system device that can send and receive IERs must include an SE module to act as the source and sink for IER traffic.

Note: A remote interrupt provides a means of inter-process communication in OPNET modeling, especially useful when two modules are not connected directly. In this case, because OE and SE modules are not connected directly, remote interrupt is used for communication between their process models.

The SE module generates packets and forwards them to lower layers. The layers below it (underneath Layer 7) are responsible for routing the IER. End-system devices can also fire non-IER (COTS) traffic. The COTS *application* and *tpal* modules implement this as the Application Layer and Transport Layer, respectively.

Note: Although there are devices (multi-homed workstations and servers) that do perform the dual tasks of serving application traffic and doing routing, these devices are excluded from the current discussion.

3.5.1 Attributes

Table 3-4 gives the minimum set of attributes that an end-system device must have.

Table 3-4: NETWARS Attributes for End-System Device

Attribute Name	Attribute Type	Description
name	String	Specifies name of device
model	String	Specifies node model (e.g., computer, DNVT)
classification	String	Specifies security classification for device; NETWARS ships with a *classification.ad.m* file which developers can use for their models.
equipment_type	Enumerated	Specifies type of equipment
availability_status	Toggle	Indicates if device is available for communication

3.5.2 Required Modules

The modules needed by devices of certain types are provided in the following tables. If one of the given protocol types is being modeled, then its corresponding modules are required. In addition, end-system devices must have at least an SE module and transmitter/receiver modules. Table 3-5 specifies the higher layer modules for a certain technology, and Table 3-6 specifies the lower layer modules. A device is built by combining the necessary modules from the two tables as specified. The SE module must have the *name* attribute set to “SE.” The OE uses the module name to identify which module/process receives the IER interrupts.

3.5.2.1 Higher Layer Modules

All end-system devices capable of sending and receiving IER traffic will have an SE module to generate the IER traffic. In addition, it may have protocol-specific modules such as the OPNET Standard (COTS) models shown in Table 3-5.

Table 3-5: Higher Layer Modules for End-System Device

Protocol Type	Required Modules
TCP	tcp (tcp_manager_v3), ip_encap (ip_encap_v4), ip (ip_dispatch, version 7.0: ip_rte_v4)
UDP	udp (rip_udp_v3), ip_encap, ip (ip_dispatch, version 7.0: ip_rte_v4)
IP	Ip (ip_dispatch, version 7.0: ip_rte_v4), ip_encap

3.5.2.2 Lower Layer Modules

The OPNET Standard (COTS) protocols shown in Table 3-6 can be used as lower layer modules. The process model in a module is specified in parentheses next to the name of the module.

Table 3-6: Lower Layer Modules for End-System Device

Protocol Type	Required Modules
Ethernet	arp (ip_arp_v4), mac (ethernet_mac_v2), point-to-point receiver module, point-to-point transmitter module
ATM	ATM_Call_Control (ams_atm_call_control), ATM_rte (ams_atm_rte), ATM_sig (ams_atm_signaling), AAL (ams_aal_disp_v3), ATM_Layer (ams_atm_layer_v3), ATM_trans (ams_atm_trans_v3), ATM_switch (ams_atm_sw_v3), point-to-point receiver module, point-to-point transmitter module
Frame relay	FRAD (frms_frad_mgr_v2), point-to-point receiver module, point-to-point transmitter module
Circuit switch	point-to-point receiver module, point-to-point transmitter module
FDDI	arp, mac (fddi_mac_v4), point-to-point receiver module, point-to-point transmitter module
Token ring	arp, mac (tr_mac_op_v2), point-to-point receiver module, point-to-point transmitter module
Serial Line Internet Proctocol (SLIP)	point-to-point receiver module, point-to-point transmitter module

Devices can be built by combining modules from the higher layer modules table with modules from the lower layer modules table. For example, an end-system device using TCP/IP over Ethernet can be built by combining the SE module and modules needed for TCP from Table 3-5 and the modules needed for Ethernet from Table 3-6. The types of transmitters and receivers to be used depend on the physical layer of the device. Transmitters and receivers can be one of three types:

- Point-to-point
- Bus
- Radio

Such an end-system device with TCP/IP over Ethernet point-to-point transceivers would appear as illustrated in Figure 3-12.

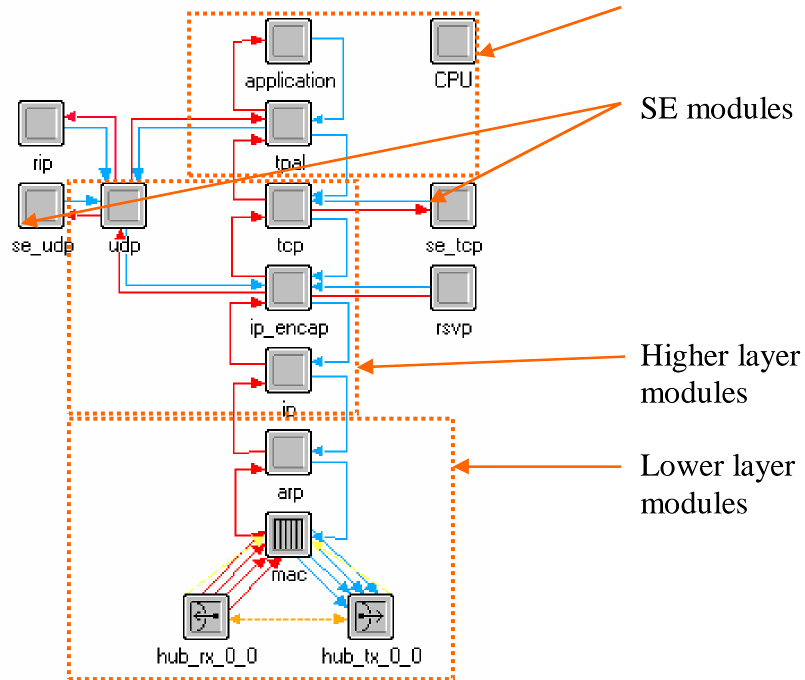


Figure 3-12: Ethernet End-System Device-Node Model

For end-system devices with radio interfaces, refer to Subsection 4.9.

It is possible to create devices with a certain transport protocol and another lower layer technology. Such an end-system device can be created by combining the modules from Table 3-5 and Table 3-6. When combining modules from the two tables, sometimes it is necessary to connect them by an interface module, shown in Table 3-7.

Table 3-7: Interface Modules for End-System Device

Higher Layer Protocol Stack	Lower Layer Protocol Stack	Interface Module Needed
TCP, UDP, IP	ATM	IPAL (ams_ipif_v4)
TCP, UDP, IP	ATM (with LANE)	arp (ip_arp_v4), LANE_IF (lms_lane_if_v3), LANE (lms_lec_v3)
TCP, UDP, IP	Frame relay	FRIPIF (frms_fr_ipif_v3)

For example, an end-system device using TCP as the transport protocol can have frame relay as the MAC technology. Such an end-system device is shown in Figure 3-13.

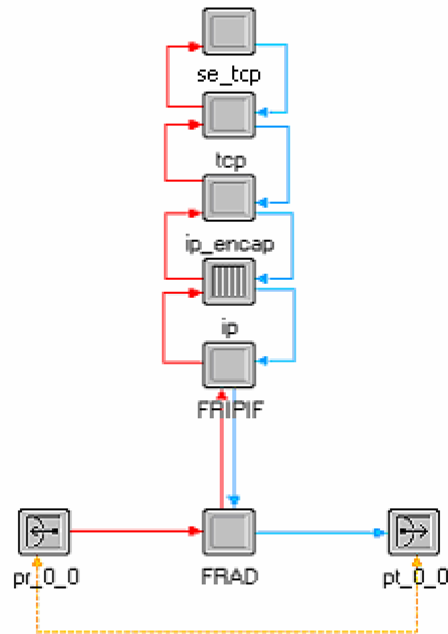


Figure 3-13: End-System Device with Frame Relay MAC Technology-Node Model

Two end-system devices that talk to each other must have the same type of transport protocol. If one of the two participating devices does not have a transport protocol, then the other must not have it either. For example, if one of them uses UDP as the transport protocol, then the other device must also use UDP as the transport protocol. An example of a valid end-system to end-system connection is shown in Figure 3-14. The connection shown between the various transmitters and receivers is *logically* bi-directional, just a way of representing bi-directional connection between the involved transmitters and receivers.

3.5.3 End-System Devices Categories

3.5.3.1 Data Traffic Only

If the end-system device supports only data traffic, then it must have the network protocol stack with the SE module, the Applications module coupled with the Transport Protocol Adaption Layer (TPAL) and Central Processing Unit (CPU) modules, or both, as explained with examples above. The SE module should have the name *se_tcp* or *se_udp*, depending on to which transport layer module each connects. For COTS traffic, the TPAL layer should be connected to the TCP and UDP modules and then to the Application module so that it serves as a go-between for the Application and transport layer modules.

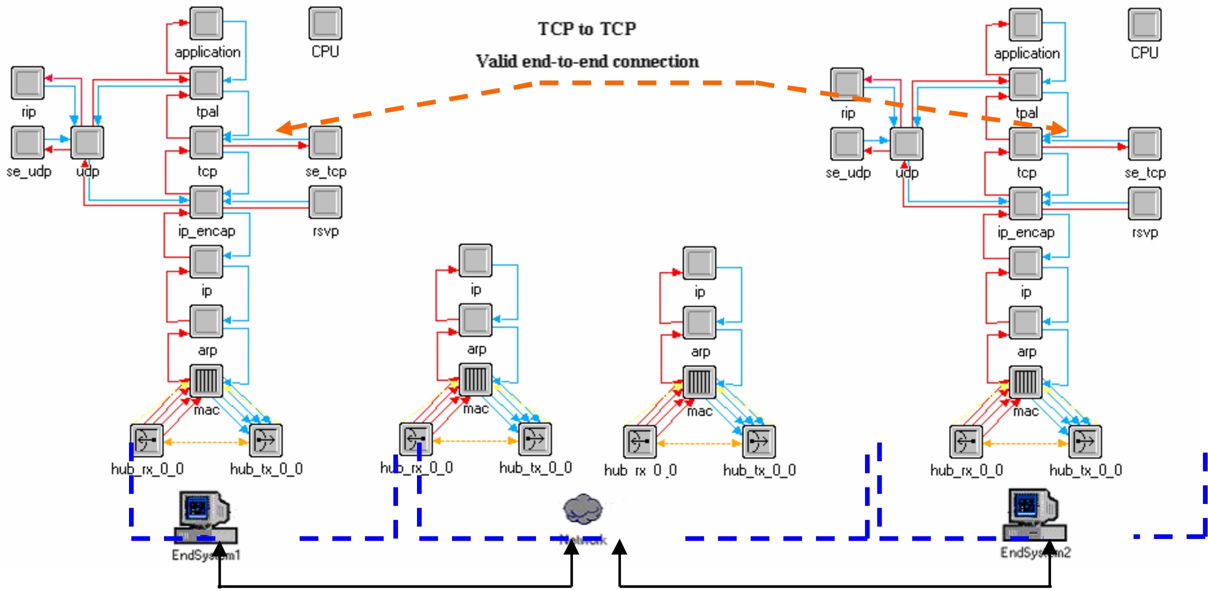


Figure 3-14: Valid End-System to End-System Connection

3.5.3.2 Circuit-Switched Voice Traffic Only

If the device supports only voice calls, it does not need the network protocol stack. In NETWARS end-system circuit-switched devices (e.g., phone), it sends out a call-setup packet (packet format *cktswpkt*) that may cause intermediate network devices to reserve bandwidth on the links and intermediate devices for the duration of the call. Refer to Subsection 3.9.

If such a purely circuit-switched device connects to other packet-switched devices, such a configuration requires use of multi-service switches (see Figure 3-15). Again, refer to Subsection 3.9 for more details.

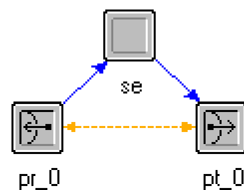


Figure 3-15: Circuit-Switched End-System Device-Node Model

However, if the voice end-system device can handle the standard voice application instead of just voice IERs, then it must include also Application, tpal, and CPU modules (see Figure 3-16).

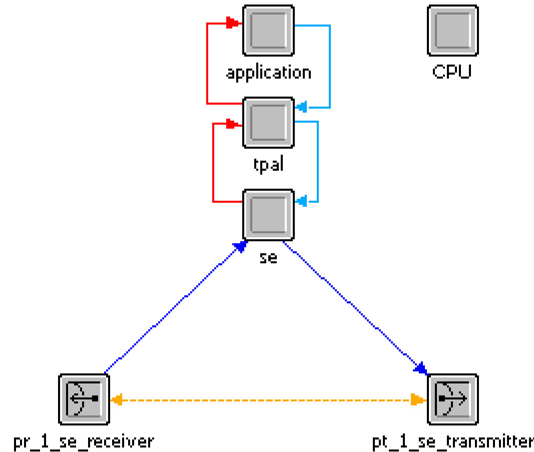


Figure 3-16: Circuit-Switched End-System Device-Voice Applications and IERs

3.5.3.3 Data and Circuit-Switched Voice Traffic

There are two ways of handling circuit switched end-system devices that handle voice applications and IERs.

If the device has only packet-switched interfaces, the voice traffic also has to be in the form of packets. The SE module generates packets at the rate specified by the OE in that OPFAC. For example, if the OE asks the end-system device to generate a call of 5-second duration every 10 seconds, the SE module in the end-system device requests to reserve the bandwidth (currently 16 Kbps fixed value for the NETWARS models) to the connecting circuit switch. These packets have to go through the entire network protocol stack like other data packets. Such a device can only be connected to other packet-switched devices. It is important for the model developer to understand the implications of developing a new end-system device like this one, including the device selection process. A new device type has to be introduced for this type of end device. Only then will the OE be able to match device type and classification and choose the end device with correct equipment type and classification. If a new end-system device type is not introduced, then the OE may try to select other devices such as computer/phone as the sink of an IER originating from this new device.

If the device has both packet-switched and circuit-switched interfaces as in Figure 3-17, one of the approaches shows that the data SE modules can send data packets over the packet-switched interfaces and the voice calls over the circuit-switched interfaces. Again, the model developer will have to introduce a new equipment type for such an end-system device for the reasons stated in the previous paragraph.

To ensure interoperability, the correct self-description information must be entered. Features such as GUI auto-addressing and port selection using Edit Ports depend on this information. Refer to the “Model Interoperability Issues” subsection and Appendix V for more information.

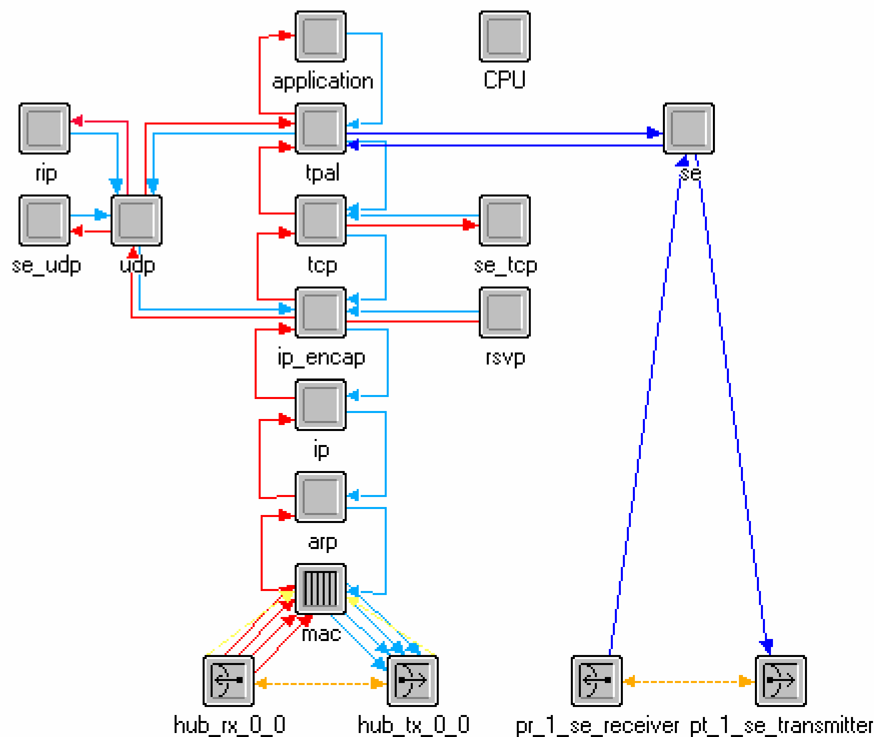


Figure 3-17: End-System Device Generating Voice and Data Traffic-Node Model

3.5.4 Interfaces and Packet Formats

When building a node model with interfaces of certain types, it is important to specify the packet formats supported on that interface. The packet formats supported by an interface depend on the MAC technology on that interface. If the created end device is to interface with a NETWARS standard model, then the developer needs to adhere to the packet formats on the MAC of the NETWARS standard model. Refer to “Appendix D: Packet Formats” for a list of the packet formats in the NETWARS standard models. Interfaces can also support custom packet formats created by a model developer.

3.5.5 Initialization

The developer must obtain handles to the statistics files for later use. This can be done through function calls documented in Appendix L: NETWARS Simulation API and Helper Functions.

3.5.6 Interfacing with Other Classes

The end-system device interfaces with other device classes as follows:

3.5.6.1 Interfacing with the OE

The SE module is responsible for all interfacing with the OE inside the OPFAC. Upon receipt of a remote interrupt from the OE with a code of *OE_SE_IER_SEND* and an ICI of type *oe_se* (see

Appendix E: Interfaces and Packet Formats), the SE will retrieve the IER information from the ICI and create an appropriate packet to send to the lower layers. For details about interrupts, refer to OPNET Modeler online documentation, Simulation Kernel manual, and the Interrupt Package chapter.

Figure 3-18 shows how the OE sends a remote interrupt to the SE.

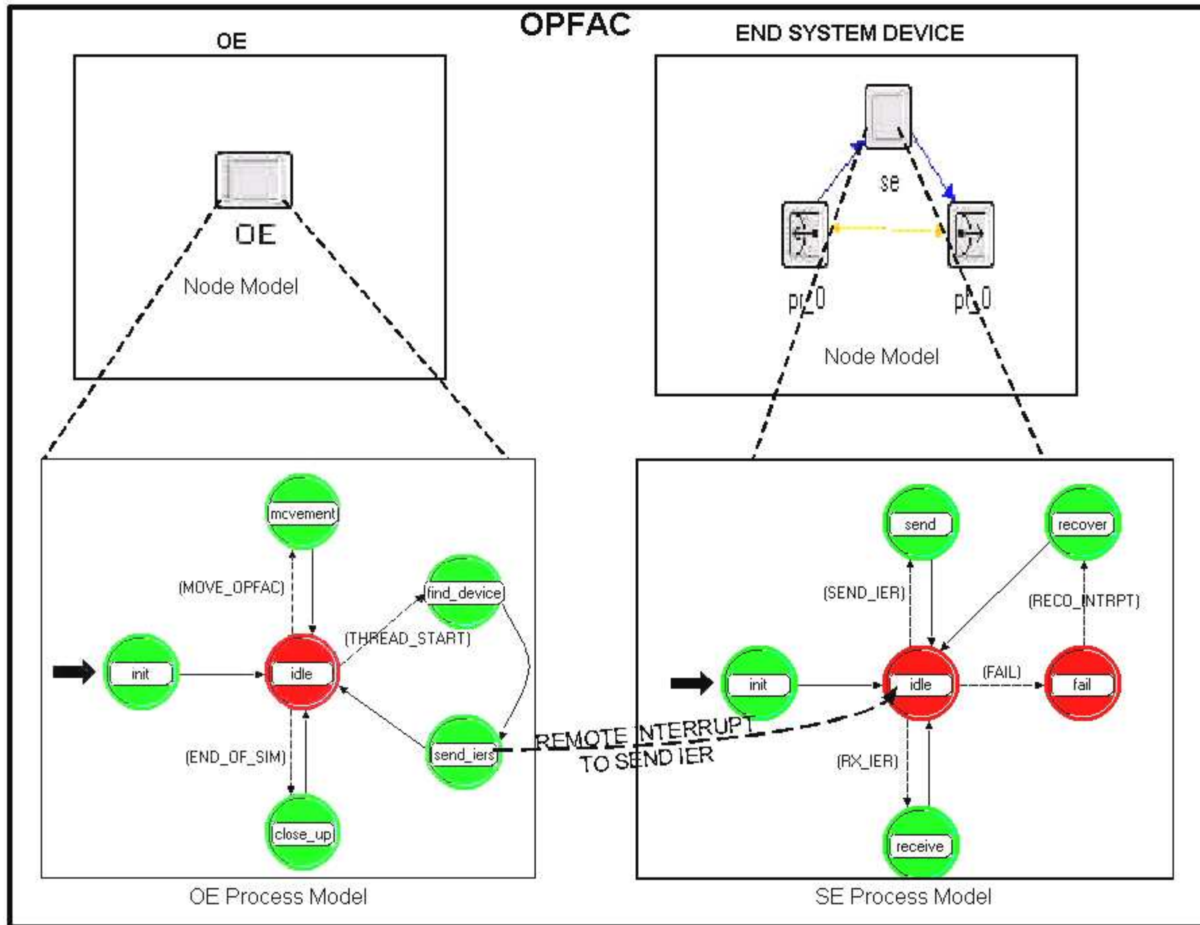


Figure 3-18: Remote Interrupt from OE to SE

3.5.6.2 Interfacing with TPAL

If the end-system device supports standard voice or VTC applications over circuit-switched environment, then it must interface with TPAL to learn when to generate application calls. Upon receipt of a remote interrupt from TPAL with a code of *TPAL_SE_APP_SEND* and an ICI of type tpal_se (see Appendix E: Interfaces and Packet Formats), the SE will generate a call for the duration specified in the ICI.

3.5.6.3 Interfacing with Networking Equipment

The end-system device is not responsible for specifying the route taken by the IER. Routing is taken care of by the networking equipment to which the end-system device is connected. The SE

module in the end-system device sends the packet down to the network protocol stack, which may encapsulate the data and sends it out on the output interface.

The *data rate* attribute on the end-system device's interfaces is typically set as "unspecified." The data rate is determined by the data rate of the link that is connected to this interface. If the *data rate* attribute is set on the interfaces, it will require the user to connect a link that has the same data rate as the value set on the interface for valid link connection. Also, the device on the other end of the link has to have either an unspecified data rate or the same data rate as specified on the interface of the first device.

3.5.7 Creating Custom Transport Protocols for End-Systems

The developer can create custom transport protocol models that can be integrated into the end-systems device model. As shown in Figure 3-17, the transport protocol models require interfacing with the other models, such as IP_Encap, TPAL, Application, OE, and SE. The custom transport protocol model can interface with the application model directly. However, it is recommended to have the transport protocol model interfaces with the application model through the TPAL model, as the primary objective of the TPAL is to provide a basic, uniform interface between application and transport layer models. Please see the "TPAL Model User Guide" for more information.

3.5.7.1 Creating Custom Transport Concerns

In order to creating a custom transport model that can be integrated into NETWARS device models, there are several concerns that developers should aware of. First, the developers must modified current SE models or develop a new SE model to interface with the new transport model. Currently, NETWARS only contains SE_udp and SE_tcp models to interface with transport protocol models. Second, new packet formats must be defined for the new transport. Developers must make sure the new formats can be able to interface with other required models. On the other hand, the new models also need to realize the packet formats that are used by other models. Lastly, developers should also need to pay attention on the ICI format. Similar to packet format, the ICI format is the most important medium for the model to communicating with each other. All newly developed and currently existing ICI format should be able to support all required models. Please see the "TCP Model User Guide" for more information.

Lastly, the OE is required to be modified to pass the IER to the newly defined transport model. In NETWARS, each IER is mapped to a corresponding transport protocol, such as TCP and UDP. The OE uses the information to pass the IER to the corresponding transport model and the associated SE model. Therefore, the OE should be modified to realize the new transport protocol.

IMPORTANT: The consequence of modifying the standard OE is serious, so please consult the NETWARS PMO before modification! Also, it is a good practice to backup the current OE model before modification.

3.5.8 Handling Background IERs

The OE node in the OPFAC sends a remote interrupt for the generation of the background IER to the IP module in the end device. Only IERs with “traffic” as “data” may be specified as background IERs. The IP module in the end node automatically generates multiple tracer packets (per the “tracer packets per interval” simulation attribute) during a period of constant hold time.

3.5.9 Handling Failure/Recovery

There are two ways of handling failure/recovery interrupts—implicitly and explicitly.

Failure/recovery can be explicitly handled by enabling the *failure interrupts* and *recovery interrupts* process attributes of the SE module’s process model and setting them to “local only.” By doing this, the SE module will receive failure/recovery interrupts whenever the *condition* attribute of the node is changed. The SE module can use these interrupts to update the *availability_status* attribute of the end-system device, preventing the OE from trying to use the failed end-system device to send IERs.

If failure/recovery is implicitly handled, once the *condition* attribute is set to “disabled,” the modules in the end-system device can no longer receive interrupts. Because the modules do not get the failure/recovery interrupts, the *availability_status* attribute of the end-system device is not updated, and the OE might try to send IERs using this failed device. In this case, the OE registers the IERs as sent, and because the end-system device is failed, it does not register these IERs as failed. If choosing this approach, additional functionality might be necessary to mark the IERs as being failed. For documentation on setting the model attributes, refer to OPNET Modeler online documentation, Modeling Concepts manual, “Process Domain” chapter, “Process Model Attributes” section. For information about handling failure/recovery, refer to Modeling Concepts manual, “Network Domain” chapter, “Modeling Node and Link Failure/Recovery” section. During failure of a device, the device flushes any queues and initiates the termination of any calls set up through it during the time of failure. The device also informs the OE to record the IER failure statistic for affected IERS during this time. The device is also required to tear down any connections it might have initiated for transmission of data IERs.

3.5.9.1 Handling Failure of Self

When the SE module in the end-system device receives a failure interrupt, it will:

- Stop transmitting and receiving IERs
- Update the *availability_status* attribute to “disabled”
- Inform the OE about the failure of the IERs generated by itself

3.5.9.2 Handling Recovery of Self

When the SE module in the end-system device receives a recovery interrupt, it must update the *availability_status* attribute to “enabled.”

3.5.10 Collecting Statistics

IER statistics are written in the Output Vector (OV) format. To enable the OE to do so, the source OE (OE of the producer OPFAC of IER) will be responsible for reporting all the IER statistics.

In the pre-2004-1 modeling architecture, the OE of the destination OPFAC (OE of the destination OPFAC of IER) reports the IER statistics. The OE used to receive a remote interrupt from either the SE of the receiving OPFAC or the OE of the source OPFAC. To write the statistics in the OV format, the OE of the source OPFAC needs to know about the reception of the IER at the destination to report the local IER statistics.

To enable this, the destination OE reports the IER statistics and the source OE is informed with a remote interrupt about the reception of the IER from the SE. This is true for the failure of an IER as well. Because the IER can be determined as failed at other locations (e.g., radio pipeline stages), this remote interrupt is generated at the source OE when an IER is either failed or received. Various interrupt codes are used (same as prior implementation) to distinguish between IER reception versus failure.

The SEs that inform their own OE about the reception are updated. For example, the "rcv_pkt" state of the se_udp process model sends a remote interrupt to its OE (destination OE) as follows:

op_intrpt_force_remote (NWC_INFORM_DEST_OE_RCVD, oe_id);

This was modified to send the interrupt to the source OE, by retrieving the source OE object ID from the IER parameters (from the source OPFAC ID).

The oe_threads process model (see Figure 3-14) is enhanced to support the threaded IER paradigm, as well as the new IER statistics architecture. The RCV_IER_INTRPT transition will occur under the following conditions:

- Interrupt sent by the destination SE (on IER reception).
- Interrupt sent by pipeline stages (on IER failure).

The process_ier state will have the same responsibilities as the current "ier_destn", but it will not perform any thread handling. If the IER belongs to a thread, then a remote interrupt to the destination OE will be sent. This interrupt will be the RCV_RXN_AT_DEST transition, which will update the thread reception statistics, if needed, and process the received reaction.

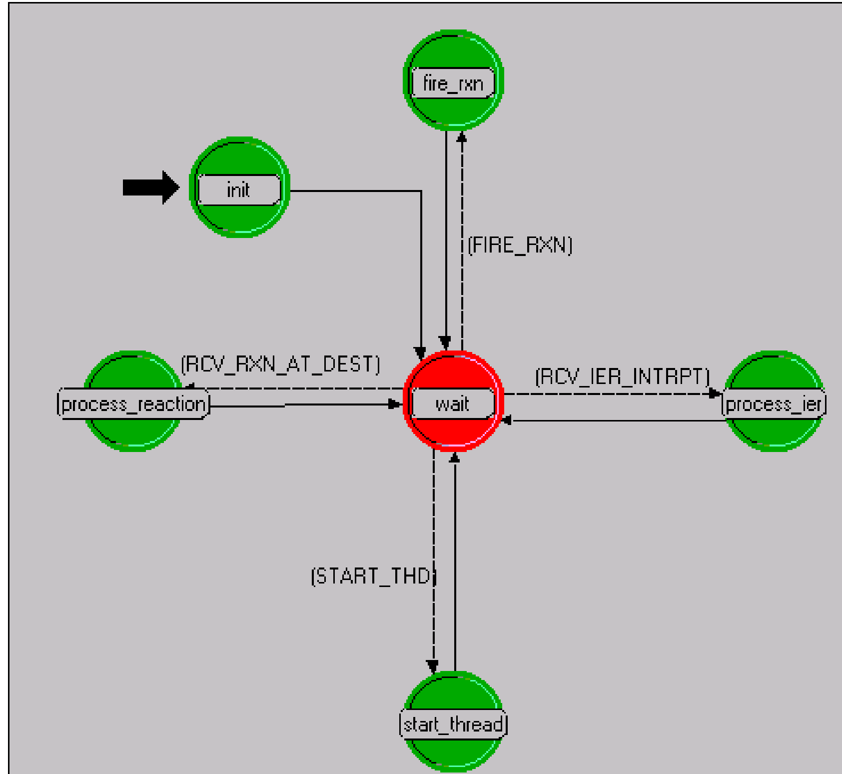


Figure 3-19: oe_threads Process Model

Table 3-8: Statistics Information Transferred by End-System Device to OE

File Name	When the File Is Updated
<scenario_name>.ier_fail	When the end-system device tries to transmit an IER and fails— • For Voice IERs, when the Acknowledgement (ACK) for a flood search is not received within a specified time-out period or when the source is busy when the ACK is received • For Data IERs, when the connection is aborted by TCP • When the end-system device fails
<scenario_name>.ier_rcvd	When a Data IER over a TCP connection or a Voice IER sent by it is received— • When it receives a Data IER over a UDP connection • When it did not get a teardown message for a voice call during the duration of the call

3.5.11 NETWARS Standard SE Models

The NETWARS standard models include seven SE models that can be used as a basis for any required device modeling, shown in Table 3-9. They provide all of the required functionality and make use of the provided APIs. Development of a new SE process model may not be required.

Table 3-9: NETWARS Standard SE Process Models

Process Model	Description
se_trafgen	Generates *data* packets in response to DATA IERs. Interfaces to TCP as the transport protocol, relying on TCP connection close messages as an acknowledgement of successful IER transmission. The parent module of this process should have the name “se_tcp.”
se_udp	Generates *data* packets in response to DATA IERs. Interfaces to UDP as the transport protocol. The parent module of this process should have the name “se_udp.”
se_sincgars	Generates *radio_packet* packets in response to both VOICE and DATA IERs and voice standard applications. The parent module of this process should have the name “se.”
se_havequick	Generates *radio_packet* packets in response to VOICE IERs and voice standard applications. The parent module of this process should have the name “se.”
dnvt_se	Generates the various circuit-switched signaling packets in response to VOICE IERs and voice standard applications. The parent module of this process should have the name “se.”
vtc_se	Generates the various circuit-switched signaling packets in response to VTC IERs and video-conferencing standard applications. The parent module of this process should have the name “se.”
se_proc_mod	Generates *data* packets in response to DATA IERs for the JTIDS radio. The parent module of this process should have the name “se.”

If a new end-system model is expected to interface with existing NETWARS standard end-system models, the matching SE process model should be used where possible. If required, a new SE process model can be developed which provides the same interfaces.

3.5.12 Example: Constructing a Computer Model

Refer to the subsection 4.4. Wired End Device Example 2 for an example.

3.6 COMPLIANCE FOR LAYER 1 NETWORKING EQUIPMENT

Layer 1 networking equipment is physical layer devices used to model repeaters, encryptors, or simply as delay elements in the network. This subsection explains how to build Layer 1 networking equipment.

There are three different types of networking equipment, depending on their functionality. The following sections explain how to build Layer 1 networking equipment.

3.6.1 Attributes

Table 3-10 describes the minimum set of attributes that a Layer 1 networking device must have.

Table 3-10: Attributes for Layer 1 Networking Equipment

Attribute Name	Attribute Type	Default Value	Description
name	String	-- *Inherent* --	Specifies name of device
model	String	-- *Inherent* --	Specifies device model, for example, CS_1005_1s_e_fr
availability_status	Toggle	Enabled	Specifies whether the device is available for communication
classification	String	Unclassified	Specifies security classification for device; NETWARS ships with a *classification.ad.m* file that developers can use for their models.
equipment_type	Enumerated	Switch, router	Identifies the device type

3.6.2 Required Modules

Layer 1 networking equipment has a processor module that accepts the packet from the receiver module, processes the packet (adds a delay, encrypts it, etc.), and sends it to the transmitter of the output interface. The type of transmitter and receiver modules will depend on the type of physical medium to which the device will be connected—bus, radio, or point-to-point.

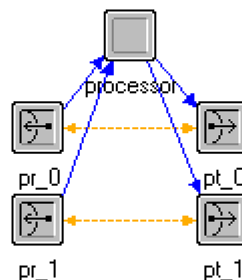


Figure 3-20: Layer 1 Networking Equipment-Node Model

3.6.3 Interfacing with Devices

Networking equipment accepts data from end-system devices and interfaces with other networking equipment to transmit it to the destination. Layer 1 networking equipment accepts packets from a device (an end-system device or other networking equipment), processes them, and sends them to the device connected on the other side. There is no routing or switching logic in these devices.

3.6.4 Handling Background Traffic

The OE module in the OPFAC and the Application model of the end workstation invoke the IP module through an API function call to generate the tracer packets. The tracer packets generated by IP are routed over the network to the IP layer in the destination SE node. In the intermediate devices in the network, the nodes may read and interpret the load represented by the tracer packet before forwarding it further in the network. The traffic represented by the tracer packet is used to artificially load the device (the queues, for example)—so explicit packets arriving at this device are processed with the load in consideration. Refer to OPNET Modeler online documentation (Modeling Concepts → Modeling Network Traffic → Working with Background Traffic) for further information.

In the NETWARS standard models that have undergone enhancement to interpret the information carried in the tracer packets, this load from the tracer packet is induced in an *input queue*. The input queue delays the explicit packet arriving before forwarding to the *output queue*. The model developer may choose to implement a similar approach to handle tracer packet loads, or to implement in some other variation, for instance, maintaining both loads (due to tracer packets and the explicit packets) in the same queue. In either way, the objective is to introduce processing delays for the explicit packets. Physical layer delays, such as transmission and propagation delays, are accounted for in the standard pipeline stages. The developer may use the TRC 170 node model as an example of a Layer 1 device capable of handling background traffic.

3.6.5 Handling Failure/Recovery

The model developer has the option of handling failure/recovery explicitly or implicitly.

If failure/recovery is handled implicitly, the OPNET Standard (COTS) failure/recovery node sets the *condition* attribute to “disabled” when the Layer 1 networking equipment fails and the device stops processing any interrupts. How this device failure/recovery is propagated to the other devices in the network depends on the routing protocols in the network. The model developer can handle the failure/recovery explicitly. By enabling the *failure interrupts* and *recover interrupts* attributes of the process model and setting them to “local only,” the process model gets an interrupt when the device fails/recovers. The following are some ways to handle failure/recovery.

3.6.5.1 Handling Failure of Self

Processing of packets should be stopped. If voice calls are set up through the Layer 1 networking device, then some cleanup might be necessary. In cases where the concept of logical links is not used, the Layer 1 networking device can do the cleanup. In cases where the logical links are

viewed by the network (like in NETWARS), the edge devices (devices at the ends of a logical link) can do the cleanup. The edge devices, such as MSE or TTC-39 switches, send keep-alive messages at regular intervals to detect the failure of the logical link. When a process running inside a device detects failure, that process (or another one that it triggers) terminates the voice calls (if any) set up over that logical link. For data packets, the process flushes the queues on the Layer 1 networking equipment.

3.6.5.2 Handling Recovery of Self

The device should re-initialize itself and prepare for processing packets again.

3.6.6 Collecting Statistics

Throughput and channel utilization statistics are written when the Layer 1 networking equipment sends out a packet. These statistics are to be written to OV using OPNET's standard Statistic package. Refer to "Appendix I: Measures of Performance in NETWARS" and "Appendix L: NETWARS Simulation API and Helper Functions" for some available function calls to write out these statistics for voice and data. These statistics may be recorded by either the edge devices or the Layer 1 networking device, depending on whether the concept of logical links is used or not. In NETWARS the concept of logical links is used, which fits well in cases where explicit packets are not modeled, for instance, during the duration of a voice call. For such cases, in NETWARS the edge devices collect these statistics. In cases where explicit packets are sent over the link through the Layer 1 networking device, for instance, data communication in NETWARS, it might be more appropriate to record these statistics at the Layer 1 device itself. For reporting statistics on the links connected (including the load represented due to background traffic), the OPNET standard pipeline stages may be used (they account for the tracer packet information received automatically). However, if the links are wireless, then the node (either edge devices in case of logical links or the Layer 1 device itself, if done otherwise) writes the statistics and accounts for the background traffic load.

3.6.7 Example: Constructing an Encryptor Model

For an example of building a Layer 1 encryptor model, refer to the "4.5. Layer 1 Device Example: Bulk Encryptor" subsection.

3.7 COMPLIANCE FOR LAYER 2 NETWORKING EQUIPMENT

Layer 2 networking equipment is devices that run a Layer 2 protocol. Switches and hubs are classified as Layer 2 networking equipment. This subsection explains how to build Layer 2 networking equipment. There are three different types of networking equipment, depending on their functionality. The following sections explain how to build Layer 2 networking equipment.

3.7.1 Attributes

Table 3-11 lists the minimum set of attributes that a Layer 2 networking device requires.

Table 3-11: Attributes for Layer 2 Networking Equipment

Attribute Name	Attribute Type	Default Value	Description
name	String	-- *Inherent* --	Specifies name of device
model	String	-- *Inherent* --	Specifies device model, for example, CS_1005_1s_e_fr
availability_status	Toggle	Enabled	Specifies if device is available for communication or not
equipment_type	Enumerated	Switch, router	Identifies device type

3.7.2 Required Modules

Table 3-12 specifies the modules required for building Layer 2 networking equipment with various interface technologies. The process model in a module is specified in parentheses next to the name of the module.

Table 3-12: Modules Needed for Various Layer 2 Protocols

Protocol Type	Required Modules
Ethernet	eth_switch (bridge_dispatch_v2), mac (ethernet_mac_v2), rx, tx
ATM	ATM_Call_Control (ams_atm_call_control), ATM_rte (ams_atm_rte) (not required for end edge devices such as ATM routers or ATM traffic sources), ATM_sig (ams_atm_signaling), AAL (ams_aal_disp_v3), ATM_Layer (ams_atm_layer_v3), ATM_trans (ams_atm_trans_v3), ATM_switch (ams_atm_sw_v3), rx, tx
Frame relay	FR_mgmt (frms_mngmt_v2), FR_trans (frms_trans_v2), FR_switch (frms_switch_v2), rx, tx
Circuit-switched (NETWARS)	circuit_switch (circuit_switch), rx, tx
FDDI	fddi_switch (bridge_dispatch_v2), mac (fddi_mac_v4), rx, tx
Token ring	stb_bridge_functions (bridge_dispatch_v2), mac (tr_mac_op_v2), rx, tx

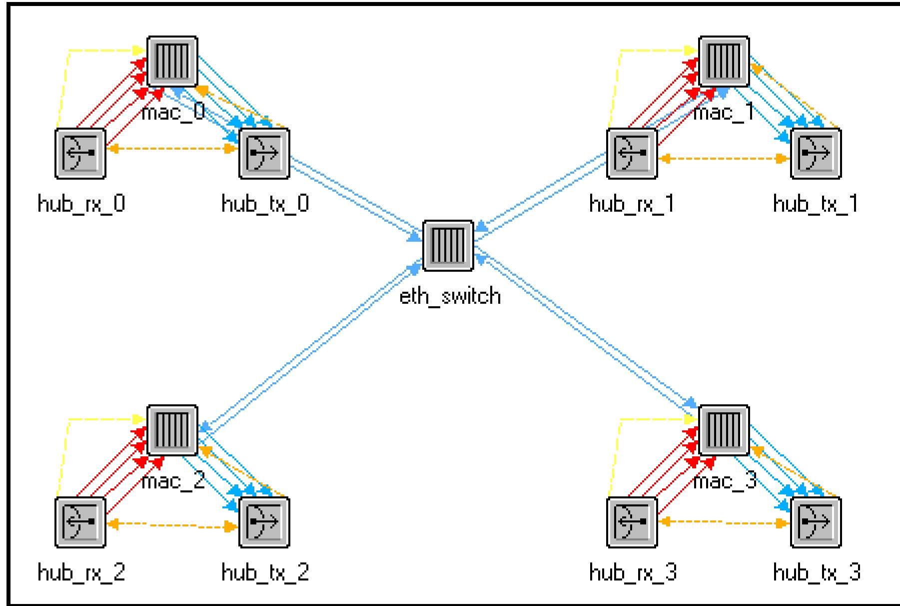


Figure 3-21: Layer 2 Networking Equipment-Node Model

Multi-service switches that have circuit-switched interfaces and packet-switched interfaces can be constructed. Table 3-13 specifies the modules needed for such devices. The process model in a module is specified in parentheses next to the name of the module.

Table 3-13: Modules Needed by Multi-Service Switch

Interface Technology	Modules Needed for a Switch with Circuit-Switched Interfaces and Packet-Switched Interfaces with the Specified Interface Technology
SLIP	voice_dispatch, voip, udp (rip_udp_v3), ip_encap (ip_encap_v4), ip (ip_dispatch, version 7.0: ip_rte_v4), SLIP interfaces
Ethernet	voice_dispatch, voip, udp, ip_encap, ip, Ethernet interfaces
Frame relay	voice_dispatch, voip, udp, ip_encap, ip, FRIPIF (frms_fr_ipif_v3), FRAD (frms_frad_mgr_v2), frame relay interfaces
ATM	voice_dispatch, voatm, ATM_Call_Control (ams_atm_call_control), ATM_sig (ams_atm_signaling), AAL (ams_aal_disp_v3), ATM_Layer (ams_atm_layer_v3), ATM_trans (ams_atm_trans_v3), ATM_switch (ams_atm_sw_v3), circuit-switch interfaces, ATM interfaces
Token ring	voice_dispatch, voip, udp, ip_encap, ip, arp (ip_arp_v4), mac (tr_mac_op_v2), token ring interfaces
FDDI	voice_dispatch, voip, udp, ip_encap, ip, arp, mac (fddi_mac_v4), FDDI interfaces

3.7.3 Initialization

The switch module in the Layer 2 networking equipment will perform the following initialization steps:

- The switch module will register itself in the process registry with the following attributes:
  - Location (string)
  - Protocol (string)

- The switch module must build switch tables with entries corresponding to its neighboring switches. One way of building such tables is by using spanning trees. The code for building spanning trees can be re-used from the OPNET Standard (COTS) models.

3.7.4 Interfacing with End-System Devices and Networking Equipment

Networking equipment accepts data from end-system devices and sends the data to the destination end-system devices. The routing information available to the networking equipment is local; it includes information only about devices that are connected to it directly and through other lower layer (Layer 1) networking equipment. If the Layer 2 networking device provides circuit capabilities, additional attributes will be required. These are documented in Subsection 8.

3.7.5 Supported Protocols

Depending on the MAC layer technology needed by the device, the model builder must use the corresponding protocol stack. For creating an Ethernet switch, the model builder must have the OPNET Ethernet protocol stack so that the switch will be interoperable with OPNET Standard (COTS) Ethernet device models. OPNET provides support for devices running the following MAC layer protocols:

- Ethernet
- Token ring
- FDDI
- Frame relay
- SLIP
- DSL
- Integrated Services Digital Network (ISDN)
- 802.11 wireless LAN.

3.7.6 Handling Background IERs

The OE node in the end OPFAC invokes the IP layer to generate tracer packets for the background IERs. The tracer packet is routed over the network to the destination end device. The intermediate network devices can read information from the tracer packet and load the device for the explicit packets arriving at the node. The NETWARS standard nodes perform this loading on an input queue before forwarding the packet to the output queue. It should be noted that background traffic could be enabled only for IERs with traffic type as data. Examples of Layer 2 devices handling background IERs in the NETWARS standard models are MSE/TTC-39 switches and the Promina switch.

One issue the model developer needs to be aware of is the requirement to appropriately modify the packets/second information in the tracer packet before performing an en-queue on the background aware buffer. The reason this may be required is that if the Layer 2 device performs segmentation and reassembly, then the packets/second information in the tracer packet are to be appropriately modified, because by default the information carried is the IP datagram packets/ second information.

Additionally, the model developer may be required to modify the bits/second information (for the processing rate from the buffer). This might be required, for example, if the device is capable of handling both voice and data—in which case the available bandwidth is dependent on the number of voice calls in progress. For this, the function *oms_buffer_bgutil_modify_average_rate (…)* may be used, which is defined in *oms_buffer_bgutil.ex.c*.

3.7.7 Handling Failure/Recovery

The manner in which Layer 2 networking equipment handles failure/recovery depends on the type of protocol it is running. The model developer has the option of handling failure/recovery explicitly or implicitly.

If failure/recovery is handled implicitly, the failure/recovery utility sets the *condition* attribute to “disabled” when the Layer 2 networking equipment fails and the device stops processing any interrupts. How this device failure/recovery is propagated to the other devices in the network depends on the routing protocols in the network.

If handled explicitly, by enabling the *failure interrupts* and *recover interrupts* attributes of the process model and setting them to “local only,” the process model gets an interrupt when the device fails/recovers. The following are some ways to handle failure/recovery.

3.7.7.1 Handling Failure of Self

- Flush the queue modules (if the Layer 2 networking equipment has any).
- Write out failure statistics for the voice IERs (if any).

3.7.7.2 Handling Recovery of Self

- Send update messages to the neighboring Layer 2 networking equipment.
- Rebuild the spanning tree.

3.7.8 Collecting Statistics

Throughput statistics are written when a packet is sent out, and queue size statistics are collected when a packet arrives or leaves a queue module in Layer 2 networking equipment. The traffic-dropped statistics are written out every time a packet is dropped from a queue of Layer 2 networking equipment. These statistics are to be written to vector files using OPNET’s standard Statistic package. Refer to “Appendix L: NETWARS Simulation API and Helper Functions” for some available function calls to write out data and voice throughput statistics.

3.7.9 Example: Constructing a Multi-Service Switch

For an example, refer to the “4.6. Layer 2 Device Example: Multi-Service Switch” subsection.

3.8 COMPLIANCE FOR LAYER 3 NETWORKING EQUIPMENT

Layer 3 networking equipment is devices that run a network layer protocol. Routers are classified as Layer 3 networking equipment. Every interface of this device has a different network address. This subsection explains how to build Layer 3 networking equipment. The current NETWARS standard device models support only IPv4 as a Layer 3 network protocol. All of the subsections of this *Guide* dealing with Layer 3 protocols document the usage of IP.

There are three different types of networking equipment, depending on their functionality. The following sections explain how to build Layer 3 networking equipment.

3.8.1 Attributes

Table 3-14 lists the minimum set of attributes that Layer 3 networking equipment must have.

Table 3-14: Attributes for Layer 3 Networking Equipment

Attribute Name	Attribute Type	Default Value	Description
name	String	-- *Inherent* --	Specifies name of device
model	String	-- *Inherent* --	Specifies device model, for example, CS_1005_1s_e_fr
availability_status	Toggle	Enabled	Describes if equipment is available or has failed
equipment_type	Enumerated	Switch, router	Describes device type
ip addr index	Integer	0	Index used for IP addressing and dynamic routing. This attribute is set on the streams into and out from the IP module.

3.8.2 Required Modules

The only higher layer protocols supported by Layer 3 networking equipment are TCP, UDP, Resource Reservation Protocol (RSVP), and various routing protocols over IP. But the networking equipment can have interfaces running different MAC layer technologies. Table 3-15 specifies the higher layer modules required for Layer 3 networking equipment.

Table 3-15: Higher Layer Modules for Layer 3 Networking Equipment

Protocol Type	Required Modules
TCP/UDP/routing protocols	tcp (tcp_manager_v3), udp (rip_ud_v3), rip (rip_v3), eigrp (eigrp), igrp (igrp), bgp (bgp), ospf (ospf_v2), rsvp(rsvp), ip_encap (ip_encap_v4), ip (ip_dispatch, version 7.0: ip_rte_v4)

All OPNET Standard (COTS) router models support a set of routing protocols—BGP, EIGRP, IGRP, OSPF, and RIP. It is possible to have different routing protocols running on different interfaces in the network. To make sure that all the OPNET Standard (COTS) routing protocols are supported, it is necessary to have all the routing protocol modules in Layer 3 networking equipment. The required modules specified above are interconnected as shown in Figure 3-22.

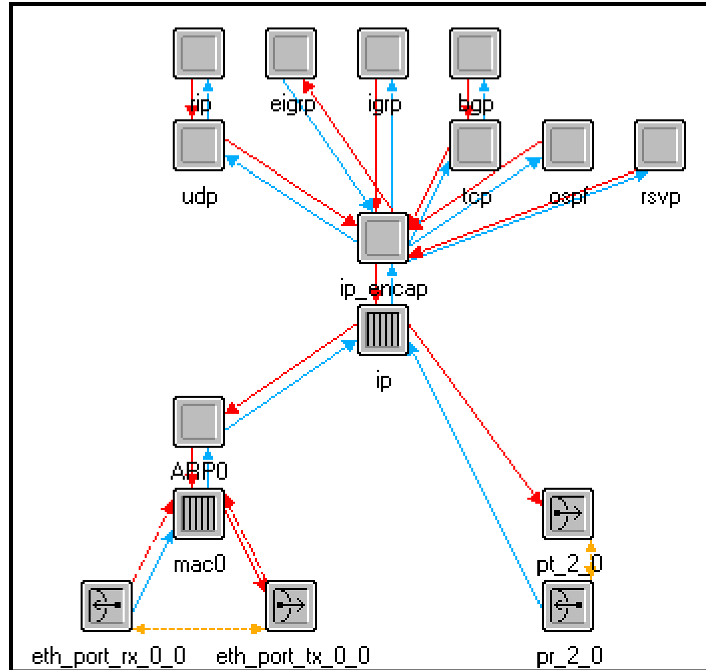


Figure 3-22: Layer 3 Networking Equipment-Node Model

Table 3-16 specifies the possible types of interfaces for the networking equipment and the modules needed for each interface technology. The process model in a module is specified in parentheses next to the name of the module.

Table 3-16: Required Modules for Various Interface Technologies

Protocol Type	Required Modules
Ethernet	arp (ip_arp_v4), mac (ethernet_mac_v2), rx, tx
ATM	ATM_Call_Control (ams_atm_call_control), ATM_rte (ams_atm_rte), ATM_sig (ams_atm_signaling), AAL (ams_aal_disp_v3), ATM_Layer (ams_atm_layer_v3), ATM_trans (ams_atm_trans_v3), ATM_switch (ams_atm_sw_v3), rx, tx
Frame relay	FRAD (frms_frad_mgr_v2), rx, tx
FDDI	arp, mac (fddi_mac_v4), rx, tx
Token ring	arp, mac (tr_mac_op_v2), rx, tx
SLIP	rx, tx

It is possible to create devices with a certain transport protocol and another lower layer technology. Such networking equipment can be created by combining the modules from Table 3-15 and Table 3-16. When combining modules from the two tables, sometimes it is necessary to connect them by an interface module, as shown in Table 3-17.

Table 3-17: Interface Modules for Layer 3 Networking Equipment

Higher Layer Protocol Stack	Interface Technology	Interface Module Needed
TCP, UDP, IP	ATM	IPAL (ams_ipif_v4)
TCP, UDP, IP	Frame relay	FRIPIF (frms_fr_ipif_v3)

3.8.3 Handling Security Classification

When connecting devices with different security classification levels, it is the responsibility of the study analyst to connect them in such a way that messages traverse only networks with the proper level of security classification (see Figure 3-23). Another option is to use encryption devices. For example, when passing classified data over an unclassified network, the message must be encrypted end to end. Lack of these encryption devices causes the simulation to assume that the encryption is present implicitly. The advantage of actually modeling the encryption devices would be increased fidelity for delay and throughput statistics.

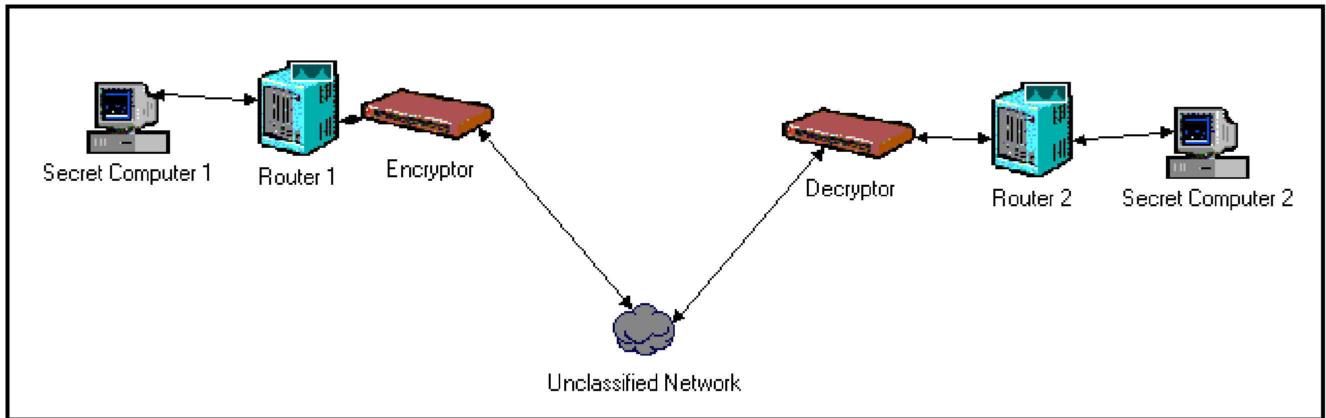


Figure 3-23: Networks with Different Security Classification Levels

3.8.4 Interfacing with End-System Devices and Networking Equipment

Networking equipment accepts data from end-system devices and routes the data to the destination end-system devices. The *data rate* attribute on the networking equipment's interfaces is typically set as "unspecified." The data rate is determined by the data rate of the link that is connected to this interface.

Networking equipment builds the routing information from routing updates sent by other networking equipment in the network that are directly connected to it. If the model developer uses custom IP routing protocols in the Layer 3 networking equipment, then when the networking equipment receives routing update messages it must update entries in the IP common route table using calls to the following functions:

- Ip_Cmn_Rte_Table_Entry_Add ()
- Ip_Cmn_Rte_Table_Entry_Delete ()

3.8.5 Supported Protocols

Depending on the MAC layer technology needed by the device, the model builder must use the corresponding protocol stack. For creating an ATM switch, the model builder must have the OPNET ATM protocol stack so that the switch will interoperate with OPNET Standard (COTS) ATM device models. OPNET provides support for devices running the following protocols:

- Ethernet
- ATM
- FDDI
- Frame relay
- SLIP
- Token ring
- DSL
- ISDN
- IEEE 802.11 wireless LAN.

The following routing protocols are supported by OPNET Standard (COTS) networking equipment:

- RIP
- IGRP
- EIGRP
- BGP
- OSPF
- Static routing.

Additional routing protocols can be added; see the following subsection for more information on this process.

3.8.6 Creating Custom Routing Protocols for IP

This subsection enumerates the required steps for writing custom IP routing protocols and the issues involved with their use in a network with other routing protocols.

3.8.6.1 Implementing a Custom Routing Protocol

The custom routing protocol must register itself as an IP higher layer protocol with a call to the function Ip_Higher_Layer_Protocol_Register () using the name of the protocol and an integer with a value above 500.

During its initialization, the custom routing protocol must also call the function Ip_Cmn_Rte_Table_Custom_Rte_Protocol_Register (), passing the name of the routing protocol as a string. This will return a routing protocol ID to be used in subsequent route table function calls. The protocol ID for a custom routing protocol has a value greater than 100.

The custom routing protocol will receive a remote interrupt with a code "*IPC_EXT_RTE_REMOTE_INTRPT_CODE*" upon initialization of the IP process model. At this time the interface table and routing table can be accessed via the process registry.

The custom routing protocols access the IP common routing table using calls to the following functions:

- *Ip_Cmn_Rte_Table_Entry_Add()*
- *Ip_Cmn_Rte_Table_Entry_Delete()*
- *Ip_Cmn_Rte_Table_Entry_Update()*

Entries to the route table will be made through calls to the function *Ip_Cmn_Rte_Table_Entry_Add()*, with updates provided through the functions *Ip_Cmn_Rte_Table_Entry_Update ()* and *Ip_Cmn_Rte_Table_Entry_Delete()*. The existing entries can be queried through calls to the *Ip_Cmn_Rte_Table_Entry_Exists()* and *ip_cmn_rte_table_lookup()* functions.

These functions are defined in the external file <opnet_dir>\<rel_dir>\models\std\ip\ip_cmn_rte_table.ex.c, and the function prototypes are in <opnet_dir>\<rel_dir>\models\std\include\ip_cmn_rte_table.h, where <opnet_dir> is the folder where OPNET is installed and <rel_dir> is the release directory (e.g., 12.0.A).

3.8.6.2 Issues with Using Custom Routing Protocols

There are some issues involved with using the custom routing protocols that the model developer may address in the following suggested manner.

3.8.6.3 Lack of Route Redistribution Capability

Some routing protocols might have a lack of route redistribution capability. This means that routes determined by these protocols cannot be used by other routing protocols and vice versa. Route redistribution is the process by which routes determined by all routing protocols running within a router node can be shared among each other.

This issue can be avoided in two ways:

- Modifying functions in the following files to include this capability:
  - *Ip_dispatch.pr.m* (version 7.0: *ip_rte_v4.pr.m*)
  - *ip_rte_v4.h*
  - *ip_cmn_rte_table.ex.c*
  - *ip_cmn_rte_table.h*
- Running the custom routing protocol on all interfaces in the network.

3.8.6.4 Lack of Route Table Import/Export Capability

The OPNET Standard (COTS) routing protocols allow the routes to be exported at the end of a simulation and to be re-imported into the network for subsequent simulations. This reduces the simulation run time. The model developer can add this functionality to the custom routing protocol if desired.

3.8.7 Handling Background IERs

Tracer packets (for the background IERs) are generated from the IP module of the end data systems. Intermediate network devices could read and load the device queues as per the traffic information specified in the tracer packet. If the Layer 3 device is an IP device, then no modification to the standard IP process model is required, because the standard IP module is capable of loading the device as per information represented in the tracer packet.

3.8.8 Handling Failure/Recovery

The manner in which Layer 3 networking equipment handles failure/recovery depends on the type of routing protocol it is running. The model developer has the option of handling failure/recovery explicitly or implicitly.

If failure/recovery is handled implicitly, this sets the *condition* attribute to “disabled” when the Layer 3 networking equipment fails and the device stops processing any interrupts. How this device failure/recovery is propagated to the other devices in the network depends on the routing protocol.

If handled explicitly, by enabling the *failure interrupts* and *recover interrupts* attributes of the relevant modules and setting them to “local only,” the process model gets an interrupt when the device fails/recovers. The following are some possible ways to handle failure/recovery.

3.8.8.1 Handling Device Failure

If the failure of the device itself is to be handled explicitly, then on receiving the failure interrupt, the appropriate module may flush the queues.

3.8.8.2 Handling Device Recovery

If the recovery of the device itself is to be handled explicitly, then on receiving the recovery interrupt, update messages may be sent to the neighboring routers to indicate that this networking equipment has recovered.

3.8.8.3 Handling Failure of Neighboring Layer 3 Equipment

This failure may be handled implicitly by the routing protocol, which may update the routing table entries that have routes via this failed router. This can be done by the routing protocol.

3.8.8.4 Handling Recovery of Neighboring Layer 3 Equipment

Similarly, this is also handled implicitly. On receiving update messages from the neighboring networking equipment that recovered, the networking equipment may recompute routes to all destinations through the recovered node and update the routing tables if the new route is better than the existing routes.

3.8.9 Collecting Statistics

Throughput statistics are written when a packet is sent out, and queue size statistics are collected when a packet arrives or leaves a queue module in Layer 3 networking equipment. The traffic-dropped statistics are written out every time a packet is dropped from a queue of Layer 3 networking equipment. These statistics are to be written to vector files using OPNET's standard Statistic package. Refer to "Appendix L: NETWARS Simulation API and Helper Functions" for some available function calls to write out data and voice throughput statistics.

3.9 COMPLIANCE FOR DEVICES WITH CIRCUIT-SWITCHED TECHNOLOGY

Circuit-switched voice devices are capable only of generating or handling voice calls. In general, this *Guide* offers a great deal of latitude to the model developer wishing to develop circuit-switched data models. To promote interoperability within the very generic notion of circuit-switched voice communications, however, the *Guide* has developed the following standards for circuit-switched voice components. There are no components that are classified purely as circuit-switched devices. Circuit-switched devices can be end-system devices, generating and receiving calls, or they can be networking equipment, switching calls between source and destination. Depending on whether they are end-system devices or networking equipment, the model developer must refer to the appropriate subsections, and make sure the device performs the necessary functions specified in those subsections.

Note that the circuit-switched models employed in the NETWARS standard models contain additional functionality beyond the OPNET Specialized (COTS) Circuit-Switched model library. As such, the Specialized Circuit-Switched model cannot be used in NETWARS.

3.9.1 Attributes

Table 3-18 lists the minimum set of attributes that an end-system device capable of generating circuit-switched calls should have.

Table 3-18: Required Attributes-Circuit-Switched End-System Device

Attribute Name	Attribute Type	Description
Call bandwidth	Double	Specifies the call bit rate originating from this end system
Maximum calls allowed	Integer	Specifies the maximum number of voice calls the device can support simultaneously

If Layer 2 networking equipment is to be capable of handling circuit-switched calls, it requires the attributes listed in Table 3-19.

Table 3-19: Required Attributes-Circuit-Switched Layer 2 Networking Equipment

Attribute Name	Attribute Type	Description
MSE topology mask	String	Differentiates a Layer 2 circuit-switched device from a Layer 3 router

3.9.2 Initialization

The switch model will construct a list of end-system devices connected to it. The switch model also constructs logical links with its neighboring switches. These logical links are used while performing voice call routing. Logical links are an abstraction for the path between two neighboring circuit-switched devices. They do not exist in the real world, but are NETWARS-specific internal data constructs that keep track of available voice channels and/or available bandwidth on the entire route between two neighboring circuit-switched devices.

3.9.3 Routing in Circuit-Switched Devices

This subsection describes the NETWARS standard implementation of routing, using a Flood Search Routing protocol. When a circuit-switched device makes a call to a destination, it initially sends a query packet to the switch to which it is connected. The switch checks if the destination device is connected to it. If not, it forwards the query packet to all the connected switches in the route to the destination. The query packet is forwarded to the next hop until it reaches the switch to which the destination is connected. A timer is scheduled on the switch to wait for the ACK.

When the query packet arrives at the switch to which the destination is connected, the switch sends an ACK back to the source end-system device. The path taken by the ACK is the chosen path. As soon as the source gets the acknowledgment packet, it gets the link where the call ACK came from. If bandwidth is available on the link, then the switch reserves it, otherwise it bumps a lower priority call and writes an IER failure statistics. It also schedules an interrupt to free bandwidth after the call duration and sends a TEARDOWN message. If there is no bandwidth available or calls to bump, the packet is dropped.

The ACK packet reserves bandwidth on the links from the source to the destination. Once the call is set up, no other packets are sent for the duration of the call. The call is released and the reserved bandwidth is freed as the call duration timer expires.

Functions for route structure handling have to be created. These functions allow for route copying, route destroying, creating pooled memory for route structures, and route reversing. The external file *flood_search_routing.ex.c* contains functions for route structure handling, with the function prototypes included in a header file *flood_search_routing.h*.

3.9.4 Circuit-Switched Links

NETWARS standard circuit-switched models conceptualize links over Layer 1 transmission devices as being “logical links.” One of the reasons for doing so is that there are no actual packets that are sent over the network to model the voice call. These circuit-switched devices maintain information about the links (which may be either wired or wireless) between the intermediate Layer 1 devices. This information is built up during initialization by the edge circuit-switched devices through a topology walk. This information is used when link voice throughput and channel utilization statistics are written out, as well as during the call setup process.

3.9.5 Interfacing with Packet-Switched Networks

When a circuit-switched device has to connect to a packet-switched network, it has to go through an intermediate device that is capable of interfacing with both circuit-switched and packet-switched networks. Such intermediate devices are called multi-service switches (e.g., media gateways).

When a multi-service switch receives a request to place a circuit-switched voice call over a packet-switched network, it performs the following operations to interface with an IP network:

- These multi-service switches publish their loopback IP address in the process registry; every other multi-service switch can use this IP address to reach this gateway.
- When doing flood search routing, an ingress multi-service switch looks at all the multi-service switches in the network and will pick only those that have advertised having a route to the destination phone, as shown in Figure 3-24. Once it knows the gateways that have a route to the destination phone, the multi-service switch opens UDP connections to the loopback IP addresses of these multi-service switches (obtained from the process registry) and sends the call query packet (encapsulated in an IP packet) to them. It also records its loopback IP address in the call query packet.
- An egress multi-service switch should record its own loopback IP address, de-capsulate the IP packet, and flood the query in the circuit-switched network.
- Only the ingress and egress loopback IP addresses are needed; routing in the data network will be done as usual by IP with routing protocol.
- ACK follows the reverse path.
- “Call estab” follows the recorded path by query packet.
- Once the call is established, bandwidth is reserved and utilization numbers are updated in the circuit-switched network only (no bandwidth reservation or link utilization update will happen in IP and ATM networks). Once bandwidth is reserved and link utilization numbers are updated in the circuit-switched network, the multi-service switch starts generating voice packets according to the codec information configured to load the IP network.
- Once the call is completed, multi-service switches in the route are notified to stop generating voice packets.

In case of call failure/bumping, bandwidth is released and link utilization numbers are updated in the circuit-switched network, and multi-service switches in the route are notified to stop generating voice packets.

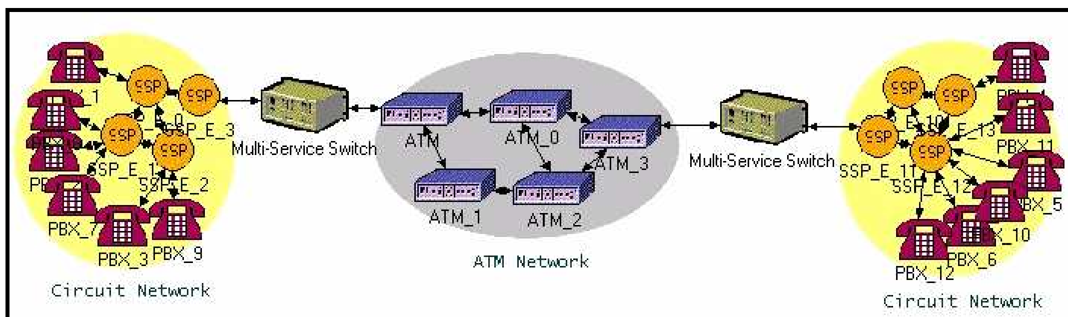


Figure 3-24: Circuit-Switched and Packet-Switched Network Intercommunication

3.9.6 Handling Background IERs

Because explicit packets are not generated for a voice call in a circuit-switched network, background IERs are handled similarly to explicit IERs in a pure circuit-switched network. However, if a voice IER is going through multi-service switches (through an IP/ATM network), background IERs are handled differently than explicit IERs. Multi-service switches generate packets to load the IP/ATM network for voice IERs. For explicit IERs, multi-service switches generate explicit voice packets. However, for background IERs, multi-service switches generate tracer packets to load the IP/ATM network (OPNET hybrid simulation model).

3.9.7 Handling Failure/Recovery

To be able to handle failure/recovery, the processor modules in the circuit-switched devices must have their *failure interrupts* and *recovery interrupts* attributes "enabled" and set to "local only."

3.9.7.1 Handling Failure of a Circuit-Switched Device in the Network

When a circuit-switched device fails, it should clear all the calls and release the channel (bandwidth) for the call. The IER statistics have to be written out and the IERs have to be marked as failed. When a NETWARS Layer 2 networking device with circuit-switched capabilities handling voice calls fails, it informs its neighboring devices, which in turn write out the IER statistics. For voice calls, in NETWARS a global list of bumped IERs is maintained to avoid race conditions like multiple switches trying to mark the same IER as failed.

3.9.7.2 Handling Recovery of a Circuit-Switched Device in the Network

The device does not do anything special on receiving this recovery interrupt.

3.9.8 Collecting Statistics

The following statistics are relevant to circuit-switched models:

- Link level
  - Link statistics are updated for the voice traffic also
- Channel level
  - Voice channel utilization
- Circuit level
  - Circuit throughput (also updated for voice)
  - Circuit utilization (also updated for voice)
- Circuit-switched node-level statistics
  - Bandwidth reserved (bits per second)
  - Total calls blocked
  - Total calls switched
  - Active calls
  - Low-priority calls dropped
- End-system node level statistics
  - Call setup time (seconds)
  - Active calls

- Total calls connected
- Total calls disconnected
- Total calls generated.

3.10 COMPLIANCE FOR WIRELESS INTERFACES

The end-system or network equipment devices in NETWARS can support both wired and RF radio interfaces. In addition to the requirements for the class of device being built, radio interfaces require additional requirements, which are documented in this subsection.

3.10.1 Attributes

A radio device needs the attributes shown in Table 3-20.

Table 3-20: Additional Attributes for Radio Devices

Attribute Name	Attribute Type	Default Value	Description
antenna_pattern	Typed file	Isotropic	Specifies the antenna pattern to be used on the radio device.
Modulation (per channel)	Typed file	—	Specifies the modulation table to be used to look up the bit error rate as a function of the signal-to-noise ratio.
Power (per channel)	Double	—	Specifies the transmitting power for the radio transmitter; this attribute will be promoted from the channel attribute of the transmitter to the node level.
Processing gain (per channel)	Double	—	Specifies the processing gain for the radio receiver; this attribute will be promoted from the channel attribute of the receiver to the node level.
min_frequency (per channel)	Double	—	Specifies the base transmitter/receiver frequency for a channel; this attribute will be promoted from the channel attribute of the transmitter/receiver to the node level.
Bandwidth (per channel)	Double	—	Specifies the transmitter/receiver bandwidth for a channel; this attribute will be promoted from the *channel* attribute of the transmitter/receiver to the node level.
data_rate (per channel)	Double	—	Specifies the data rate on the channel in the node; this attribute must be promoted.
net_id[4] (per radio tx and rx module)	Integer	–1	When two radios share the same net_id, they are in the same radio network. This extended attribute must be promoted.
Spreading code (per channel)	Double	0	Specifies the frequency hop group to which the radio belongs.

Apart from these attributes, the pipeline stage[5] attributes shown in Table 3-21 and Table 3-22 also must be set on the radio transmitter and receiver modules. The pipeline stage attributes are

[4] This attribute is particularly important in radio broadcast networks where all the radios in the same broadcast network will have the same net_id. Also, radios connected by the Line of Sight link will have the same net_id.

required by OPNET's radio pipeline stages. Most of the attributes defined in Table 3-20 are available on the radio transmitters and receivers. They should be promoted to the node level to use the Scenario Builder features to create radio links and broadcast networks.

3.10.1.1 Transmitter Pipeline Stage Attributes

All of the attributes shown in Table 3-21 are of type Typed File.

Table 3-21: Pipeline Stage Attributes on Radio Transmitter

Pipeline Stage Attribute Name	Default Value	Description
txdel model	dra_txdel	Computes the transmission delay associated with the transmission of a packet.
rxgroup model	dra_rxgroup	Determines the possibility of radio interaction between a transmitter channel and a receiver channel.
chanmatch model	dra_chanmatch	Characterizes the type of interaction between a transmitter channel and a receiver channel.
closure model	dra_closure	Dynamically determines the ability of a transmitter channel to reach a receiver channel.
tagain model	dra_tagain	Computes the antenna gain provided by the transmitter's antenna module in the direction of a particular receiver.
propdel model	dra_propdel	Computes the propagation delay associated with the transmission of a packet.

3.10.1.2 Receiver Pipeline Stage Attributes

All of the attributes shown in Table 3-22 are of type Typed File.

Table 3-22: Pipeline Stage Attributes on Radio Receiver

Pipeline Stage Attribute Name	Default Value	Description
ragain model	dra_ragain	Computes the antenna gain associated with the receiver's antenna for an incoming transmission.
power model	dra_power	Computes the received power for an incoming transmission.
bkgnoise model	dra_bkgnoise	Computes background noise affecting the incoming transmission.
Inoise model	dra_inoise	Computes interference noise affecting the incoming transmission.
snr model	dra_snr	Computes the signal-to-noise ratio for the incoming transmission.
ber model	dra_ber	Computes the bit error rate for the incoming transmission.
error model	dra_error	Computes the number of bit errors in a segment of the incoming transmission.

[5] OPNET models packet transmission across communications channel using a special mechanism called the Transceiver Pipeline. For more details on Pipeline stages, refer to OPNET Modeler online documentation, Modeling Concepts Manual, Chapter 6: Communication Mechanisms, topic Comec.4: Communication Link Models

Pipeline Stage Attribute Name	Default Value	Description
ecc model	dra_ecc	Determines the acceptability of an incoming transmission.

3.10.2 Required Modules

A radio device requires a radio transmitter and a radio receiver for transmitting and receiving data. An antenna module may be used if the modeling engineer wants to specify the antenna pattern. If an antenna module is not present, the pattern is considered to be "isotropic" by default.

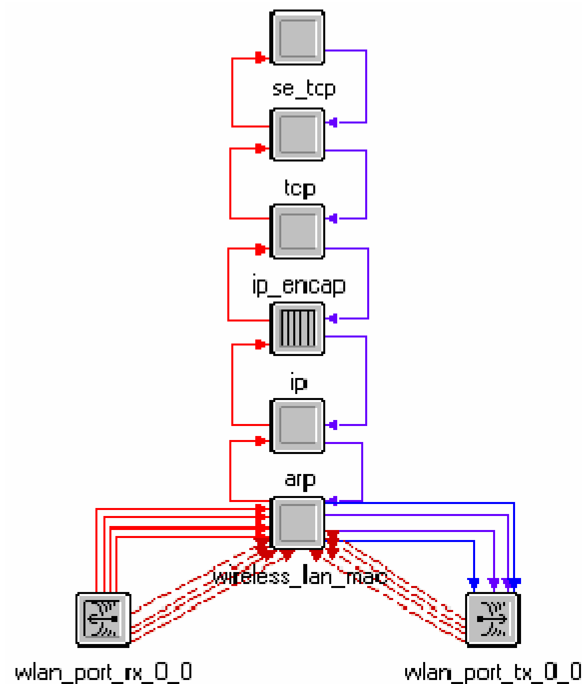


Figure 3-25: Radio End-System Device-Node Model

3.10.3 Initialization

There are no initialization steps specific to a radio device. If this is an end-system device with radio interfaces, look for the initialization steps under Subsection "3.5. Compliance for End-System Devices" that deals with building end-system devices.

3.10.4 Interfacing with Other Classes

A radio device can talk to another radio device or a satellite device if the two devices are within range and have matching frequencies, modulation, and data rates. Closure between the two devices is computed by the *closure* pipeline stage.

The OPNET Simulation kernel manages the transfer of packets from the source to the destination as a series of computations, each of which models particular aspects of the link behavior. These

computations are performed using pipeline stages. Each radio transmitter and receiver has a set of pipeline stage attributes that can be changed to modify the behavior of the link.

A model developer building a radio device can specify these pipeline stages on the transmitter and receiver to model the desired behavior. For more information about the transceiver pipeline stages, refer to the OPNET Modeler online documentation, Modeling Concepts → Communication Mechanisms → Communication Link Models section.

3.10.5 Interfacing with TIREM

Terrain Integrated Rough Earth Model (TIREM) is a set of libraries that facilitate modeling radio interference due to terrain. This feature is enabled through calls from the transceiver pipeline stages. (Files with the extension .ps.c implement pipeline stages.)

3.10.6 Restrictions in Building Radio Devices

There are some restrictions in building radio devices. Not all point-to-point interface types can be replaced by radio interfaces. Table 3-23 enumerates the restrictions and changes needed to build ports of different types with radio interfaces.

Table 3-23: Restrictions in Building Radio Devices

Interface Technology	Restrictions Involved in Building Ports with Radio Interfaces
SLIP	No restrictions. The point-to-point interfaces can be replaced by radio interfaces.
Ethernet	The point-to-point interfaces can be replaced by radio interfaces, and the behavior of the Ethernet MAC module has to be changed. Refer to OPNET's 802.11 (wireless LAN) models for more information.
ATM	An ATM port's point-to-point interfaces cannot be replaced by radio interfaces. A node with just radio and point-to-point interfaces is created, and the ATM node is connected by a point-to-point link to this node.
Frame relay	This combination is currently not supported.
FDDI	This combination is currently not supported.
Token ring	This combination is currently not supported.

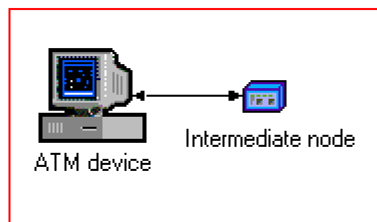


Figure 3-26: ATM Device Radio Interface

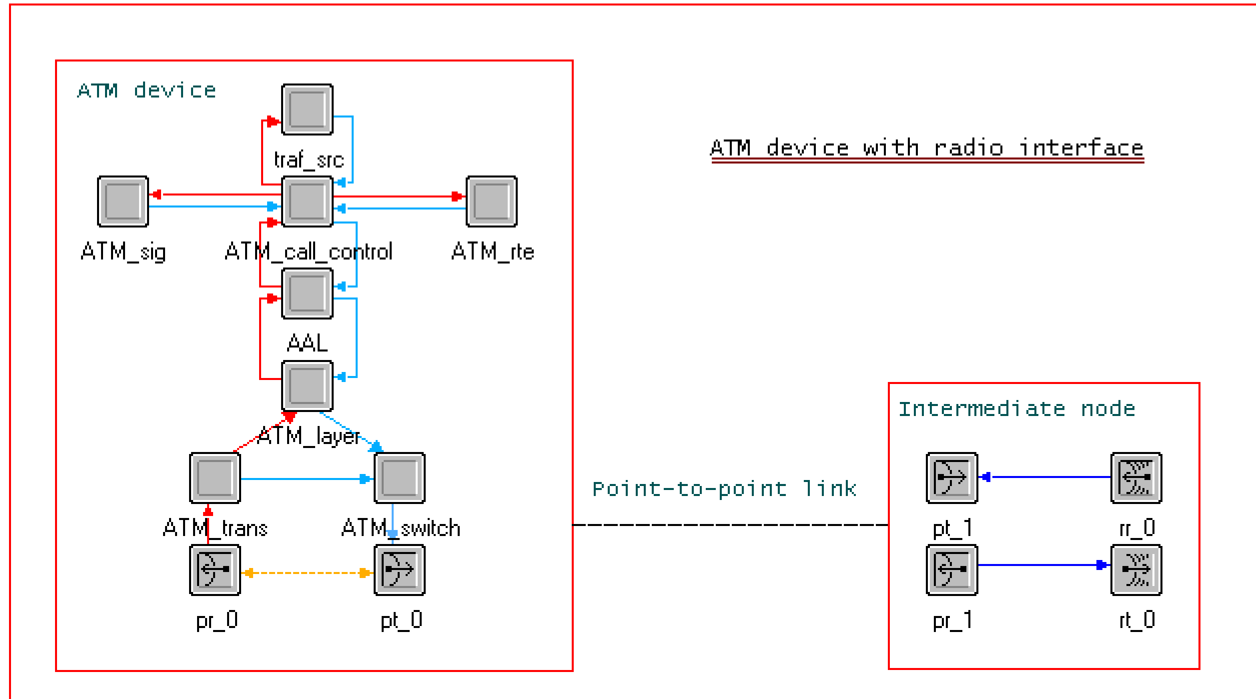


Figure 3-27: Internal Representation of ATM Device and Intermediate Node

3.10.7 Handling Failure/Recovery

There are no failure/recovery handling procedures specific to radio devices, although, if standard interface technology is not used, the appropriate module should flush out the queues inside. In NETWARS models, currently the devices connected to the radio devices perform the IER cleanup operations in case the radio device fails.

3.10.8 Collecting Statistics

Broadcast network utilization statistics are collected for broadcast radios.

3.10.9 Building Custom Pipeline Stages

When building a radio device, the model developer can use the OPNET Standard (COTS) pipeline stages on the radio transmitters and receivers. Model developers wishing to customize them to better suit their needs, may do so by creating custom pipeline stages. Custom pipeline stages can be built based on the OPNET Standard (COTS) pipeline stages. For more information about the stages, refer to the OPNET Modeler online documentation, General Models manual, “Pipeline Stages/Radio Link” chapter.

3.10.10 Satellite Considerations

A satellite device can be modeled as a networking device with radio interfaces, as documented above. The current NETWARS standard device model library includes geosynchronous

(geostationary) satellites, together with various ground terminals. A geosynchronous satellite is modeled using a radio device with an altitude set at 35,786 kilometers.

If the satellite device to be modeled is not geosynchronous but has another type of orbit, it must be built as an OPNET satellite node. Designating a device as a satellite node type creates an additional attribute that must be set, as shown in Table 3-24.

Table 3-24: Required Satellite Device Attributes for Moving Orbits

Satellite Device Attribute Name	Attribute Type	Default Value	Description
Orbit	Typed file	None	The orbit for the satellite device

When an orbit is specified, the node position information is ignored and the position at any point in time is determined from the orbit.

3.10.11 NETWARS Standard Geostationary Satellite Communications System Models

A satellite communications system can be modeled in two ways, either based on the NETWARS Standard Geostationary satellite model or built as a new stand-alone satellite communications system. If a new stand-alone satellite communications system is developed, no additional requirements beyond those listed above are required.

If satellite communications interoperability is required with the NETWARS Standard Geostationary satellite models, additional attributes are required. These additional attributes will provide a mechanism for the configuration of communications through the Scenario Builder GUI.

A ground terminal device model that can communicate with the NETWARS Standard Geostationary satellite models.

In addition to the attributes described below, the ground terminal model must have its *equipment_type* attribute set to “Satellite terminal” in order for Scenario Builder to discover it during link deployment and for the CP to recognize it during its runs.

The satellite and satellite terminal models employ the radio transceiver pipeline stages shown in Table 3-25.

Table 3-25: Radio Transceiver Pipeline Stages

Stage	Function	Module	File
0	Receiver Group	Tx	dra_rxgroup.ps.c
1	Transmission Delay	Tx	dra_txdel.ps.c
2	Link Closure	Tx	dra_closure.ps.c
3	Channel Match	Tx	dra_chanmatch.ps.c
4	Transmission Antenna Gain	Tx	dra_tagain.ps.c
5	Propagation Delay	Tx	dra_propdel.ps.c
6	Receiver Antenna Gain	Rx	dra_ragain.ps.c

Stage	Function	Module	File
7	Power Calculation	Rx	nwra_power_tirem.ps.c
8	Interference Noise	Rx	dra_bkgnoise.ps.c
9	Background Noise	Rx	dra_inoise.ps.c
10	Signal to Noise Ratio	Rx	dra_snr.pr.c
11	Bit Error Rate	Rx	dra_ber.ps.c
12	Error Allocation	Rx	dra_error.ps.c
13	Error Correction	Rx	dra_ecc.ps.c

For an example, refer to subsection 4.11, Satellite Terminal Generic Example.

3.10.12 Generic Satellite Device Model (for Bent Pipe Links)

To learn how to create a device of this type, refer to subsection 4.11, Satellite Terminal Generic Example.

3.10.13 Generic Satellite Ground Terminal Device Model (for Bent Pipe Links)

To learn how to create a device of this type, refer to subsection 4.11, Satellite Terminal Generic Example".

3.10.14 TSSP Satellite Terminal Device Model

To learn how to create a device of this type, refer to subsection 4.12, Satellite Terminal with TSSP Example.

3.10.15 Broadcast Radio Considerations

Integrating a custom radio with the broadcast network framework involves modifying some files that define this framework. NETWARS refers to broadcast radios as those that share a medium access via a protocol, such as a Time Division Multiple Access (TDMA)-based protocol.

The file *<NW DIR>\Scenario_Builder\<OPNET Rel>\netwars\rules\net_configs* defines the types of networks supported by the broadcast network. This file must have an entry for the custom radio technology to identify its—

- Radio type (just a unique string)
- Classification by default
- Data rate by default
- MOP probe status by default
- Supported capacities
- Supported data packet formats
- Supported voice packet formats.

The radio device model must also have properly named ports and port self-description. The port names should conform to the formats—

`"<technology name>_tx_<n>"` (for radio transmitters)

"<technology name>_rx_<n>" (for radio receivers)

This will require a port description with the name of—

"<technology name>_tx_<start..n> / <technology_name>_rx_<start..n>"

The port self-description will require its *interface type* attribute value set to—

"radio_rt:<technology name>"

Lastly, the radio transmitter and receiver channels will need to support the packet formats specified in the *net_configs*.

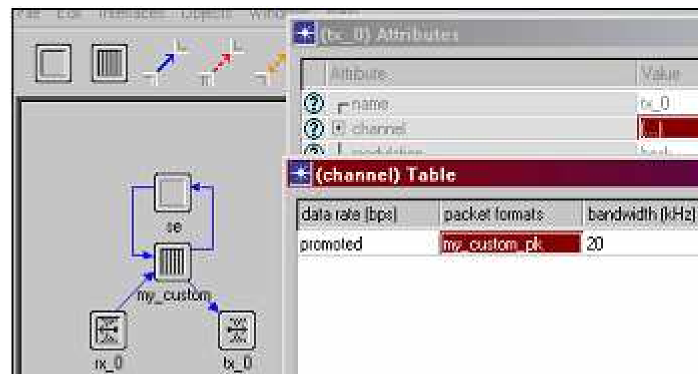


Figure 3-28: Channel Table

Each channel of the device to participate in broadcast networks will need to have the following attributes promoted to the node level:

- Data rate
- Minimum frequency
- Spreading code
- Power (transmitter only).

3.11 COMPLIANCE FOR LINK MODELS

Links connect devices in a network. NETWARS supports two different kinds of links: physical links that represent actual links that physically connect two devices, and links that only serve as logical entities that represent a physical connection, such as two radio interfaces configured to operate over the same frequency.

Both link types, physical and logical, display in the scenario. Although model developers can develop devices that work with the existing framework of the logical links, such as satellite terminals and Line of Site (LOS) radios, model developers outside of the NETWARS program, the target audience of this document, can only create new link models for physical links. Creating logical links requires access to a layer of NETWARS implementation not exposed as open source.

This subsection explains generically how to build a link model that represents a physical link connecting two devices. An important concept to note here is that OPNET models packet transmission across communication channels using a special mechanism called the transceiver pipeline. Typically, model developers refer to this in the context of radio transmission, but OPNET has a set of stages for point-to-point and bus transmission as well. This subsection explains the use of the point-to-point pipeline.

Note: "Point-to-point," in the context of the transceiver pipeline, simply means two endpoints of a link, not necessarily the technology serial or Point-to-Point Protocol (PPP).

For more details on the transceiver pipeline mechanism, refer to OPNET Modeler online documentation, Modeling Concepts Manual, "Communication Mechanisms" chapter, "Comec.4: Communication Link Models" subsection.

3.11.1 Attributes

This subsection describes the minimum set of attributes a link must have, as shown in Table 3-26.

Table 3-26: Required Attributes on Link Model

Attribute Name	Attribute Type	Default Value	Description
name	String	*-- Inherent --*	Specifies name of link
model	String	*-- Inherent --*	Specifies link model, for example, 100BaseT
data rate	Double	—	Specifies combined speed of data transmission over all channels in link
channel count	Integer		Specifies number of channels in link
packet formats	String	All	Specifies packet formats supported by link
closure model	Typed file	dpt_closure	Determines connectivity between transmitter and receiver
coll model	Typed file	dpt_coll	Used to determine if a collision has occurred on a link

Attribute Name	Attribute Type	Default Value	Description
ecc model	Typed file	dpt_ecc	Determines whether a packet can be accepted
error model	Typed file	dpt_error	Determines number of errors in a packet
propdel model	Typed file	dpt_propdel	Calculates propagation delay between a transmitter and a receiver
txdel model	Typed file	dpt_txdel	Calculates transmission delay associated with transmission of a packet. Default value is dpt_txdel.

The attributes closure model, coll model, ecc model, error model, propdel model, and txdel model correspond to various pipeline stages.

3.11.1.1 Dependencies

- The link model must support all packet formats supported by the transmitters and receivers in the devices to which it will connect.
- The link model must match the data rate supported by the transmitters and receivers in the devices to which it will connect.
- Failure to satisfy the above constraints will result in inconsistent links, and traffic cannot flow over inconsistent links.
- When creating a new link type, the LinkTypeMap.gdf file needs an entry for that new link type for it to function with NETWARS' Link Deployment Wizard (LDW). The LDW uses this file to match links to appropriate ports. NETWARS maintains this file under <NETWARS DIR>\User_Data\Rules.

Usually the data rates on the transceivers are left as "unspecified," which means the data rate taken by the transceivers during the simulation will be the data rate of the link.

3.11.2 Building Custom Pipeline Stages

When building a link model, model developers can use the OPNET Standard (COTS) pipeline stages. If model developers wish to customize them to better suit their needs, they may do so by creating custom pipeline stages. Custom pipeline stages can be built based on the OPNET Standard (COTS) pipeline stages. For more information about the OPNET Standard (COTS) pipeline stages, refer to the OPNET Modeler online documentation, General Models manual. Note that if a pipeline stage drops a packet where a non-ACK-based protocol, such as UDP, serves as the transport protocol, the pipeline stage must write out the failure statistic for the IER.

A link model can have model attributes, and the model developer can write code in the pipeline stages to deal with these. An example of a model attribute is *background utilization*, which allows the user to specify utilization on the link as a percentage of the total link bandwidth. This is a way of loading the link with traffic in addition to the IER traffic, and it allows the user to study the link performance under varying loads. The pipeline stage `dpt_propdel_bgutil` uses the *background utilization* attribute. The *background utilization* attribute can be imported in

NETWARS using the COTS Traffic import of the Cisco eHealth Traffic. For further details, refer to the *NETWARS User Manual*.

3.11.3 Handling Background Routed Traffic

The model developer has to specify pipeline stages on the link models that can handle tracer packets generated from the end-system IP module. OPNET Standard models have pipeline stages that can handle the load represented in a tracer packet and accordingly subject explicit packets to appropriate transmission and propagation delays. These pipeline stages record the statistics on the links with the appropriate background load specified on them. Refer to the `dpt_propdel_bgutil` and `dpt_txdel_bgutil` pipeline stage models with the OPNET Standard models as a baseline for creating custom pipeline stages that support background routed traffic.

3.11.4 Handling Failure/Recovery

A link model does not do anything itself to handle its failure/recovery. The devices to which the links are connected handle a link's failure/recovery.

3.11.5 Building Simplex Links, Buses, and Bus Taps

The process of building simplex links, buses, and bus taps is the same as building duplex links. In the Link Model editor, there is a field called "Link Types." Depending on what type of link is needed, one of the available link types is chosen. The possible types of links that can be created are—

- ptsimp (point-to-point simplex)
- ptdup (point-to-point duplex)
- bus
- bus tap.

The radio links (including the satellite links and broadcast networks) created in the Scenario Builder do not have an associated link model. They are notional links where the communication is established using correct settings for the radio device model attributes.

3.11.6 Collecting Statistics

A link model cannot be programmed to collect statistics. In OPNET, strictly speaking, there is no process model (code) within a link model (lk.m). The simulation kernel collects statistics on the link model.

Although a user can define statistic handles in a process model and write to them in a link model (pipeline stage), the pipeline stage needs to get a reference to the handle, and this can be done via the `oms_pr_*` kernel procedures. Other ways exist, but most model developers use this mechanism.

3.11.7 Documentation

To document a link model, the following information must be provided in the *Comments* section of the *Interfaces* option in the Link Model Editor:

- **General Description of the Link Model.** Provides a brief description of the link model.
- **Link Interfaces.** Documents the types of devices to which this link connects.
- **Data Rate.** Specifies the data rate for this link.
- **Packet Formats.** Specifies the packet formats supported by this link.
- **Comments.** Gives any additional comments or restrictions on using this link.

The self-description information must be set on the link models. This information, although currently not used by the Scenario Builder, may be used to get interface type information (equivalent to packet formats).

3.12 COMPLIANCE FOR UTILITY NODES

Utility nodes provide a simplified and unified location for information about a network. They do not represent actual devices in the network; rather, they represent information about a network.

3.12.1 Attributes

Table 3-27 and Table 3-28 give the minimum required and optional attributes for utility nodes.

Table 3-27: Required Attributes for Utility Nodes

Attribute Name	Attribute Type	Description
name	String	Specifies name of utility node
model	String	Specifies device model

Table 3-28: Optional Attributes for Utility Nodes

Attribute Name	Attribute Type	Description
utility_technologies	String	A list of packet formats supported by models using utility node
End node (N)[6]	String	Specifies full hierarchical name of an end node, where 'N' is an integer value (1, 2, etc.) These attributes are only mandatory if needed by the utility node. Attributes named as such can be placed within compound attributes to build a table.

3.12.2 Required Modules

The required modules depend on the purpose of the model. They should be designed to work with multiple instances of the same models so a simulation will not be confused by the presence of several of the same utility modules.

3.12.3 Interfacing with Other Classes

A utility node interfaces with other classes using any OPNET-supported techniques, including the OPNET process registry, which allows for the publishing of information that is available to other models, global variables, and structures or directly setting attributes of other objects. The models using the utility nodes should be designed to work with multiple instances of the utility node.

3.12.4 Interfacing with the Scenario Builder GUI

The *utility_technologies* attribute is used for objects that will be setting *end node (N)* attributes. The *utility_technologies* attribute must contain a listing of all supported packet formats that are used by the end nodes. The Scenario Builder GUI will then use this information to create a pop-

[6] For example, if a Promina utility node contains information about one Promina circuit connected between edge devices *ed1* and *ed2*, then it will have attributes *end node (1)* and *end node (2)*. The values of these attributes will be the full hierarchical names of *ed1* and *ed2*.

up list of devices within the scenario that also support this packet format. This mechanism is for the convenience of the user. The user can then select from this list to fill in all *end node (N)* attributes. A good example of this would be a circuit configuration utility with an attribute called *circuit_config* and subattributes called *end node (1), end node (2),* and *bandwidth (bps)*. With a *utility_technologies* attribute set to “cs_special” and the *circuit_config* attribute promoted, the NETWARS Scenario Builder user of this model would see pull-down menu options under the *circuit_config* attribute for the *end node (N)* attributes of every device in the NETWARS Scenario that supports packet format “cs_special.” In this way, the user would have an easy way to set up “cs_special” circuits.

4 EXAMPLES

This section discusses the approach a model developer should take to build a device model, a networking protocol, and so forth. Each step in the approach is illustrated using an example device or protocol. These examples serve as a code reference for the developer to develop other models and include a detailed discussion at the code level to help developers understand the underlying concepts and methodology to develop similar, new models.

Please note that this is not a discussion on the use of the OPNET Modeler's various editors[7] and model hierarchies. The OPNET Modeler development environment is used to develop the models.

The discussion is based on certain assumptions about the device model or the protocol in hand. These assumptions are discussed in the "High-Level Design" subsection of the corresponding code example.

Supplemental files for each of these examples, including the relevant node models, process models, external C code, and header files are provided separately for reference.

[7] Please refer to the OPNET Modeler's online documentation on the Node Editor and Process Editor in the Editor Reference section.

4.1 TRAFFIC MODEL EXAMPLE

The basic ideas behind creating traffic models were discussed in section 3.1. The purpose of this section is to introduce the major steps that were used to support the Net-Centric Enterprise Service (NCES) application models development with ACE whiteboard.

NCES applications are based on Service-Oriented Architecture (SOA). In order to model the dynamic interaction characteristic of NCES applications, the following approach was applied. First, the developers defined the scope of the model and gathered the corresponding architectural information and testing data from the application developers and associated programs. Second, the developers analyzed the collected data and identified all possible dynamic interactions/cases of the applications. Third, the developers created a time sequence diagram, as shown in Figure 4-1, to document the dynamic interactions.

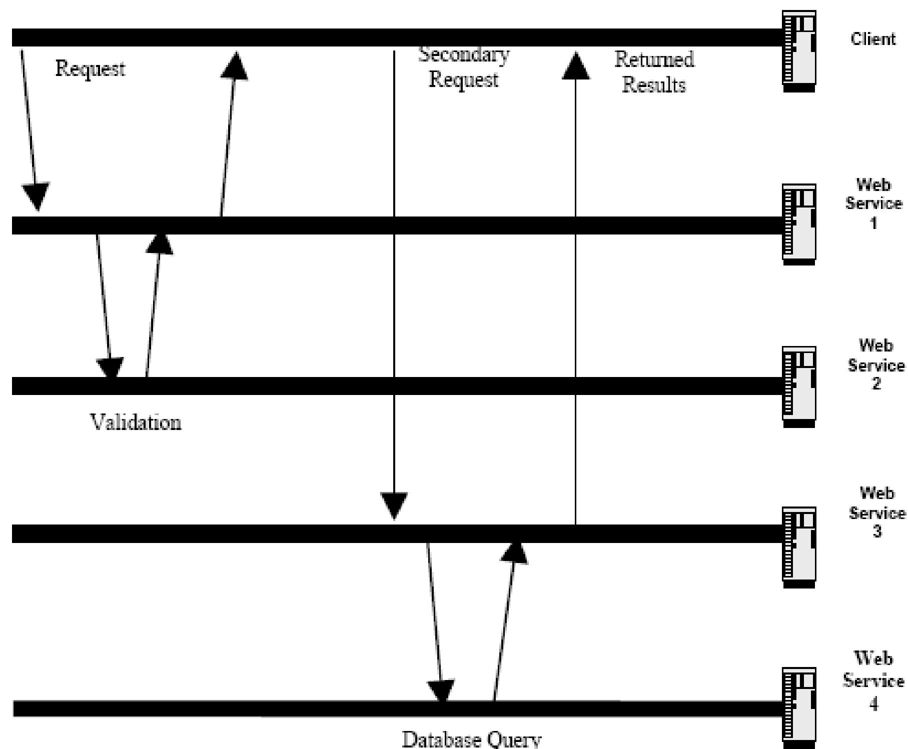


Figure 4-1: Time Sequence Diagram

The next step was to apply the time sequence diagram to design the traffic model architecture. The architecture included the following information: number of tiers, tier names, reusable interactions, message sizes, message interarrival periods, and interaction logics. In the final step, the developer used the architecture to create the application models in ACE whiteboard and apply Python scripts to implement interactions logics. Please refer to "ACE Whiteboard Tutorial: Modeling an Application using Logic Scripts (Advanced)" in OPNET documentation for more examples.

Please contact Defense Information Systems Agency (DISA) GE34 for detailed NCES Modeling and Simulation (M&S) information.

Other than SOA applications, ACE Whiteboard can also be used to model the dynamic interactions of operational scenarios that include logical decisions, such as the following Communities of Interest (COI) publish and subscribe operational scenario:

1. An intelligence cue of type X arrives at a command center. Data is posted on the X-Community of Interest (COI) web site.
2. An alert is sent to all members of the X-COI who subscribe to that kind of cue.
3. Some members of the X-COI are available, others are not. (Some are off-shift; some are already involved in other incidents, perhaps of the same type or perhaps of different types.) The ones who are available say so (e.g., with messages in the X Chat Group).
4. The available X-COI members download material from the web site.
5. The X-COI has a teleconference.

4.2 ROUTING PROTOCOL EXAMPLE

The following subsection discusses the issues that a developer confronts when interfacing a custom routing protocol with standard protocol stack.[8] A code-level discussion is presented on the various steps a developer needs to take to create a working device model that includes custom routing.

4.2.1 High-Level Design

Although a developer can select from a range of algorithms when developing the routing protocol itself, the following discussion deals with how to make this algorithm interoperate with other standard technologies such as the IP and transport layers, which are already modeled in the standard model library that comes with the OPNET Modeler.

The following are the design decisions[9] made for the protocol under discussion:

- **Protocol Type.** The routing protocol is a distance-vector protocol.
- **Routing Metric.** The routing metric is hop counts.
- **Routing Updates.** The routing updates are sent at regular intervals and when the network topology changes. When a router receives a routing update that includes changes to an entry, it updates its routing table to reflect the new route.
- **Timers.** Route timers are implemented for this routing protocol, including the Route Timeout Timer and the Garbage Collection Timer.
- **Layer 3 Technology.** The Layer 3 technology used here is IP.

The Routing Element[10] “RE” represents this custom routing layer for the device under discussion. This is interfaced with the IP layer. Typical Layer 3 networking equipment is shown in Figure 4-2.

---

[8] For more information on the OSI layer (protocol stack), please refer to Section 2, Prerequisites for Designing and Building NETWARS Models of Model Development Guide v.1.4 for the suggested networking references.

[9] These design decisions give the reader ideas on what the basic tenets are on which the custom routing protocol under discussion is based on and may not be included in the following discussion.

[10] Routing Element “RE” is just an arbitrary name chosen for discussion here and should not be misinterpreted as being a routing protocol for NETWARS. Also, the user should not draw any analogy between the NETWARS’ SE or OE.

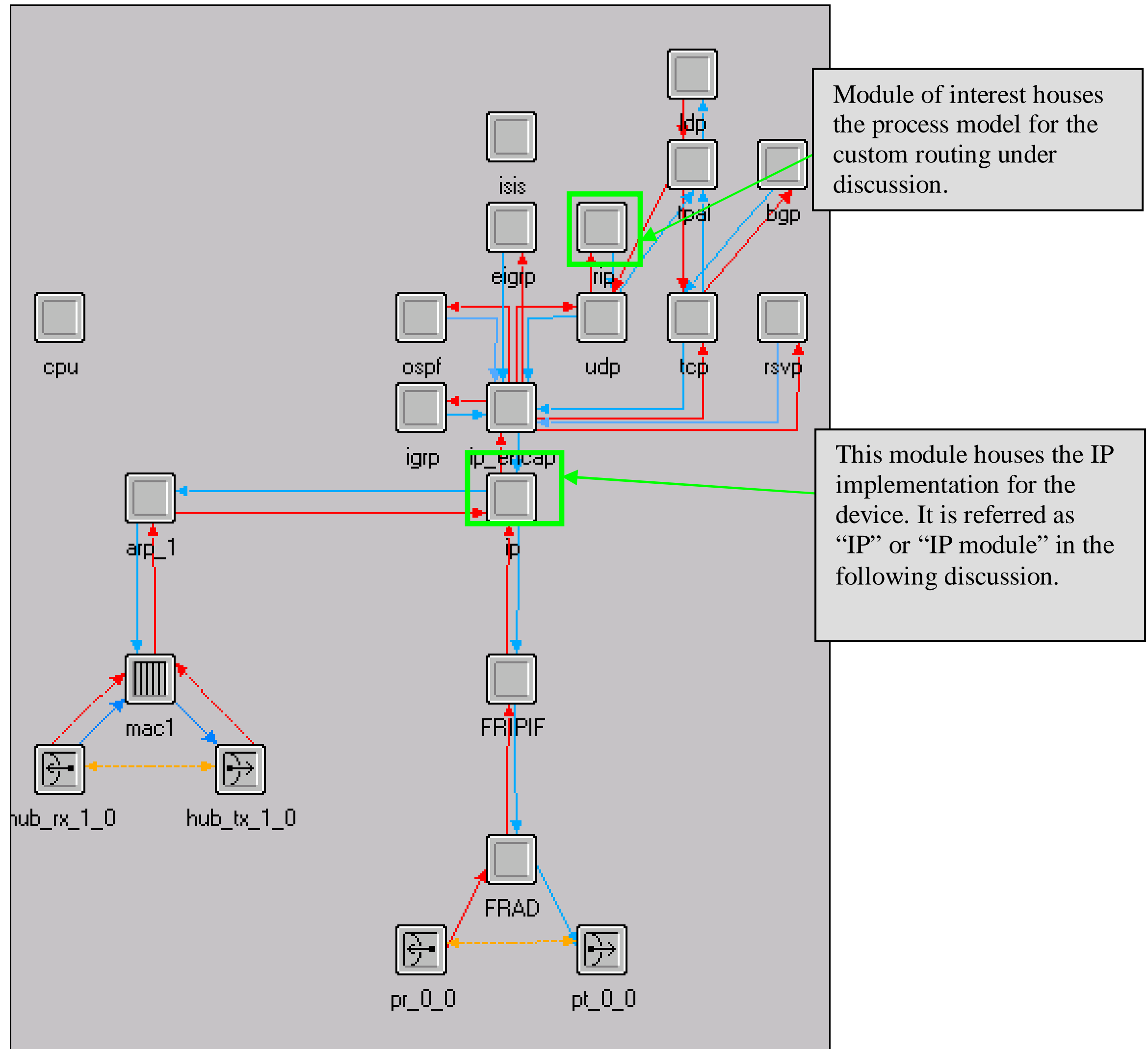


Figure 4-2: Layer 3 Networking Equipment

In this figure, isis, rip, ospf, igrp, eigrp, and bgp represent the actual routing protocols. The RE module can be added at the location of the "rip" module because this routing protocol closely resembles our RE based on the high-level design decisions taken earlier. The intention is to look closely at the process model inside this module that performs the various interfacing functions in which we are interested.

- **Register the Routing Protocol.** This is required because the custom routing protocol requires a distinctive ID that it will later use when modifying the route entries in the IP Common Route Table.[11]
- **Make the Routing Protocol Available.** The routing protocol should be available to be configured on the interfaces of the router.

[11] IP Common Routing Table refers to the routing table information that the routing device (e.g. router) has. This common routing table is populated by one or more routing protocols.

- **Initialization.** The routing protocol must access the IP module of this router and retrieve the information stored by the IP in the process registry.[12] This gives the routing protocol information regarding the gateway status of the device, interface information, and so forth. Here, the routing protocol can initialize the routing tables for the first time.
- **Routing updates.** The IP common route must be updated with the entries that the routing protocol may want to add or delete.

The following subsections provide detailed discussion on the topics listed above. At the end of following discussion, the reader should have developed a fair understanding on how interfacing with the IP is done for a routing (custom) protocol.

4.2.2 Interfacing with the IP Discussion

4.2.2.1 Registering the Protocol

The protocol needs to register itself in the OPNET Model Support (OMS) process registry and also with IP. Both these steps need to be performed upon receiving the "begin sim" (*begsim*[13]) interrupt.

The function that is used to register the routing protocol with IP is—

int Ip_Cmn_Rte_Table_Custom_Rte_Protocol_Register (char* custom_rte_protocol_label_ptr).

This function returns a unique integer that is used as the routing protocol ID. This unique routing protocol ID is used for all calls to Ip_Cmn_Rte_Table API[14] functions.

```
/* Register the Routing Protocol with IP and get the unque Routing Protocol ID  */
/* In the actual implementation the developer is suggested to use the name of  */
/* Routing Protocol itself as the argument to the following function call.     */
custom_routing_protocol_id = Ip_Cmn_Rte_Table_Custom_Rte_Protocol_Register ("Custom Routing Protocol");
```

4.2.2.2 Initialization of the Routing Protocol

After registering the protocol with the IP as discussed in Subsection 4.2.2.1, the IP sends remote interrupts to all the routing protocols registered with it.

The remote interrupt received from the IP is as follows:

```
#define IP_NOTIFICATION     ((intrpt_type == OPC_INTRPT_REMOTE) && \
     (routing_table_import_export_flag != IP_RTE_TABLE_IMPORT) && (intrpt_code == 0))
```

While registering in the OMS process registry, the attribute named protocol of the process handle must be set to same string used for registering with IP.[15] The following section of the code is an

[12] The information stored in the process registry can be retrieved by the other process models. Please refer to the OPNET Modeler documentation on the Process Registry under General Models | OPNET Model Support package for details on how to use process registry.

[13] Please refer to OPNET Modeler documentation on Event Schedule Simulation under Modeling Concepts | Modeling Framework for more information on the *begsim* interrupt.

[14] Details on the API are provided in Section 4.2.2.4.

[15] Code where IP does the OMS process registry not shown.

example of how to register the routing protocol in the OMS process registry. The start time attribute in the following code refers to the start time for the routing protocol; this could be an attribute on the custom routing protocol process model.

```
/* Obtain the object id of the "RE" module.                    */
own_id = op_id_self ();

/* Obtain the surrounding node's objid.                        */
own_node_objid = op_topo_parent (own_id);

/* Obtain the RE process's prohandle.                          */
own_prohandle = op_pro_self ();

/* Initially the route table is empty.                         */
route_table = OPC_NIL;

/* Obtain the name of the process -- the "process model"      */
/* attribute of the surrounding module.                        */
op_ima_obj_attr_get (own_id, "process model", proc_model_name);

/* Register the Custom Routing processin the model-wide process registry.*/
own_process_record_handle = (OmsT_Pr_Handle) oms_pr_process_register (own_node_objid,
    own_id, own_prohandle, proc_model_name);

/* Register any other attributes that may be of interest to other processes     */
/* The label passed in the actual routing prtocol implementation may be the name of the protocol itself */
oms_pr_attr_set (own_process_record_handle,
    "protocol",              OMSC_PR_STRING,     "Custom Routing Protocol",
    "Custom Routing Start Time",        OMSC_PR_NUMBER,     start_time,
    OPC_NIL);
```

To perform some other functions, including the process of finding which interfaces have this routing protocol enabled, the module needs to get the process registry information of the IP. The string “ip” needs to be used to discover IP-registered process registries.

```
/* Obtain the process record handle of the ip process residing in the local node.   */
proc_record_handle_list_ptr =  op_prg_list_create();
oms_pr_process_discover (OPC_OBJID_INVALID, proc_record_handle_list_ptr,
    "node objid",        OMSC_PR_OBJID,       own_node_objid,
    "protocol",          OMSC_PR_STRING,      "ip",
    OPC_NIL);
```

The information retrieved above includes gateway/router status of the node, interface information, IP route table, and so forth. From the IP process registry, the custom routing protocol can then identify the interfaces on which it is enabled. This is a two-step process:

- Get a pointer to the data structure storing the IP information and retrieve information such as interface information, IP common route table,[16] etc.

```
process_record_handle = (OmsT_Pr_Handle) op_prg_list_access (proc_record_handle_list_ptr, OPC_LISTPOS_HEAD);

/* Obtain a pointer to the shared module memory of IP */
oms_pr_attr_get (process_record_handle, "module data", OMSC_PR_ADDRESS, &ip_mod_mem_ptr);

/* Obtain the interface information from the IP process_record_handle. */
oms_pr_attr_get (process_record_handle, "interface information", OMSC_PR_ADDRESS, &ip_info_ptr);

/* Obtain a reference to the IpT_Cmn_Rte_Table object      */
/* for this node. This object is created and registered by IP.   */
/* This is a reference to the "common route table" that     */
/* will be updated/modified by all routing protocols running*/
/* on this node.                                            */
oms_pr_attr_get (process_record_handle, "ip route table", OMSC_PR_ADDRESS, &ip_route_table);
```

[16] This ip_route_table pointer is needed every time the routing protocol needs to modify the IP common route tables with its entries.

- Loop through the list of interfaces maintained by IP. If the routing protocol was enabled on a particular interface, then its protocol ID is present in the "routing_protocols_lptr" list of that interface. For each entry access, enter a new route[17] (of "0" cost) into the IP common routing table.

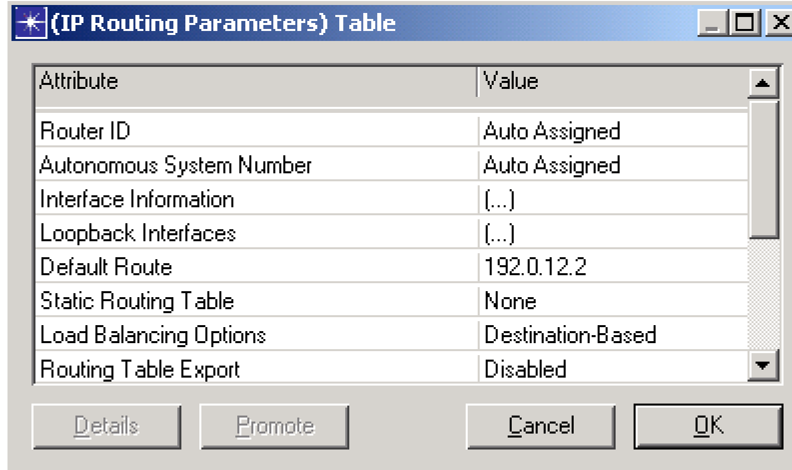
```
/* Get the pointer to the ip interface table from the Interface */
/* information retrieved from the Process Registry.             */
iface_table_ptr = ip_info_ptr->ip_iface_table_ptr;

/* Obtain the size of the ip interface table.         */
ip_iface_table_size = op_prg_list_size (iface_table_ptr);

/* Loop over each element in the IP interface list published by */
/* by IP and if this interface has been assigned "Custom Routing Protocol" as its       */
/* routing protocol, create a corresponding entry in the        */
/* routing table.                                               */
for (i = 0; i < ip_iface_table_size; i++)
    {
    /* Obtain a handle on the i_th interface.                   */
    ip_iface_elem_ptr = (IpT_Interface_Info*) op_prg_list_access (iface_table_ptr, i);

    if (ip_interface_routing_protocols_contains (ip_iface_elem_ptr->routing_protocols_lptr,
        IpC_Rte_Custom) == OPC_TRUE)
        {
        /* Obtain the internal ip address corresponding to the full */
        /* IP network address.                                      */
        ip_internal_address = ip_rtab_network_convert (ip_iface_elem_ptr->network_address);

        /* Obtain the Custom Routing Protocol's interface pointer in order to read user configuration */
        crp_intf_ptr = (RipT_Interface_Table_Elem *) op_prg_list_access (crp_intf_table_lptr,i);

        /* Add an entry in the routing table with a cost of 0.      */
        custom_rte_new_entry_add (&route_table, ip_iface_elem_ptr, ip_internal_address,
            ip_iface_elem_ptr->network_address, ip_iface_elem_ptr->addr_range_ptr->subnet_mask,
            ip_iface_elem_ptr->addr_range_ptr->address, 0, ROUTING_PROTOCOL_VERSION_CONSTANT,
            crp_intf_ptr->triggered_mode, version);
        }
    }
```

Note that the "IpC_Rte_Custom"constant is used to check whether the interface is using Custom Routing Protocol. This enumerated value comes from IpT_Rte_Protocol enumeration defined in the ip*_rte_v4.h* header file of the OPNET standard model library.

4.2.2.3 Support for Routing Protocol Configuration

All the router devices in OPNET/NETWARS have parameters available for configuration (as part of the *IP Routing Parameters* device attribute). To change any of this attribute's properties, as is done in this section, open the *ip_dispatch.pr.m* file in OPNET Modeler and open its model attributes (Interfaces -> Model Attributes). This particular attribute includes information such as router ID, loop-back information, interface information, and so on (as shown in Figure 4-3).

[17] Please refer to the function rip_rte_new_entry_add() of rip_v3 process model of the OPNET standard model library for details on how to add a new route entry to the IP Common Route Table. Also refer to Section 4.2.2.4 for details on the APIs for the IP common route table.



(IP Routing Parameters) Table

Attribute	Value
Router ID	Auto Assigned
Autonomous System Number	Auto Assigned
Interface Information	(...)
Loopback Interfaces	(...)
Default Route	192.0.12.2
Static Routing Table	None
Load Balancing Options	Destination-Based
Routing Table Export	Disabled

Figure 4-3: IP Routing Parameters Attribute

Certain parameters can be at higher levels of granularity, on an interface basis. This information includes parameters such as the IP address information, the routing protocol, and the QoS profile. This is where the user can configure which routing protocol to use for that interface (as shown in Figure 4-4).

(Interface Information) Table

Name	Status	Address	Subnet Mask	MTU (bytes)	Metric Information	Routing Protocol(s)
IF0	Active	192.0.12.1	255.255.255.0	ATM	Default	OSPF
IF1	Active	Auto Assigned	Auto Assigned	ATM	Default	OSPF
IF2	Active	Auto Assigned	Auto Assigned	ATM	Default	OSPF
IF3	Active	Auto Assigned	Auto Assigned	ATM	Default	OSPF
IF4	Active	Auto Assigned	Auto Assigned	ATM	Default	OSPF

Figure 4-4: Interface Information Attribute

In order to use the custom routing protocol, the IP module's process model (*ip_dispatch*) must be updated. The model attribute "Routing Protocol" must be edited[18] to include the custom routing protocol (IP Routing Parameters | Interface Information | Routing Protocol(s)). A new symbol map must be added for this attribute[19] (as shown in Figure 4-5).

18 Adding a new attribute to the process model does not require the developer to compile the process model.
19 To make this change available during the simulation, the process model must be saved. No recompilation is necessary.

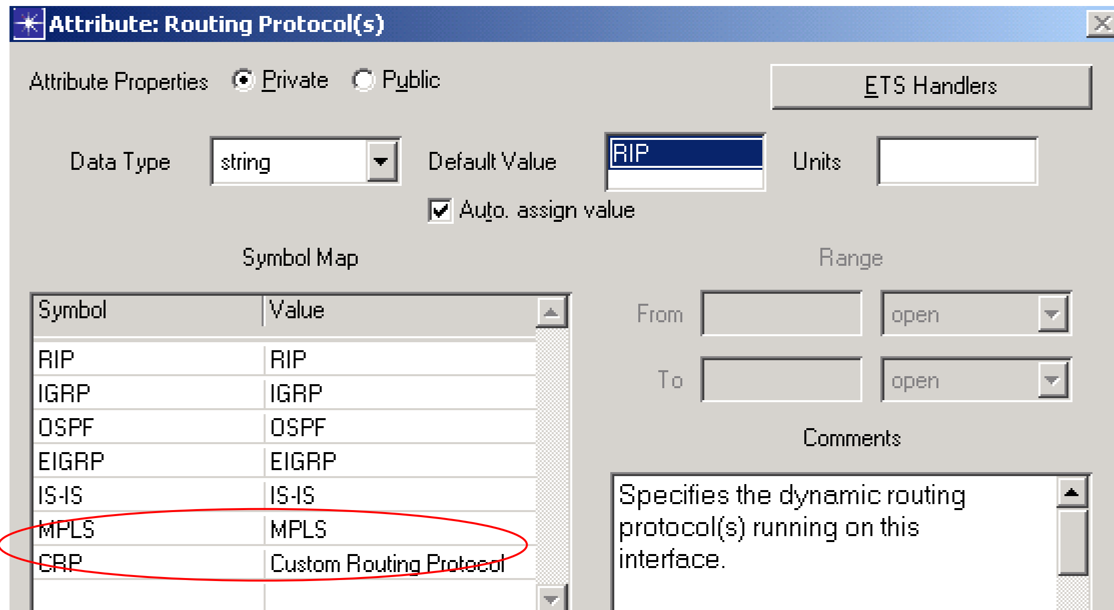


Figure 4-5: Routing Protocol Attribute Properties

The “loop-back interfaces” attribute must also be updated in a similar way to include the custom routing protocol.

4.2.2.4 IP Common Route Table API Functions

These API functions can be used by the custom routing protocol to interact with the IP common routing table and modify the entries when the protocol finds a change in the route entry. The functions shown in Table 4-1 can be used to insert and remove routes into/from the common route table.

Table 4-1: Available IP Common Route Table API Functions

Route Management API	Description
Ip_Cmn_Rte_Table_Custom_Protocol_Register (char* custom_rte_protocol_label_ptr)	Registers the custom routing protocol with the common route table. A unique protocol_id is returned for accessing the route table.
Ip_Cmn_Rte_Table_Entry_Add (IpT_Cmn_Rte_Table* route_table, void* src_obj_ptr, IpT_Address dest, IpT_Address mask, IpT_Address next_hop, IpT_Port_Info[20] port_info, int metric,	Adds a route entry to the common route table. This function checks for an already existing entry.

[20] The port_info structure tells IP which outgoing interface needs to be used to reach the specified next_hop. This structure contains two fields: intf_index and intf_name. The intf_index is the index of the interface in the interface table maintained by IP, and the intf_name is the name of the corresponding interface. This structure can be populated using the ip_rte_addr_local_network function. Please refer to the “ip_cmn_rte_table.h“ and “ip_rte_support.h“ for the definition of the structure and the declaration of the function, respectively.

Route Management API	Description
int proto, int admin_distance)	
Ip_Cmn_Rte_Table_Route_Delete (IpT_Cmn_Rte_Table* route_table, IpT_Address dest, IpT_Address mask, int proto)	This function is used to delete an entire destination entry from the IP Route Table. This deletes all the route table entries that this destination may have.
Ip_Cmn_Rte_Table_Entry_Delete (IpT_Cmn_Rte_Table* route_table, IpT_Address dest, IpT_Address mask, IpT_Address next_hop, int proto)	This function is used to delete a next hop from the entry from the IP Route Table.
Ip_Cmn_Rte_Table_Entry_Exists (IpT_Cmn_Rte_Table* route_table, IpT_Address dest, IpT_Address mask, int admin_distance)	This function determines whether a route exists in the common route table.
Ip_Cmn_Rte_Table_Entry_Update (IpT_Cmn_Rte_Table* route_table, IpT_Address dest, IpT_Address mask, IpT_Address next_hop, int proto, int new_metric)	This function is used to change the metric associated with a current route table entry. The entry for the given destination is searched for the next hop given, assuming a matching protocol ID, and then the metric associated with the given next hop is changed.

4.2.2.5 Function Arguments:

The arguments for these functions are discussed below:

- **route_table.** Pointer to the IP common route table
- **src_obj_ptr.** Pointer to the entry in the source routing protocol; can be set as OPC_NIL for custom protocols
- **dest.** IP Address of the destination network
- **mask.** Subnet mask of the destination network
- **next_hop.** IP address of the interface that should be used as the next hop for the destination addressed entered
- **port_info.** Contains the “addr_index” of the interface used to reach the next hop
- **metric.** Metric value assigned to this next hop; this is the cost associated with the next hop[21]

[21] The custom routing protocol may implement its own metric, the way of determining cost (e.g., hop count, link bandwidth).

- **proto.** The unique protocol that entered this route[22]; the protocol ID, obtained from Ip_Cmn_Rte_Table_Custom_Rte_Protocol_Register, in case of a custom routing protocol
- **admin_distance.** The preference associated with this entry.

4.2.3 Notes

Following are some other useful notes that may help the developer of the custom routing protocol.

4.2.3.1 Simulation Attributes[23]

- **IP Routing Table Export/Import.**[24] This attribute can be used to export the routes developed by the routing protocol; a text file (*.gdf) is generated in the primary mod_dirs.[25] To use the already existing routes, this attribute should be set to "2" (as opposed to "1," for the export).
- **IP Dynamic Routing Protocol.** This simulation attribute can be set if the custom routing protocol needs to be run over the complete network. This preference set here takes precedence over the local specification.

[22] Because this API is entering the route to the IP common route table where more than one routing protocol may enter a route to the desired destination, this protocol ID distinguishes the routes added by different routing protocols.
[23] Please refer to the OPNET Modeler online documentation (Modeling Concepts → Process Domain) for details on the simulation attributes.
[24] The simulation attribute can be added in the "start_scm" batch file (located at Sim_Domain\bin) where the simrun executable is called (e.g., IP Routing Table Export/Import 1).
[25] This is the mod_dirs attribute for the env_db file of the simulation domain. For details on the mod_dirs preference and setting environment attributes, please refer to OPNET online documentation (Modeling Concepts → External Interfaces → System Environment).

4.3 WIRED END DEVICE EXAMPLE

4.3.1 Problem Statement

The objective is to build[26] an end node model that generates and receives data IERs. The following subsection discusses in length with the help of a code example how the process model implementation works for such a node model.

4.3.2 High-Level Design

4.3.2.1 Node Model Discussion

In this particular example, the following high-level decisions (assumptions) are made for the end device.

- **Transport Protocol.** TCP is the supported protocol for the transport layer. Other options are UDP or a custom transport protocol.
- **Layer 3 Protocol.** IP is used as the Layer 3 protocol.
- **Routing.** Routing is not performed by the end device, therefore, no routing protocol decisions have to be made.
- **Lower Layers.** Ethernet is the supported data-link layer technology.

An application layer must be designed to interface with the transport layer. The System Element "SE" represents the application layer in the NETWARS end-device models.

With this information, the high-level node model representation would be similar to the one represented in Figure 4-6.

[26] Please refer to the Model Development Guide v3.0, Subsection 3, Compliance for End System Devices, on the approach and methodology for creating an end-device model.

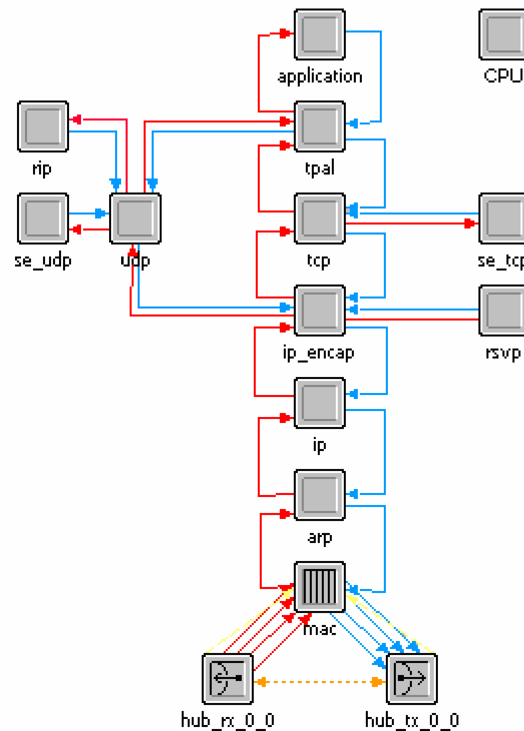


Figure 4-6: End-Device Node Model

The node model in Figure 4-6 is the actual node model of the NETWARS standard node models; "NW_ethernet_wkstn_adv"[27] (NETWARS 2006-2.1).

For details on how to design the node model for an end-device model, please refer to the "3.5. Compliance for End-System Devices" subsection. The following discussion about the process model development assumes that the minimum required attributes[28] for the end-device are set.

4.3.3 Detailed Design: Event Response Table

The process model is designed to satisfy the functionality of the device node discussed above. A functional process model diagram is presented at the end of this subsection.

4.3.3.1 Module Context and Functionality

Context:

In almost all cases, process models describe the behavior of a single module within a node model, consisting of many modules.[29] The role of the process model can then generally be described by the interactions that it has with the other modules in the node model. From the point

[27] Please refer to Figure 3-7 (Ethernet_wkstn_adv—Node Model) of the Model Development Guide v3.0.

[28] Please refer to Section 3, Compliance with End-System Devices, for the set of minimum attributes required for an end-device model.

[29] Please refer to the *NETWARS Model Development Guide, Section 3,* "NETWARS Component Classes," for discussion on the top level component classes and "interfaces."

of view of other modules in the node model, only the external "black-box" behavior of their process model(s) is of concern, not their internal implementation. It is therefore an important first step in the development of a process model to identify the other system components (modules) with which it must interact.

In case of an end-device node model, the "se" module must interact with the following other modules (refer to Figure 4-7):

- oe (of the OE node model)
- tcp (Transport Layer Protocol of the end device node model).

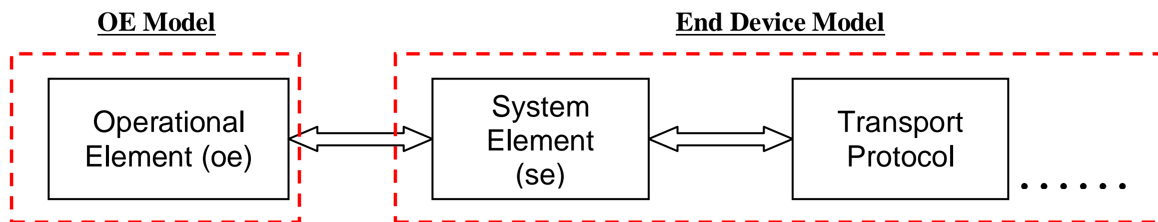


Figure 4-7: Interfacing Modules of "se"

Functionality:

Because the development of the process model for the *se* module is discussed in the following subsections, the functions of a "System Element" are enumerated below so that it can be related to the event response table developed for the process model. The main function of the *se* module is to interact with the *OE* and the *tcp* module and perform the following functions:

- End-device selection (OE)
- Traffic generation
- Handling of TCP connections
- Traffic reception
- Handling of failure/recovery.

This high-level functionality is represented in

Figure 4-8:[30]

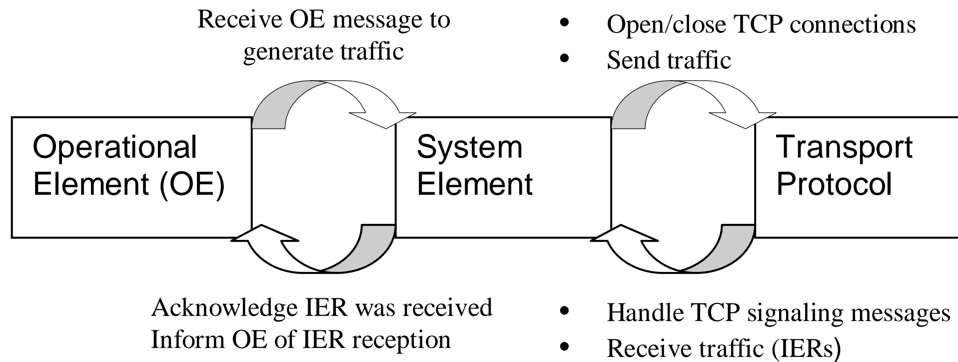


Figure 4-8: High-Level Functions of "se_tcp" Module

The "se_tcp" module uses a single process model called "se_trafgen" which is developed in the following subsections (although it is possible to have multiple process models to perform the same function). For further details, refer to the OPNET online documentation (see the section titled "Process Domain" under the Modeling Concepts menu).

4.3.3.2 Events

The Simulation Kernel (e.g., Failure/Recovery interrupts) or another process within the same process hierarchy may call upon the *se_trafgen* process model to respond to an interrupt. In both cases, however, an event must first occur for the *se_tcp* module that encompasses the process model. Logical events may be generated from three types of sources:

1. modules outside the node model
2. other process models within the same node model
3. the process model itself

There is no general method for determining the interrupts of a process model; however, the activities of the encompassing module (in this case *se_tcp*) as a whole and the interactions of the module are a good starting point. The first goal of this stage is simply to determine which logical events this process model must be prepared to receive.

[30] Please refer to Figure 4-16: Workflow Diagram for SE Process Model of the Model Development Guide v3.0.

Table 4-2 lists all the possible events that the *se_trafgen* process model can receive, their source, and the communication mechanism.

Table 4-2: Event Description Table

Logical Event	Event Name	Event Description
Generate traffic	OE_INT	This event describes the *OE*s informing the se_tcp to fire an IER.
IER Acknowledgement	IER_ACK	This event is the acknowledgement of an IER
Receive traffic	INCOMING_PKT	This event is the reception of the packet from the lower layers.
Receive TCP signaling	TCP_MESSAGE	These are the *tcp* handshake messages that are sent from the *tcp* module.
Device failure	FAILURE	This is the failure information sent to the se_tcp module from the simulation kernel.
Device recovery	RECOVERY	This is the recovery information sent to the se_tcp module from the simulation kernel.

The following table lists the events identified in the table above, with their source and the interrupt type used by the source to inform the "se_tcp" of the event.

Table 4-3: Event Communication Mechanisms

Event Name	Source		Communication Mechanism[31]
	Node	Module	
OE_INT	OE	oe	Remote Interrupt
IER_ACK	Current	tcp	Stream interrupts
INCOMING_PKT	Current	tcp	Stream Interrupt
TCP_MESSAGE	Current	tcp	Stream interrupt
FAILURE	Failure Recovery	n/a	Failure Interrupt
RECOVERY	Failure Recovery	n/a	Recovery interrupt

4.3.3.3 States

Now, the state decomposition must be performed that forms the basis of a state transition diagram (STD) that is represented by a process model in OPNET. The goal here is to define a set

[31] This is not the only possible way that this communication can be executed; there might be other ways, although they are not discussed here. For further details, please refer to the OPNET online documentation (i.e., the section titled "Communication Mechanisms" under the Modeling Concepts menu).

of discrete states that will later be connected with transitions to form an STD. At this point, only the states need be identified.

The guidelines are those mentioned in the OPNET online documentation.[32] The following table lists all the states this process model may have and its description. All these states are "Un-forced" or red states where the process rests. The "Forced" or the green states are incorporated for convenience and clarity of execution.

Table 4-4: State Description Table

State Name	State Type	Description
wait	Un-forced	Waiting for an interrupt from interfacing module(s) or from simulation kernel.
failed	Un-forced	Waiting for an interrupt from the simulation kernel to recover the node.

4.3.3.4 Event Response Table

For most process models, it is only possible for a subset of the logical events to occur while the process is located in a given state. This is generally because the involvement of the process itself is required in the interactions that result in the event. For example, in this process, a "recovery" event in the "wait" state is not possible because the device has not failed as yet. The following table enumerates which events are possible/desirable in which states.

Table 4-5: Event Feasibility Table

State Name	Logical Event	Feasibility
wait	Generate traffic	Feasible
	Receive traffic	Feasible
	Receive TCP signaling	Feasible
	Failure	Feasible
	Recovery	Not feasible
failed	Generate traffic	Not feasible
	Receive traffic	Not feasible
	Receive TCP signaling	Not feasible
	Failure	Not feasible
	Recovery	Feasible

In addition to "wait" and "failed," other forced states will be introduced in this process model to act as the placeholder for the code, and to handle the *se_trafgen*'s functionality. These *forced (green)* states are:

[32] See the subsection titled Process Modeling Methodology in the section titled Process Domain, under the Modeling Concepts menu.

- **open_conn.** This state performs the function of opening the TCP connection for every IER to be sent by the se_tcp.
- **rcv_pkt.** This state handles the reception of packets from the lower layers.
- **process_message.** This state handles the TCP *handshake* packets received from the tcp module.
- **init.** This state creates lists to store the client and the server connection handles and also creates segmentation and reassembly buffers.

In addition to these states, there is a precursor state *wait_for_tcp* that ensures that the TCP protocol has been initialized before the code in the *init* state is executed.

Once the feasible events associated with each state for the process model are determined, the next step is to develop an event response table that describes the process' possible courses of action for each feasible state-event pair. The following table lists every feasible state-event pair in the two left columns. For each such pair, at least one transition is defined.

Table 4-6: Event Response Table

<table>
<tr><th>Current State</th><th>Logical Event</th><th colspan="2">Condition</th><th>Action</th><th>Interim State (Forced State)</th><th>Next State</th></tr>
<tr><td rowspan="6">wait</td><td>Generate traffic</td><td colspan="2">None</td><td>Open TCP connection</td><td>open_conn</td><td>wait</td></tr>
<tr><td>Receive traffic</td><td colspan="2">None</td><td>Put the packet in the reassembly buffer and close the TCP connection</td><td>rcv_pkt</td><td>wait</td></tr>
<tr><td rowspan="4">Receive TCP signaling</td><td colspan="2">To open a new connection</td><td>Open a new server connection</td><td>process_message</td><td>wait</td></tr>
<tr><td rowspan="3">Informing the status of an existing connection</td><td>established</td><td>Send the packet and then close the connection</td><td>process_message</td><td>wait</td></tr>
<tr><td>close</td><td>Inform the OE that the IER is received successfully</td><td>process_message</td><td>wait</td></tr>
<tr><td>aborted</td><td>Inform OE of the IER failure that the connection aborted</td><td>process_message</td><td>wait</td></tr>
<tr><td></td><td>Device Failure</td><td colspan="2">None</td><td>Free up the IER and TCP connection related memory. Set the availability of the device as non-available.</td><td>None</td><td>failed</td></tr>
<tr><td>failed</td><td>Device</td><td colspan="2">None</td><td>Set the</td><td>None</td><td>wait</td></tr>
</table>

Current State	Logical Event	Condition	Action	Interim State (Forced State)	Next State
	Recovery		availability status of the device *available*.		

The process model based on the previous table should look similar to that in Figure 4-9:

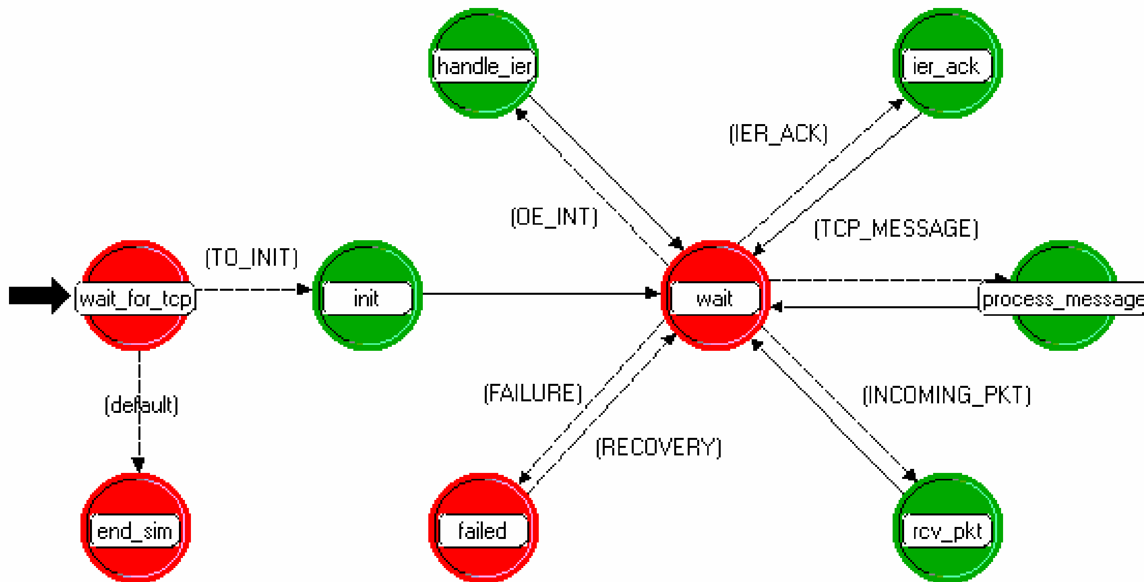


Figure 4-9: se_trafgen Process Model

4.3.4 Implementation

The following sections discuss functions that each state must perform and associated code snippets. This subsection touches upon the code for all the important functions of this end-device, but does not include all code that may be written for the end-device model to be complete.

The code is written for individual states of the process model, the compilation of which produces the "C" code representation of the process model.

4.3.4.1 Open Connection State Implementation

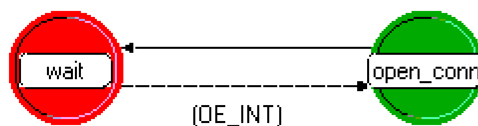


Figure 4-10: Open Connection State

The execution should come to this state from the "wait" state, on reception of a stream interrupt from the tcp module. The transition for this state is "INCOMING_PKT," which is defined in the header block (of the OPNET Modeler process editor) as—

```
#define OE_INT          ((intrpt == OPC_INTRPT_REMOTE) && !strcmp (icitype, "oe_se"))
```

In the enter execs of this state, the following functions are performed:

1. **Creating a Packet.** The information regarding the IER is retrieved from the ICI associated with the interrupt; after that, a packet with format "data" is created. In this packet, information related to the IER as well as the current node is set as follows:

```
code = op_intrpt_code();

/* remote interrupts are received from the OE  */
/* Get the interrupt code */
/* the code should tell us what to do */
if (code == OE_SE_IER_SEND)
    {
    op_ici_attr_get(iciptr, "ier_parameters_ptr", &ierp);

    if (!_strcmpi(ierp->ier_desc, "DATA"))
        {
        /* read information from the ici */
        op_ici_attr_get(iciptr, "consumer_pf_addr", &dest_node);

        /* evaluate priority */
        priority = precedence_evaluate(ierp->ier_priority);

        if ( (connInfo = getConnection(dest_node, priority)) == OPC_NIL )
            {
            connInfo = createConnection(dest_node, priority, CONNACTIVE);
            }

        ierInfo = createIerInfo(ierp, priority,connInfo->connectionId);
        if (connInfo->state == CONNRQSTOPEN)
            {
            scheduleIer(connInfo, ierInfo);
            }
        else
            {
            scheduleAndSendIer(connInfo, ierInfo);
            }

        setConnTimer(connInfo);
        }

    /* Note - don't delete the ici from the OE - it expects it still to be ok */
    }
```

2. **Registering with TCP API Package.**[33] When an application registers itself with the API package, it is returned as a handle that contains relevant data to accomplish all subsequent interaction with TCP. The registration process, by itself, discovers the TCP layer to which the application is connected and store the TCP Object ID in the interface handle. Also registered in the same handle is a pointer to the next available local port on the TCP layer. This procedure does not facilitate reusing port values but always increments the next available local port. This is performed by calling the OPNET tcp api in the following code snippet:

```
clientConnInfo->tcp_handle = tcp_app_register (op_id_self ());
```

[33] Please refer to the OPNET Modeler online documentation, section on Model Library → Standard → TCP Model User Guide → Model Interfaces → Application Layer Interfacing for details on the use of these APIs.

3. **Open a TCP connection.** In this state, it opens a TCP connection to the node whose IP address equals the given remote address on given local and remote ports. TCP connections must be opened in active[34] mode. "Command" passed as an argument to this function is used to distinguish between the active and passive modes. Because this is a client connection, it is opened in an active mode. The NW_TCP_PORT is the default local port on which the connection could be opened; the user can define its value. The connection id returned is then stored in a list with other connection information. This is performed in the following code snippet using the TCP API:

```
    /* open a new passive connection */
    connInfo->connectionId = tcp_connection_open
    (&(connInfo->tcp_handle), 0, -1, NW_TCP_PORT, TCPC_COMMAND_OPEN_PASSIVE, 0);
    currentPassiveConnection = connInfo;
    }
connInfo->state = CONNRQSTOPEN;
op_prg_list_insert (connection_info, connInfo, OPC_LISTPOS_TAIL);
```

4.3.4.2 *Receive Traffic State Implementation*

Figure 4-11: Receive Traffic State

The execution should come to the *rcv_pkt* state from the *wait* state, on reception of a stream interrupt from the tcp module. The transition for this state is *INCOMING_PKT*, which the header block defines as—

```
#define INCOMING_PKT    (intrpt == OPC_INTRPT_STRM)
```

The *rcv_pkt* state performs the following functions:

First, it puts the packet received into the reassembly buffer and then tries to remove a complete packet from this buffer. If this state cannot reassemble a packet completely, it destroys the packet. The *OE* then reports the IER as received after it receives a close connection message.

[34] Please refer to the OPNET Modeler online documentation, section on Model Library → Standard → TCP Model User Guide → TCP Commands and Indications for details.

```
    /* Discard incoming data packet and send IER acknowledgement interrupt back to
sending node */

    /* We put the packet in the reassembly buffer. If a full IER has been   */
    /* reassembled, we take the packet out and destroy it. */

        op_sar_rsmbuf_seg_insert (rsmbuf_handle, pkt);
        pkt = op_pk_get(code);
        } /* ends new while loop for getting more than one packet from stream */

    /* Initialize the pointer to the packet, used later to determine if a   */
    /* packet has been reassembled.                                          */
    /* not needed because we have a separate temp packet* to handle the
    packet from the SAR buffer*/
/*  pkt = OPC_NIL;*/

    /* Only one packet at most can get reassembled as a result of a single  */

    /* insertion. Therefore, we do not need to check the complete pk count. */
    sar_pkt = op_sar_rsmbuf_pk_remove (rsmbuf_handle);
    /* while loop added to handle more than one completed packet in the reassembly buffer i
    while (sar_pkt != OPC_NIL)
        {
        if (sar_pkt != OPC_NIL) {
        /* get the needed information from the packet and destroy the packet */
            op_pk_nfd_get(sar_pkt, "APPLICATION ID", &ier_id);
            op_pk_nfd_get(sar_pkt, "PRECEDENCE", &priority);

            pkIci_p = op_pk_ici_get(sar_pkt);
            op_ici_attr_get(pkIci_p, "sourceNode",&sourceNode);
            op_ici_attr_get(pkIci_p, "sourceConnId",&sourceConnId);
            destroyIerPacket(sar_pkt);
```

After this, the server connection is closed, as shown in the code snippet below:

```
/*
 *  Close a connection
 */
void
closeConnection(TrafgenT_Connection *connInfo)
    {
    FIN(closeConnection(connInfo));
    connInfo->state = CONNRQSTCLOSE;
    tcp_connection_close (connInfo->tcp_handle);
    FOUT;
    }
```

4.3.4.3 Process Message State Implementation

Figure 4-12: Process Message State

The execution should come to the *process_message* state from the *wait* state, on reception of a stream interrupt from the *tcp* module. The transition[35] for this state is *TCP_MESSAGE*, defined in the header block as—

```
#define TCP_MESSAGE     ((intrpt == OPC_INTRPT_REMOTE) && strcmp (icitype, "oe_se") && strcmp (icitype, "ier_ack"))
```

The following functions are performed in this state:

1. A new server connection is opened if the associated ICI indicated a new connection to be opened. It calls a function *se_open_server_tcp_conn* defined in the function block.

[35] We have defined this transition as one in which the interrupt received is a remote interrupt, and the source of this interrupt is an IER (by checking that the ici type is "ier_ack").

```
if (!strcmp (icitype, "tcp_open_ind")) {

    /*This machine is acting as a server for an incoming IER        */
    /* the current passive connection is now bound to an active connection */
    /*spawn a new passive connection to receive futher requests */
    /*indicate to TCP that we are ready to  */
    /*receive data.          */

    /* uninstall the ici before we use the TCP API or else we get an error later */
    op_ici_install(OPC_NIL);

    /*we will receive one packet because data IERs consist of a sole packet.    */
    tcp_receive_command_send (currentPassiveConnection->tcp_handle, 1);

    /*because the TCP api uses forced interrupts to open a connection   */
    /*we have to schedule a procedure interrupt to open the new passive */
    /*connection.    */

    op_intrpt_schedule_call (op_sim_time (), 0, se_open_server_tcp_conn, 0);


    /* Ideally we would like to find the destNode objId and the priority of the
    /* connection here and specify it in the new bound connection info
    bindPassiveConnection(destnode, priority);
    Until we figure out how to do this, we will retrieve this info from the ici
    accompanying the incoming packets.
    */


    /* Note - do not destroy the open indicator ICI or else TCP complains */
}
```

This function opens a TCP connection to the node whose IP address equals the given remote address on given local and remote ports. TCP connections must be opened in passive mode. “Command” passed as an argument to this function is used to distinguish between these two modes. Because this is a server connection, it is opened in a passive mode. The connection id returned is then stored in a list with other connection information. This is performed in the following code snippet:

```
/*
 * Open a new Passive connection
 */
static void
se_open_server_tcp_conn (Vartype * ptr, int code) {
    FIN(se_open_server_tcp_conn ( ptr, code));
    createConnection(-1, -1, CONNPASSIVE);
    FOUT;
}
```

Finally, after the connection is open and the transport connection is established, the application processes are ready to receive messages from peers. This information must be passed on to TCP, and the following tcp api accomplishes that operation.

```
/*we will receive one packet because data IERs consist of a sole packet.    */
tcp_receive_command_send (currentPassiveConnection->tcp_handle, 1);
```

2. This state handles the TCP control messages as well. Based on the type of message, it is *switched* (using the C switch/case statements) to the appropriate case.

For the tcp message indicating the successful establishment of the connection, the data is sent and the connection is closed (see below):

```
switch (status) {
        case TCPC_IND_ESTAB:
            /* Send any scheduled IERs and also set up to receive */
            /* an IER from the far end */
            connInfo->state = CONNOPEN;
            sendScheduledIers(connInfo);
            tcp_receive_command_send (connInfo->tcp_handle, 1);

            break;
```

Once the TCP close control message is received, the OE is informed that the IER[36] is received, and related memory for the connection is freed up (see below).

```
case TCPC_IND_CLOSED:
    /* Check for any IERs that were not acknowledged or sent
       before the connection was closed */
    while ((ierInfo = removeNextIer(connInfo)) != OPC_NIL) {
        notifyOE(ierInfo->ier_p,NWC_INFORM_SRC_OE_FAIL,"Unable to transmit TCP message????");
        destroyIerInfo(ierInfo);
    }

    destroyConnection(connIndex);

    break;
```

For the connections that are aborted, the OE is informed of the IER failure, and related memory is freed up.

```
case TCPC_IND_ABORTED:
    /*Connection failed.  Record IER failure and free memory.*/

    while ((ierInfo = removeNextIer(connInfo)) != OPC_NIL) {
        notifyOE(ierInfo->ier_p,NWC_INFORM_SRC_OE_FAIL,"Unable to transmit TCP message????");
        destroyIerInfo(ierInfo);
    }

    destroyConnection(connIndex);

    break;
```

4.3.4.4 Failure State Implementation

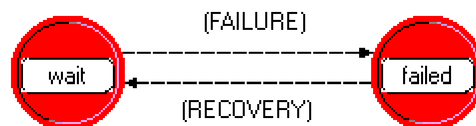


Figure 4-13: Failure State

The execution should come to the *failed* state from the *wait* state upon reception of a failure interrupt from the kernel. The transition for this state is *FAILURE*, defined in the header block as—

```
#define FAILURE          (intrpt == OPC_INTRPT_FAIL)
```

[36] Please refer to "Appendix G: Constants" of the Model Development Guide v3.0 on details on the codes used by the OE to communicate with SE and vice-versa.

In this state, the first thing that is done is to set the availability status attribute as disabled, as shown below:

```
/* Computer not available now */
op_ima_obj_attr_set (my_nd_id, "availability_status", OPC_BOOLINT_DISABLED);
```

The next important action in this state is to fail the IERs for which the connection is open. Because the supporting transport protocol is TCP, the IER is said to be "not received" until a TCP close acknowledgement is received, meaning the IER data may have reached the destination but may still be marked as failed. The following code fails the IERs with open connections and frees up any related memory:

```
for (index = 0; index < num_conxns_open; index++) {
    connInfo = (TrafgenT_Connection *)op_prg_list_remove (connection_info, OPC_LISTPOS_HEAD);

    /* Inform the OE about the device failure and it will be responsible for the updating of the thread and IER statistics */
    while( (ierInfo = removeNextIer(connInfo)) != OPC_NIL) {

        notifyOE(ierInfo->ier_p, NWC_DEVICE_FAILURE,"TCP unable to tranmsit the message due to device failure.");
        destroyIerInfo(ierInfo);
    }

    /* Free up the associated memory */
    destroyConnInfo(connInfo);

}
```

The execution goes back to the "wait" state when the recovery interrupt is received in this state.

4.4 WIRED END DEVICE EXAMPLE 2

4.4.1 Overview

This subsection explains the construction of an end-system device using an example. The example end-system device is a computer that generates data IERs over TCP/IP with Ethernet as the MAC technology. The computer is built from an existing OPNET Standard (COTS) device—an ethernet_wkstn_adv model.

4.4.2 Steps

Because this is an end-system device, it needs a module that communicates with the OE to get the IER information—the SE module. The SE module generates data IERs upon receiving remote interrupts from the OE in its OPFAC. It generates the IERs and forwards them to the network protocol stack, where they are sent out on to the network through the Ethernet interfaces. Because TCP is an acknowledgement-based scheme, the end-system device sending the IER marks it as received when the connection close request, for the connection over which IER was sent, is received. The SE module also handles the failure/recovery of the computer. Because it transmits only data IERs and uses TCP/IP as the underlying protocol, it needs the TCP/IP protocol stack. It also uses Ethernet as the MAC technology.

Rather than assembling all the modules needed for communication in the OPNET simulation environment, begin by modifying an OPNET Standard (COTS) model —an ethernet_wkstn_adv model. Not all components need to be built by modifying an existing model; components can be built from scratch as well. The ethernet_wkstn_adv node model is shown below:

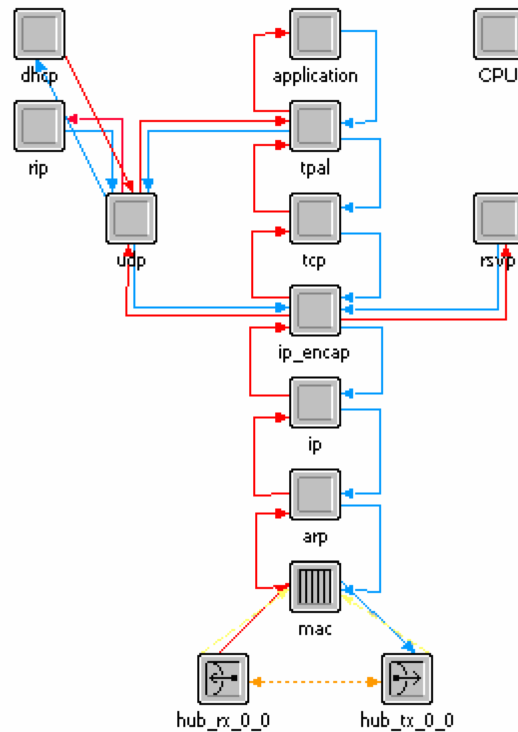


Figure 4-14: Ethernet_wkstn_adv-Node Model

Step 1. From the ethernet_wkstn_adv node, the CPU, application, RSVP, UDP, RIP, Dynamic Host Configuration Protocol (DHCP) and TPAL modules must be removed:

- In a node editor window, open the ethernet_wkstn_adv node model.
- Select the mentioned modules and hit CTRL-X.

Note that the packet streams connected to and from the modules are deleted automatically.

Step 2. Add the SE module on top of the TCP module and connect them to the incoming and outgoing packet streams:

- Left-click the "create processor" toolbar button.
- Left-click the area above the TCP module. This creates a processor module on top of the TCP module.
- Right-click the created module and name the module *se_tcp* by modifying the module attributes.
- Left-click the "create packet stream" toolbar button.
- Create an incoming packet stream by first left-clicking the *tcp* module and then the *se_tcp* module.
- Create an outgoing packet stream by first left-clicking the *se_tcp* module and then the *tcp* module.

The node model for the computer should look like Figure 4-15.

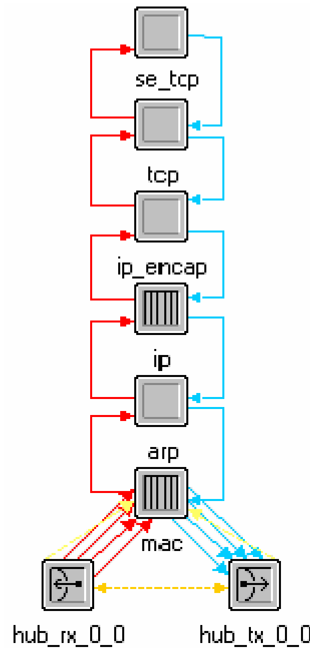


Figure 4-15: Computer-Node Model

Step 3. The "Model Attributes" for this node must be set as follows:

- Under the "Interfaces" menu, choose the "Model Attributes" option.
- Set the following attributes and their types in the "Model Attributes" table. The NETWARS program suggests that you use the already existing *public*[37] definitions of these attributes, which we have named the same as the attribute names themselves.

Table 4-7: End-System-Model Attributes

Attribute Name	Attribute Type
classification	String
equipment_type	Enumerated
availability_status	Toggle

Step 4. The SE module now should house the process model created in the following subsection:

- Right-click on the *se_tcp* module and change the "process model" attribute to be the name of the process model, *se_trafgen*, created in the next subsection.

[37] Please refer to the OPNET Product documentation, Modeler Documentation → OPNET Editors Reference → Process Editor section, for further details.

4.4.3 Process Model: SE

Figure 4-16 contains a workflow diagram of a simple SE process model.

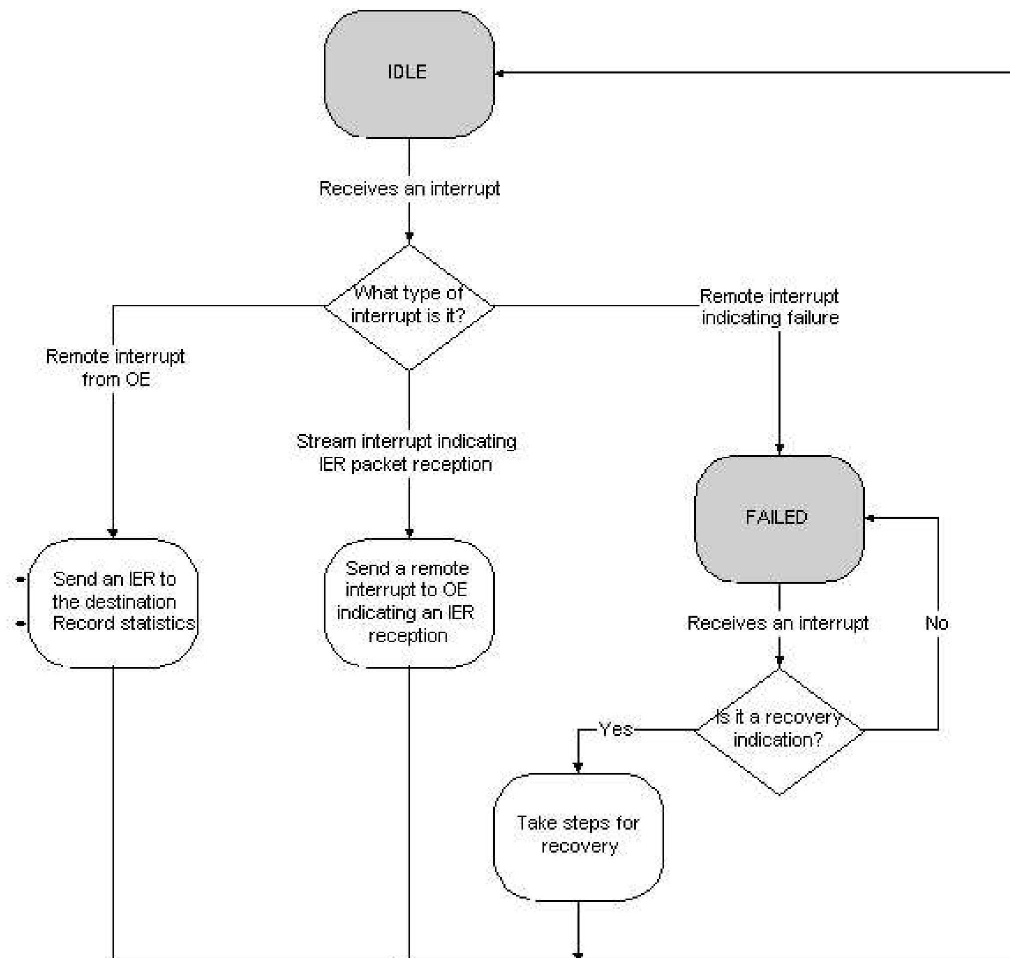


Figure 4-16: Workflow Diagram for SE Process Model

During initialization, the process reads in attribute values and creates any necessary structures, as well as obtaining pointers to the statistic files.

Referring to Figure 4-16, above, when the computer receives an interrupt from the OE to generate an IER, it transitions to the *send* state, sends the IER to the protocol stack, and goes back to the *idle* state. When it receives a failure interrupt, it transitions to the *fail* state and stays there until it receives a recovery interrupt, at which point it transitions to the *recover* state and performs the steps needed for recovery. Then it transitions back to the *idle* state.

The *se_tcp* module uses the following APIs to interface with the TCP module:

- **`tcp_connection_open ().`** To open a TCP connection with the destination

- **`tcp_receive_command_send ()`** Used by the receiving SE module to indicate to the TCP module to forward the IERs to itself
- `tcp_data_send ()`. To send IERs.

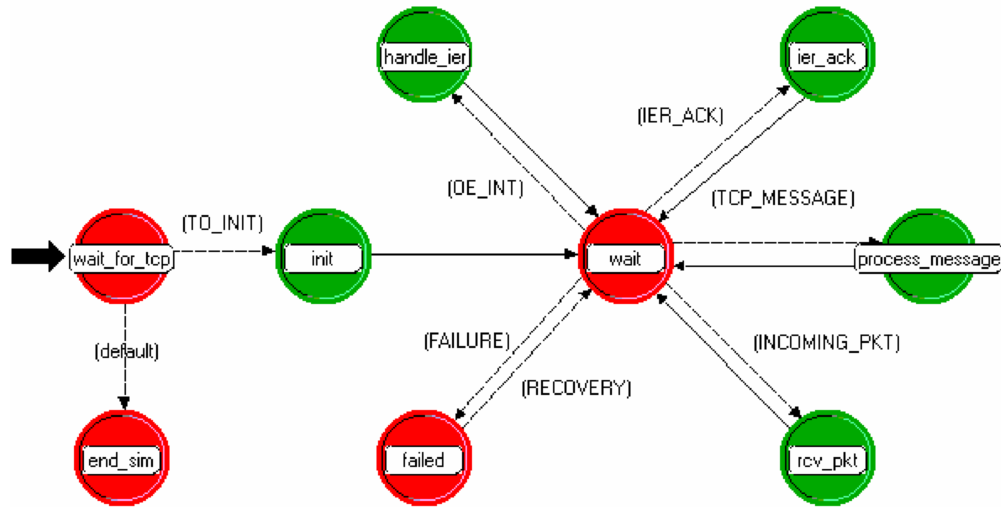


Figure 4-17: Process Model for SE Module in Computer

4.4.4 Statistics

The *se_tcp* process model is responsible for informing the OE of the failed IERs. There could be several reasons for failure in data communication, such as the TCP socket failure or congestion in networks. The *se_tcp* process model informs the OE (using the codes describes in Appendix F). The IER is "received" only when the source of the traffic (IER) receives a tcp acknowledgement (connection close indication) and code NWC_INFORM_SRC_OE_RCVD is used (in the remote interrupt) to inform the OE at the source OPFAC to collect the IER Received statistics. In the following sample code, the process model records the statistics due to TCP socket open failure (for a more detailed example, please refer *se_trafgen.pr.c* / *se_trafgen.pr.m* files).

```
case TCPC_IND_ABORTED:
    /*Connection failed.  Record IER failure and free memory.*/

    while ((ierInfo = removeNextIer(connInfo)) != OPC_NIL) {
        notifyOE(ierInfo->ier_p,NWC_INFORM_SRC_OE_FAIL,"Unable to transmit TCP message????");
        destroyIerInfo(ierInfo);
    }

    destroyConnection(connIndex);

    break;
```

Figure 4-18: Code 1-Inform OE of IER Failure, Will Record Statistics

Interfacing with the statistics, such as writing success and failure statistics, is normally accomplished through the APIs. Refer to "Appendix L: NETWARS Simulation API and Helper Functions"

4.5 LAYER 1 DEVICE EXAMPLE: BULK ENCRYPTOR

4.5.1 Overview

This subsection explains the construction of Layer 1 networking equipment using an example. The objective is to construct an encryptor device. The example networking equipment is an encryptor with two ports. It accepts packets from a classified network, encrypts the packet, and sends it over an unclassified network. When it accepts packets from the unclassified network, it decrypts the packet and forwards it on to the classified network. It encrypts only the payload of the packet. The header is left intact. The encryptor model is constructed from scratch.

4.5.2 Steps

Step 1. Two transceiver pairs are created:

- In a new node editor window, using “create point-to-point receiver,” place two point-to-point receiver and transmitter pairs and “create point-to-point transmitter” toolbar buttons.
- Once the transceiver modules are in place, create logical connections between them by using the “create logical tx/rx association” toolbar button.

Step 2. A processor to house the encryptor process model is created and connected to the transceiver pair:

- Left-click the “create processor” toolbar button.
- Left-click the area above the transceiver modules.
- Right-click the created module and name the module “encryptor” by modifying the module attributes.
- Left-click the “create packet stream” toolbar button.
- Create incoming packet streams by first left-clicking on the receiver modules and then on the encryptor module.
- Create outgoing packet streams by first left-clicking on the encryptor module and then on the transmitter modules.

The resulting encryptor device looks like Figure 4-19.

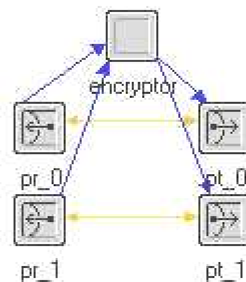


Figure 4-19: Encryptor-Node Model

4.5.3 Process Model

Step 1. A workflow diagram of a simple encryptor model is designed.

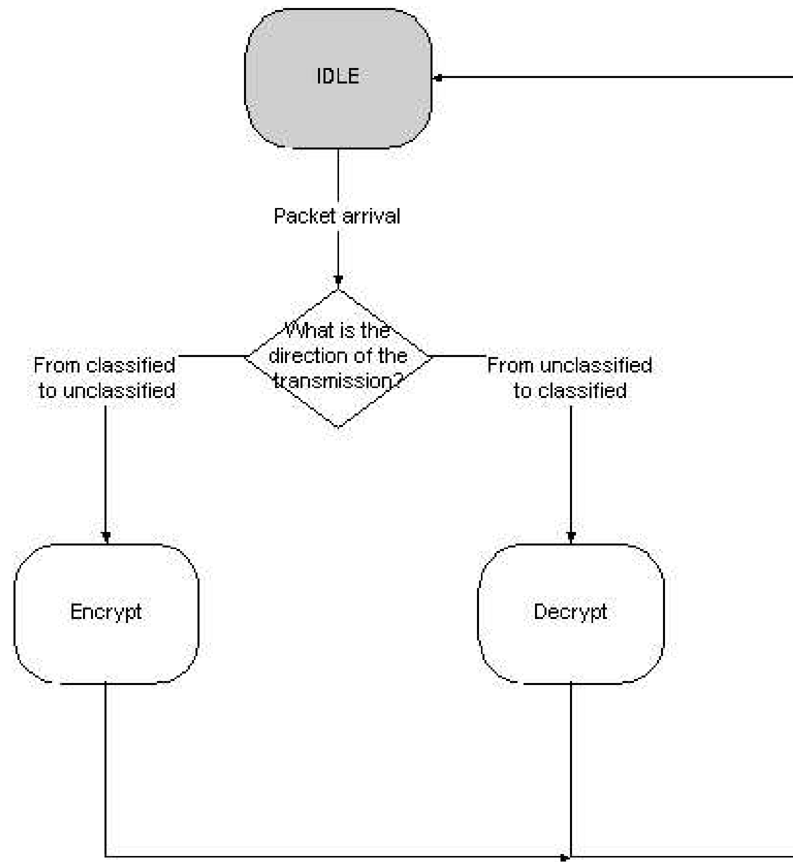


Figure 4-20: Data Flow for Encryptor

Step 2. The encryptor performs its initialization functions in the *init* state and transitions to the *idle* state, where it waits for a packet. When the packet arrives, it checks the direction from which the packet is coming. If the packet is from a classified network and going to an unclassified network, it encrypts the packet and sends it on the appropriate output interface. It decrypts the packet for a packet going in the opposite direction.

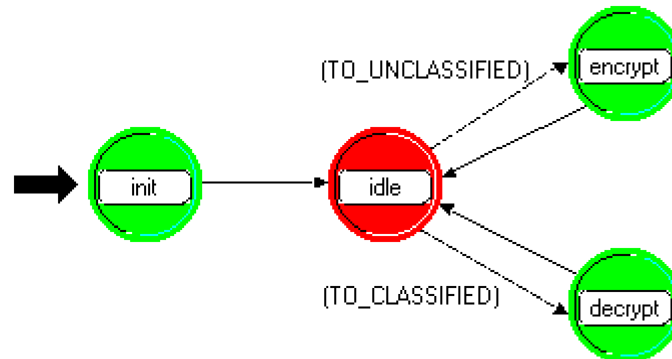


Figure 4-21: Process Model for Encryptor

Figure 4-22 shows a sample code block from the *encrypt* state:

```
/*  Get the index of the stream on which the    */
/*  packet was received.                        */
stream = op_intrpt_strm ();

/*  Get the packet  */
pkptr = op_pk_get (stream);

/*  Encrypt the packet's payload    */
enc_pkptr = get_encrypted_packet (pkptr, encryption_info_ptr);

/*  Send the encrypted packet on the output interface   */
op_pk_send (enc_pkptr, (1 - stream));
```

Figure 4-22: Code 2-Encrypting a Packet

The model developer must write the function `get_encrypted_packet ()` that takes in a packet and encrypts it. All other functions are OPNET kernel procedures. It is important to note that the code listed above uses the expression *(1 - stream)*, which only works if all stream numbers are zero and one, and both the incoming and outgoing stream connected to a particular rx/tx pair are given the same stream number (i.e., if *pr_0* is connected by incoming stream zero, then *pt_0* must be connected by outgoing stream zero). Similarly, *pr_1* and *pt_1* should both use stream number one. An example cryptographic device, which performs similar functionality, is the KG-194 node model; the process model is crypto.pr.m (these cryptographic device models are available with NETWARS version 3.0).

4.6 LAYER 2 DEVICE EXAMPLE: MULTI-SERVICE SWITCH

4.6.1 Overview

The example considered here is a multi-service switch that has circuit-switched and ATM interfaces. The objective is to explain the construction of Layer 2 networking equipment using an example. The example networking equipment is a multi-service switch that is used for interfacing a circuit-switched voice network with an ATM data network. This is a switch with one ATM and two circuit-switched interfaces. It needs two pairs of circuit-switched transceivers and one pair of ATM transceiver. It also has the ATM protocol stack. In addition to these modules, a module for switching is needed. This device needs one ATM port and the ATM protocol stack. Therefore, this node is built by modifying an OPNET Standard (COTS) model—*atm_uni_dest_adv*. The atm_uni_dest_adv node model is shown in Figure 4-23:

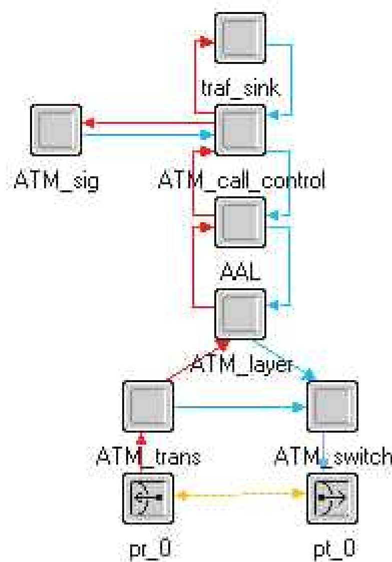


Figure 4-23: Atm_uni_dest_adv Switch-Node Model

4.6.2 Steps

Step 1. From the atm_uni_dest_adv node model, the traf_sink module is removed:

- In the node editor window, open the atm_uni_dest_adv model.
- Select the module mentioned above and hit Ctrl-X to remove them from the workspace.

Step 2. This device has two circuit-switched ports. Therefore, two transmitters and receivers are added:

- Left-click the create point-to-point receiver tool button.
- Left-click in the node editor workspace to create two instances of the point-to-point receiver.
- In a similar way, create two transmitter objects.
- Associate the transceiver pairs with a transmitter/receiver association object.

Step 3. Two processor modules are created, and the circuit-switched ports are connected to one of them:

- Place a processor module in the workspace and name it voice_dispatch.
- Connect the transmitters and receivers to the voice_dispatch module.
- Create another processor and call it voatm.

Step 4: Connect the circuit-switched ports to the ATM stack through the voatm and voice_dispatch modules using packet streams.

The completed node model looks like Figure 4-24.

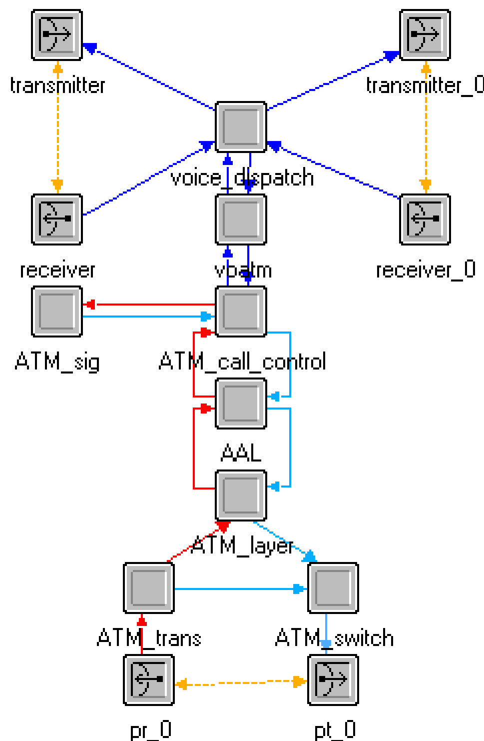


Figure 4-24: Multi-Service Switch-Node Model

Step 6. Create process models for the voatm and voice_dispatch modules and set the *process model* attributes for these two modules appropriately.

Step 7. Add the required NETWARS attributes. Refer to Step 3 under Subsection 4.4.2.

4.6.3 Process Models: Voice Dispatch and Voice Over ATM

- **voice_dispatch.** Takes the ckswpkt packet and passes it to the appropriate convergent module. A multi-service switch like this can potentially have additional types of interfaces like IP and Frame Relay. There are different convergent modules depending on the protocol stack desired. In the example, there is only one convergent module, the voatm module. So, the voice_dispatch module forwards call-setup packets to the voatm

module.

Similarly, when the voice_dispatch module receives packets from the voatm module, it must determine which one of the circuit-switched interfaces to send the packet on.

- **voatm.** When the voice_dispatch module forwards the packet to the voatm module, the voatm module generates ATM cells at a rate that depends on the call generation rate and forwards the packets to the ATM stack. When the voatm module gets data packets from the ATM stack destined to one of the circuit-switched interfaces, it destroys the data packets and sends the appropriate control packets (call-setup, ack) to the voice_dispatch module.

  The voice module is responsible for informing ATM of the circuit setup. The voice call setup message must be translated to the appropriate ATM call setup message for circuit reservation. Likewise, on the other end, the ATM device must inform the voatm module of the call setup message and forward it on. On the "source" side, there needs to be flooding on the other circuit switch interface.

4.7 LAYER 3 DEVICE EXAMPLE: CUSTOM ROUTER

4.7.1 Overview

This subsection explains the construction of Layer 3 networking equipment using an example. The example considered is an IP router with one serial port, one Ethernet port, and a custom routing protocol (called MRP,for Military Routing Protocol) running over TCP. The router is built from an existing OPNET Standard (COTS) device—a CS_1005_1s_e_sl_adv router model.

4.7.2 Steps

This router has a custom routing protocol, MRP, running on top of TCP. The router has two ports—one Ethernet port and one SLIP port. The router is constructed from an existing OPNET Standard (COTS) model—a CS_1005_1s_e_sl_adv router model. Rather than assembling all the modules needed for communication in the OPNET simulation environment, begin by modifying an OPNET Standard (COTS) model— a CS_1005_1s_e_sl_adv router model. The CS_1005_1s_e_sl_adv node model is shown below, in Figure 4-25:

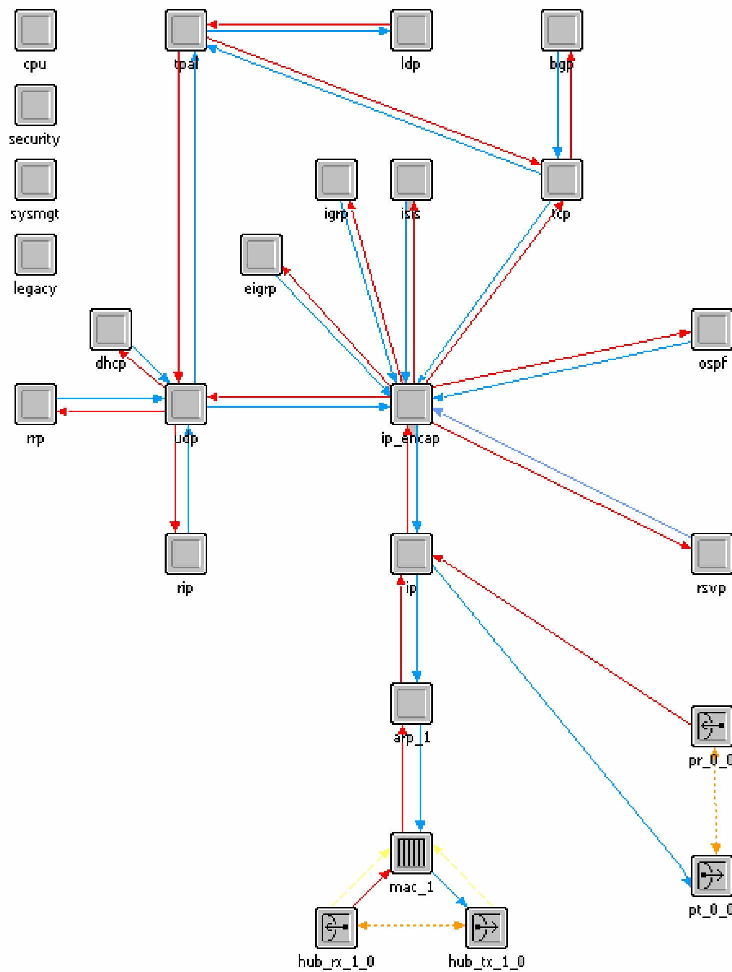


Figure 4-25: CS_1005_1s_e_sl_adv Router-Node Model

Step 1. The custom routing protocol module is added on top of the TCP module and is connected to it with an incoming and outgoing packet stream:

- Left-click the create processor toolbar button.
- Left-click the area above the tcp module. This creates a processor module on top of the tcp module.
- Right-click the created module and name the module "mrp" by modifying the module attributes.
- Left-click the create packet stream toolbar button.
- Create an incoming packet stream by first left-clicking the tcp module and then the mrp module.
- Create an outgoing packet stream by first left-clicking the mrp module and then the tcp module.

After the changes have been made, the node model for the router looks like Figure 4-26.

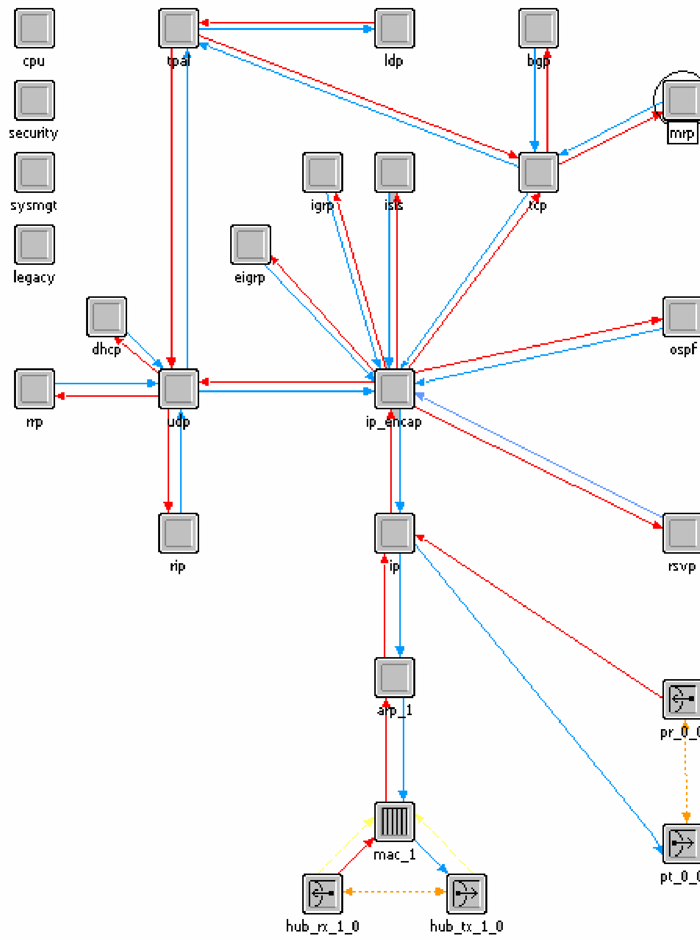


Figure 4-26: Router with Custom Routing Protocol-Node Model

Step 2. Add the required NETWARS attributes. Refer to Step 3 under Subsection 4.4.2.

4.7.3 Process Model: Custom Routing Protocol

The process model that implements the custom routing protocol looks like Figure 4-27.

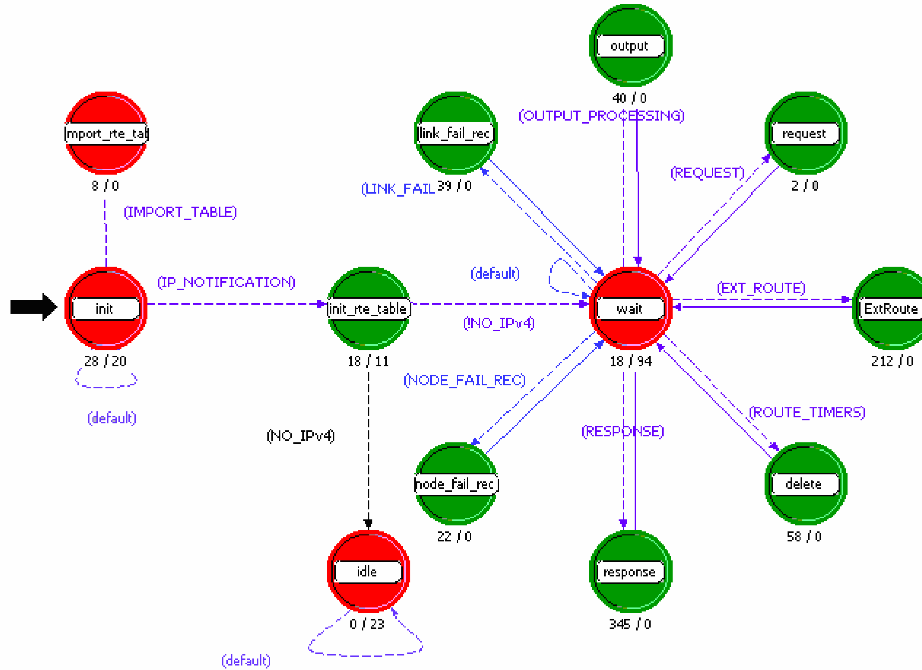


Figure 4-27: Process Model for Custom Routing Protocol

In the *init* state, the custom routing protocol registers itself as an IP higher-layer protocol using a call to the function `Ip_Higher_Layer_Protocol_Register()`. It must also register itself in the IP common routing table with a call to the function `Ip_Cmn_Rte_Table_Custom_Rte_Protocol_Register()`.

When the IP process mode has been initialized, the custom routing protocol module receives a remote interrupt with code `IPC_EXT_RTE_REMOTE_INTRPT_CODE`. On receiving this remote interrupt, it transitions to the *init_rte_table state*, where it can start accessing the routing table via the process registry. Then it transitions to the *wait* state.

When the custom routing protocol receives route update messages, it makes or changes entries in the common routing table using calls to the functions:

- `Inet_Cmn_Rte_Table_Entry_Add()`
- `Inet_Cmn_Rte_Table_Entry_Delete()`
- `Inet_Cmn_Rte_Table_Entry_Update()`

These functions are defined in the external file OPNET\<rel_dir>\models\std\ip\ip_cmn_rte_table.ex.c and the prototypes for these functions are in OPNET\<rel_dir>\models\std\include\ip_cmn_rte_table.h, where <rel_dir> is the release directory (e.g., 12.0.A).

4.8 CIRCUIT-SWITCHED DEVICE EXAMPLE: END SYSTEM

4.8.1 Overview

This subsection explains the construction of a circuit-switched device using an example. The example used here is a phone (a circuit-switched end-system device) that generates calls based on its interaction with the OE. This example shows how to build the device from scratch.

4.8.2 Steps

The phone has three modules, as shown in Figure 4-28:

- An SE module that generates calls in response to interrupts from the OE
- A transmitter that supports only packets of type cktswpkt
- A receiver that supports only packets of type cktswpkt.

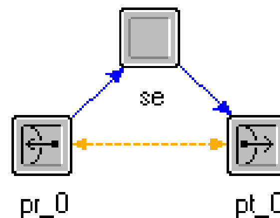


Figure 4-28: Phone-Node Model

The transmitter and the receiver are connected to the SE module by packet streams, as shown in Figure 4-28. The transmitter and receiver are logically associated with each other.

Note that in order to generate calls initiated by the standard voice application in addition to voice IERs, the device would require additional application, TPAL, and CPU modules.

Step 1. A transceiver pair is created:

- In a new node editor window, a point-to-point receiver and transmitter pair is created by using the “create point-to-point receiver” and “create point-to-point transmitter” options.
- Once the transceiver modules are in place, logical connections between them are created using the “create logical tx/rx association” option.

Step 2. A processor to house the *se* process model is created and connected to the transceiver pair:

- Left-click the “create processor” toolbar button.
- Left-click the area above the transceiver modules.
- Right-click the created module and name the module “se” by modifying the module attributes.
- Left-click the “create packet stream” toolbar button.

- Create an incoming packet stream by first left-clicking the receiver module and then on the SE module.
- Create an outgoing packet stream by first left-clicking the SE module and then the transmitter module.

Step 3. The “Model Attributes” for this node must be set as follows:

- Under the “Interfaces” menu, choose the “Model Attributes” option.
- Set the attributes and their types shown in Table 4-8 in the “Model Attributes” table.

Table 4-8. Circuit-Switched End-System Device-Model Attributes

Attribute Name	Attribute Type
equipment_type	Enumerated
availability_status	Toggle
Call Bandwidth	Double
Max Calls Allowed	Integer

Step 4. The SE module to house the process model is created in the following subsection:

- Right-click the SE module and change the “process model” attribute to be the name of the process model created in the following subsection.

4.8.3 Process Model: se

The *se* module is responsible for interacting with the OE to generate calls.

Step 1. A workflow diagram of a simple SE process model is designed.

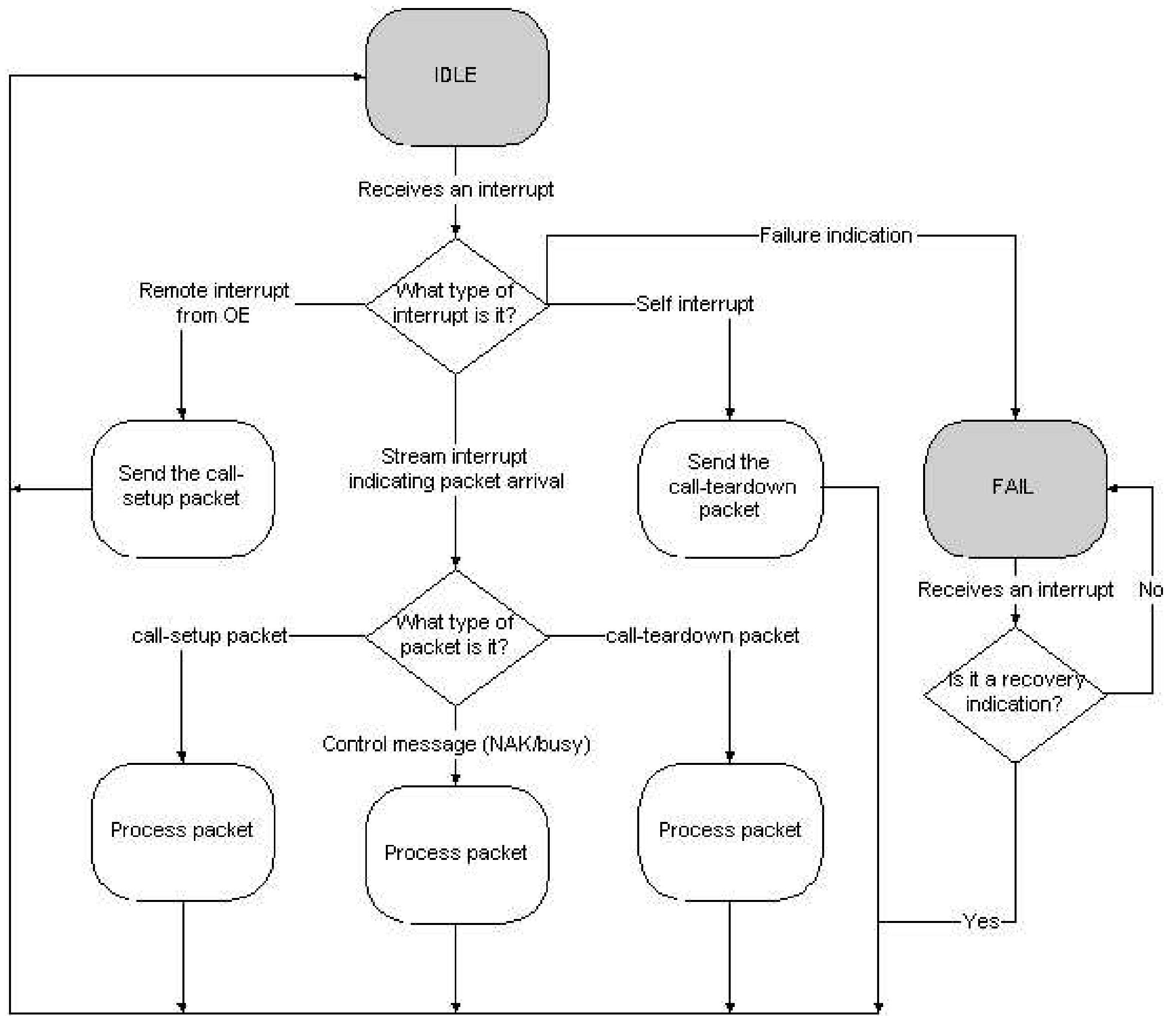


Figure 4-29: Data Flow for Phone

Step 2. The process model for the *se* module might look like that shown in Figure 4-30.

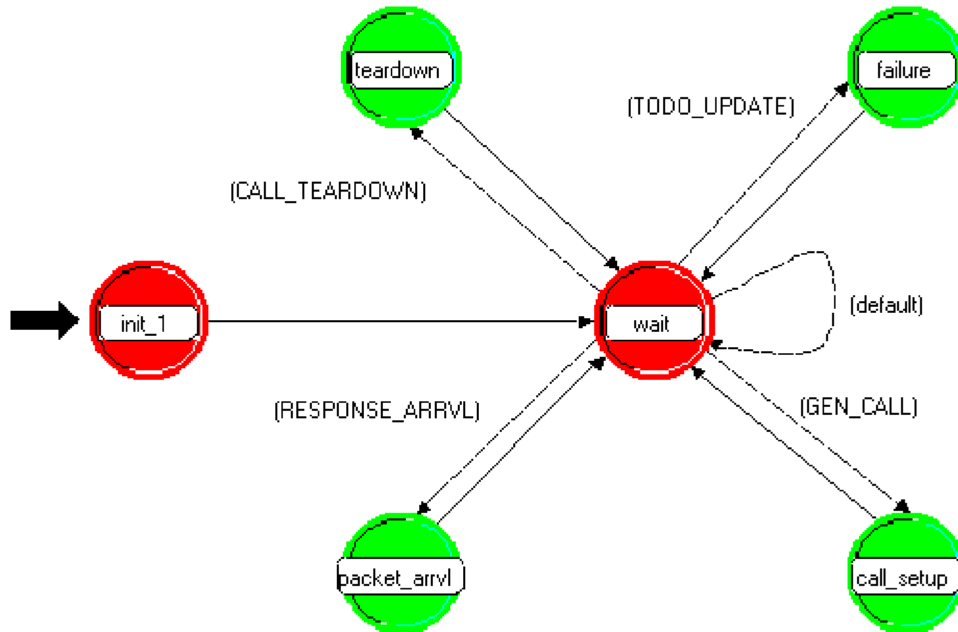


Figure 4-30: Process Model for SE Module

The initialization steps are performed in the *init_1* state. When the phone receives an interrupt from the OE requesting a call setup, the process model transitions to the *call_setup* state, gets the necessary information from the *oe_se_ici* ICI, creates a call-setup packet, and sends it to the transmitter module.

When the phone receives a packet, the process model transitions to the *packet_arrival* state and processes the packet. This packet could be an ACK packet indicating that the call was successfully set up, a Negative Acknowledgement (NACK) indicating that the call setup failed, or a request for a call setup from a remote phone. The *packet_arrival* state takes the necessary action, depending on the type of packet.

When the phone receives a failure interrupt, it transitions to the *failure* state and takes the necessary steps to handle the interrupt. It recovers when it receives a recovery interrupt.

4.9 WIRELESS DEVICE EXAMPLE

4.9.1 Overview

This subsection explains the construction of a radio device using an example. This end-system device uses the OPNET standard wireless LAN MAC model to communicate voice or non-IP data IERs. Start with the OPNET Standard (COTS) model *wlan_station_adv* node model. The *wlan_station_adv* is a simple radio device that sends out a packet to the destination specified in the *wlan_mac_intf* module using IEEE 802.11 interface. By adding an SE module to interface to the OE and setting the destination address to be that of the gateway radio device, we have a simple radio end-system device. The *wlan_station_adv* node model is shown below in Figure 4-31.

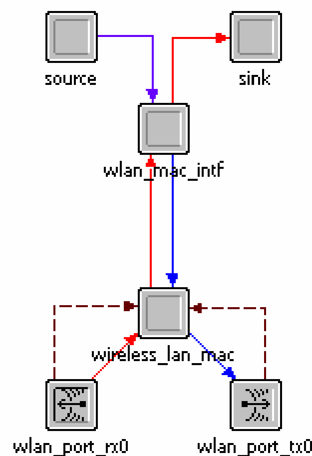


Figure 4-31: wlan_station_adv-Node Model

4.9.2 Steps

Step 1. The source and sink modules are replaced with an *se* module:

- In a node editor window, open the wlan_station_adv node model.
- Select the mentioned modules and press CTRL-X.

Note that the packet streams connected to and from the modules is deleted automatically.

Step 2. The SE module is added on top of the wlan_mac_intf module and is connected to it with an incoming and outgoing packet stream:

- Left-click the create processor toolbar button.
- Left-click the area above the wlan_mac_intf module.
- Right-click the created module and name the module *se* by modifying the module attributes.
- Left-click the create packet stream toolbar button.

- Create an incoming packet stream by first left-clicking the *wlan_mac_intf* module and then the *se* module.
- Create an outgoing packet stream by first left-clicking the *se* module and then the *wlan_mac_intf* module.

Step 3. Because this is an end-system device, it has *classification*, *equipment_type*, and *availability_status* as model attributes. Set the model attributes for this node as follows:

- Under the “Interfaces” menu, choose the “Model Attributes” option.

Set the attributes and their types shown in Table 4-9 in the Model Attributes table.

Table 4-9. Radio End-System Device-Model Attributes

Attribute Name	Attribute Type
classification	String
equipment_type	Enumerated
availability_status	Toggle

The node model looks like Figure 4-32.

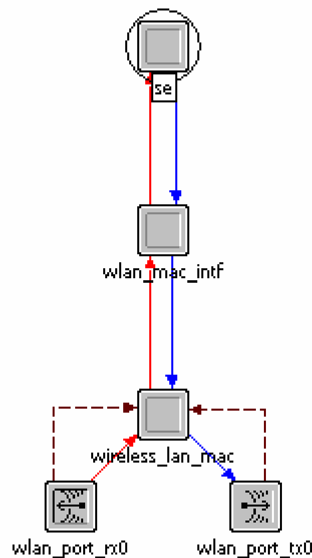


Figure 4-32: Radio SE model-Node Model

Step 4. The *se* module now houses the process model created in the following subsection:

- Right-click the *se* module and change the process model attribute to have the name of the process model.

4.9.3 SE Process Model

The process model for this device is similar to the for constructing a computer model process model except that packets are sent to the lower layer directly without using the TCP interface. Refer to the Process Model section of the example wired end device for more information.

4.10 WIRELESS DEVICE EXAMPLE 2

4.10.1 Problem Statement

The following discussion provides implementation-level guidelines for developing a radio end-device NETWARS model. Relevant aspects, such as OE-SE interaction, are presented in detail; however, other aspects of the radio itself—such as the medium access control—are left out because the details are specific to the type of radio being modeled.

The discussion aims to provide details for an radio end-device that is capable of generating both voice and data IERs.

4.10.2 High-Level Design

4.10.2.1 Node Model Development

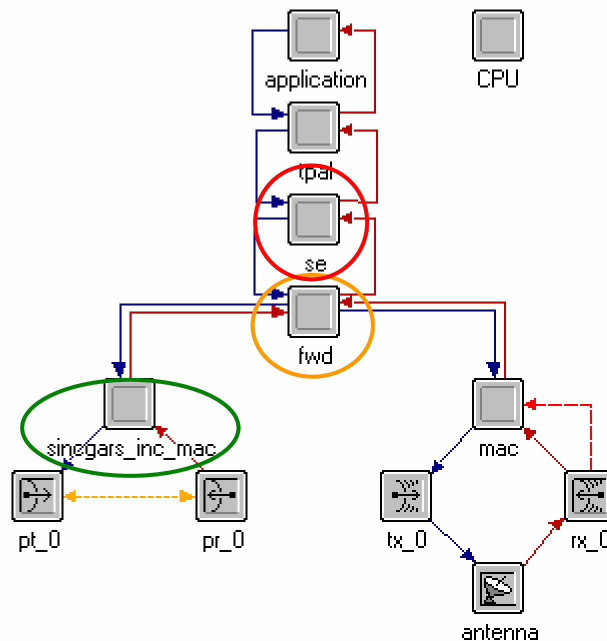


Figure 4-33. Radio End Device Node Model

The node model in Figure 4-33 above is a NETWARS sincgars_rt node model and it shows a device with two interfaces—a wired interface and a radio interface. The node is also capable of generating NETWARS IER traffic from the *se* module.

The *se* module is responsible for generating the IER and reporting the IER receptions through interaction with the OE in the OPFAC.

The *fwd* module is responsible for performing appropriate forwarding decisions—either to and from the *se* module or to and from the *mac* module.

The *mac* module is responsible for the medium access control to the wireless interface. The functions of this module depend on the technology a particular device uses. Hence, the implementation details for this module are not discussed.

4.10.3 fwd module: Detailed Design

4.10.3.1 Module Context and Functionality

This module is responsible for handling the packets that arrive either from the se module (which generates the traffic) or from the wired interface of this device. The se module is responsible for generating the traffic. The fwd module is an interfacing module between the mac and the se/wired_mac module. Based on the packets received from either of these modules, it determines the destination module and forwards it on. It is necessary to provide any required encapsulation or decapsulation so that the packet format of the packet is the one supported at the destined module.

4.10.3.2 Events

There are three different events that can happen at this module. They are:

- Receive packet from SE
- Receive packet from INC (wired interface)
- Receive packet from the MAC (radio mac).

4.10.3.3 States

Based on the packet this module receives, it forwards it to the relevant destination module and waits for the arrival of the next packet. Thus, the only real state this module can be in is the *Wait* state, although there can be a few transitory states this module can go to, where it performs the forwarding functions.

4.10.3.4 Event Response Table

The detailed design approach followed in this subsection is very similar to that followed in the wired end device code example (see Subsection 4.3).

Table 4-10: Event Response Table for "fwd" Process

Current State	Logical Event	Condition	Action	Next State
Init	Simulation start		Perform initialization steps, initialize state variables.	Wait
Wait	Packet arrival	Packet arrived from *se* or *wired_mac*	Forward packet to the *mac* module[38]	Wait
		Packet arrived from *mac*	If packet is designated to *se*, send the packet to *se*. Otherwise, forward the packet to the *wired_mac* interface.	Wait

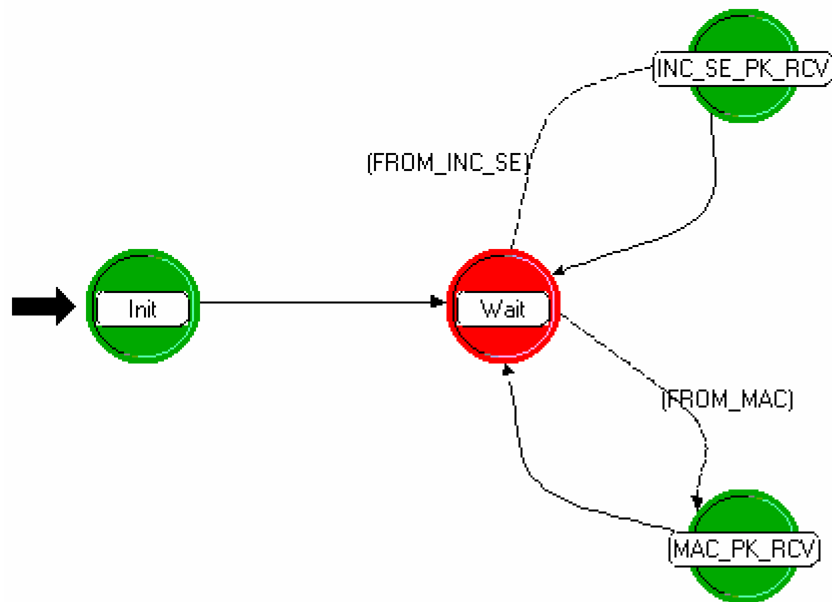


Figure 4-34: fwd Module Process Model

[38] Note that this is an example—in this node, packets from the wired interface are just forwarded to the wireless interface. Equivalently, we could consider forwarding the packets to the "SE" module,or some split in between based on other logic considerations.

4.10.3.5 Implementation Details

Init State Implementation:

In this state, the radio availability is set to enable if it is not a part of any broadcast network, and other state variables are also initialized, including the power, fec-comsec, and the module ids like the *mac* module id and the *se* module id.

MAC_PK_RCV State Implementation:

The execution reaches this state when the fwd module receives a packet from the mac layer.

```
#define FROM_MAC (intrpt_type == OPC_INTRPT_STRM && in_strm == strm_from_mac)
```

In this state, we are receiving the packet from the mac layer. Depending on the destination of the packet, the packet is sent to the transmitter or the se.

```
op_pk_nfd_get (pkt, "packet type", &packet_type);
if (packet_type == PACKET_FROM_INC)
    {
    /* it's a ip dgram, send it to the inc. */
    /* Decapuslate and get the ip_dgram_v4 packet. */
    op_pk_nfd_get (pkt, "data", &out_pkt);
    op_pk_send (out_pkt, strm_to_sincgars_inc_mac);
    op_pk_destroy (pkt);
    }
else if (packet_type == PACKET_FROM_SE)
    {
    op_pk_send (pkt, strm_to_se);
    }
```

INC_SE_PK_RCV State Implementation:

The execution reaches this state if the fwd module receives a packet from either the *se* module or the INC device connected to the radio.

```
#define FROM_INC_SE (intrpt_type == OPC_INTRPT_STRM && \
    (in_strm == strm_from_sincgars_inc_mac || in_strm == strm_from_se))
```

In this state, packets are received from either the INC or the *se* module. If the packet is from the INC, then it is an ip_dgram_v4 format, and needs to be encapsulated and a radio_packet created to forward to the *mac*. If the packet is from the *se* module, then it is already in the *radio_packet* format and can be sent to the *mac*.

Please note that in the following piece of code, the IER parameter structure is reallocated, and some of the information is populated. This is done to ensure a correct IER statistics update. In the case of a radio transmission device, the functions of reliable transmission control protocol (e.g., TCP) are not implemented; therefore, if an IP datagram is lost, it basically means the failure of the IER. Therefore, if, during the radio transmission, a packet is marked as “noise,” this means the associated IER has failed and needs to be reported. The memory associated with IER parameters must be freed at the receiving end or where the IER is marked failed.

```
op_pk_format (pkt, format);

if (strcmp (format,"ip_dgram_v4") == 0)
    {
    /* Receive packet from Inc. */
    /* Encapsulate the packet to radio_packet. */
    op_pk_nfd_access (pkt, "fields", &dgram_ptr);
    data_size = op_pk_total_size_get(pkt);
    out_pkt = op_pk_create_fmt ("radio_packet");

    ierp = nw_oe_ier_create ();

    /* Only data can be coming from the Inc. */
    strcpy (ierp->ier_desc, "DATA");
    /* tos is the priority level of an IER. */
    priority = get_precedence_str (dgram_ptr->tos);
    if (priority)
        strncpy (ierp->ier_priority, priority, 64);


    /* The rest of the IER information is not needed for the radio to proceed, so
    /* just initialize it to 0 or null string. */
    ierp->thread_id = OPC_NIL;
    ierp->thread_start_time = 0;
    ierp->thread_src_platform = 0;
    pid = op_pk_id (pkt);
    /* Set the id to the packet id for debugging purpose. */
    sprintf (temp_name, OPC_PACKET_ID_FMT, pid);
    strncpy (ierp->ier_id, temp_name, 64);
    ierp->ier_class = 0;
    ierp->ier_perish = 0;

    ierp->ier_src_platform = 0;
    /* ier_start_time is set to current time for debugging purpose */
    ierp->ier_start_time = op_sim_time ();
    ierp->ier_equipment = 0;

    /* Encapulate the ip_dgram to the radio_packet in data field. */
    op_pk_nfd_set (out_pkt, "data", pkt);
    op_pk_nfd_set(out_pkt, "IER", ierp,  nw_oe_ier_copy,
              nw_oe_ier_destroy, sizeof (IER_Parameters));
    op_pk_nfd_set (out_pkt, "packet type", PACKET_FROM_INC);
    op_pk_nfd_set (out_pkt, "destination", NwbC_To_All_Destination);
    if(!sincgars_is_routing_pkt (dgram_ptr))
        op_pk_total_size_set(out_pkt, data_size);
    }
else
    {
    /* from SE. */
    out_pkt = pkt;
    }


op_pk_send_delayed (out_pkt, strm_to_mac, fec_comsec);
}
```

4.10.4 mac Module

No specific MAC is detailed here because the medium access control for the broadcast medium is specific to the type of radio being modeled. Typical schemes might be TDMA or Frequency Division Multiple Access (FDMA), for example, to provide access to the shared broadcast medium. The NETWARS model suite has radio models with specific MAC implementations; please refer to SINCGARS and EPLRS as example radios.

The MAC module should essentially guarantee that the packets arriving from the “fwd” module (in the example node above) are sent over the wireless broadcast medium using an access control mechanism.

4.10.5 se Module

4.10.5.1 Module Context and Functionality

The *se* module is responsible for generating traffic based on the information received from *OE*. This module also acts as the traffic destination (or sink). All the traffic destined for a particular device reaches the *se* module, which writes the IER statistics.

The two modules with which se interfaces are oe and the forwarding module (shown in Figure 4-35.)

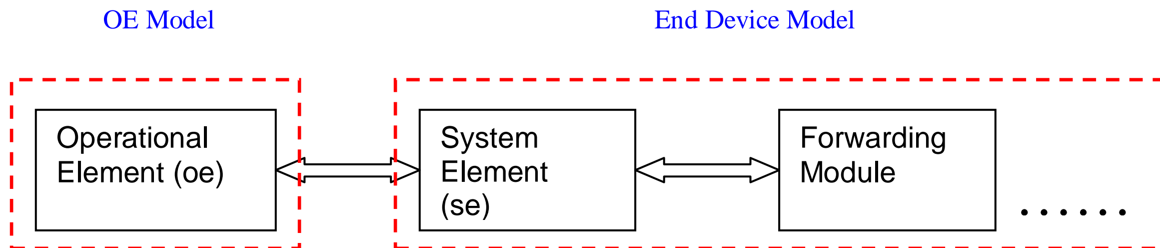


Figure 4-35: SE Module Interfaces

4.10.5.2 Events

Two events can occur at this module:

- Packet arrival from the forwarding module. This packet signifies the reception of the IER for which this device is destined.
- Reception of information from the *OE* to start a new IER.

4.10.5.3 States

The only true state this module can be in is the *Wait* state, in which the module's process model executes after processing either of the above-mentioned events. However, there can be two transitory states where the processes execute to perform the necessary functions based on the events.

4.10.5.4 Event Response Table

Table 4-11: Event Response Table for Radio SE Module

Current State	Logical Event	Condition	Action	Next State
Init	Simulation start	None	Perform initialization.	Wait
Wait	Remote interrupt	None	Generate IER, send IER out to "fwd" module.	Wait
	Stream interrupt	None	Process incoming packet. Inform OE, which records the IER statistics.	Wait

4.10.5.5 Radio SE Process Model

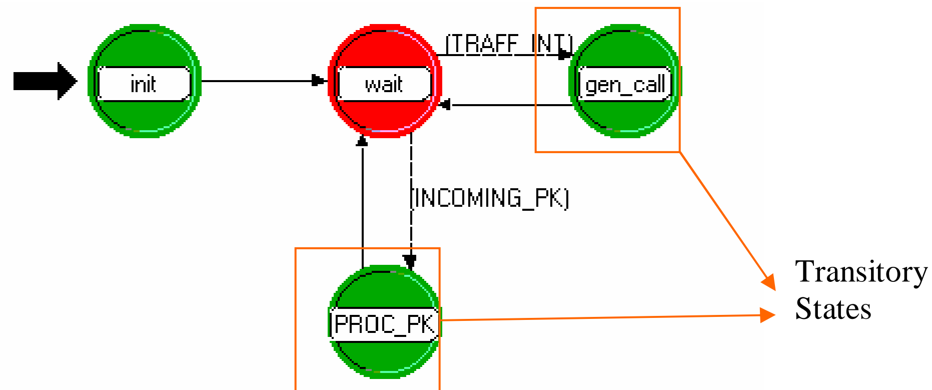


Figure 4-36: Radio SE Process Model

4.10.5.6 Implementation Details

Init State Implementation:

In this state, some of the state variables, including the node id, process id, and oe id for the OPFAC are set.

```
/* Determine object IDs of self, RT and platform */
my_id = op_id_self();
my_nd_id = op_topo_parent(my_id);
my_platform_id = op_topo_parent(my_nd_id);

/* Initialize flags and get the value of the trace attribute */
/* use the globally defined trace flags */
/* in combination with the node specific flag every time a message is to be printed. */
read_local_flags(my_id, my_nd_id, my_platform_id,
                 &mynet, &mynode, pfname, nodename, procname);

/* Initialize to idle */
call_in_progress = 0;

/* get object ID of OE */
oe_nd_id = nw_oe_find(my_platform_id);

if (oe_nd_id != OPC_OBJID_INVALID)
    {
    oe_id = op_id_from_name(oe_nd_id, OPC_OBJTYPE_PROC, "OE");
    }
else
    {
    /* Flag a message to user about missing OE */
    op_sim_message ("Unable to find OE in the OPFAC, IERs from this OPFAC will be igno
    }
```

Gen_Call State Implementation:

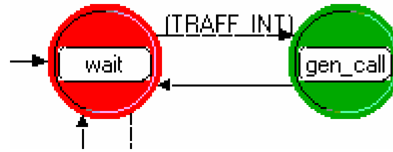


Figure 4-37: Gen_Call State

The control reaches this state if the se receives a remote interrupt from the OE:

```
#define OE_INT          (intrpt == OPC_INTRPT_REMOTE)
```

First retrieve the IER parameters from the ICI associated with the remote interrupt. Make sure that the interrupt code used by the OE is “OE_SE_IER_SEND”. Then determine whether the IER to be generated is a “voice” or a “data” IER.

```
code = op_intrpt_code();

/* remote interrupts are received from the OE  */
/* Get the interrupt code */
/* the code should tell us what to do */
if (code == OE_SE_IER_SEND)
  {
    oe_ici = op_intrpt_ici();
    op_ici_attr_get(oe_ici, "ier_parameters_ptr", &ierp);

    .....
  }
```

If the IER to be generated is of type “voice,” then—

1. Create the packet—set the fields on the packet, such as destination radio ID, flag to indicate that the IER was generated by a radio device, and so forth.
2. Set the radio as “being busy” for the duration of the call. For the radio, “being busy” can be set by marking the radio as “not available.”[39]
3. Send the packet out to the “fwd” module.

[39] The “being busy” flag may be reset after the call is complete to signal to the OE that the radio is available for future IER generation. The reset may be performed, for example, by the “mac” module—after the call is complete. The attribute to be reset for availability is a node-level attribute—”availability_status.”

```
if (!_strcmpi(ierp->ier_desc, "VOICE"))
  {
    if (!call_in_progress)
      {
        /* make a copy of the IER paramaters */
        ier_copy = nw_oe_ier_copy (ierp, sizeof (IER_Parameters));

        /* Generate voice packet */
        pkt = op_pk_create_fmt("radio_packet");
        op_pk_nfd_set (pkt, "packet type", PACKET_FROM_SE);

        /* set destination address information and precedence in packet */
        op_pk_nfd_set(pkt, "IER", ier_copy,
                      nw_oe_ier_copy,
                      nw_oe_ier_destroy,
                      sizeof (IER_Parameters));

        op_pk_nfd_set (pkt, "destination", dest_pf_addr);

        /* Stamp voice packet for statistic reports */
        op_pk_stamp(pkt);

        /* Send voice packet */
        op_pk_send(pkt, MOD_PROC_OUT);

        call_in_progress = 1;

        current_frslno = ierp->fr_serial_number;
        op_ima_obj_attr_set (my_nd_id, "availability_status", 0);

        op_intrpt_schedule_call(op_sim_time() + ierp->ier_size, 0, call_status_reset, 0);
      }
```

If the IER is of type “data,” then—

1. Create the packet—set the fields on the packet, such as destination radio ID, flag to indicate that the IER was generated by a radio device, and so forth.
2. Set the size of the packet in bits to the IER size indicated.
3. Send the packet out to the “fwd” module.

```
else if (!_strcmpi(ierp->ier_desc, "DATA"))
  {
    /* make a copy of the IER paramaters */
    ier_copy = nw_oe_ier_copy (ierp, sizeof (IER_Parameters));

    /* Generate data packet */

    pkt = op_pk_create_fmt("radio_packet");
    op_pk_nfd_set (pkt, "packet type", PACKET_FROM_SE);

    /* set destination address information and precedence in packet */
    op_pk_nfd_set(pkt, "IER", ier_copy, nw_oe_ier_copy,
                  nw_oe_ier_destroy, sizeof (IER_Parameters));

    op_pk_nfd_set (pkt, "destination", dest_pf_addr);


    /* Set size of data packet */
    op_pk_total_size_set(pkt, ierp->ier_size * 8);

    /* Stamp data packet for statistic reports */
    op_pk_stamp(pkt);

    /* Send data packet */
    op_pk_send(pkt, MOD_PROC_OUT);

  }
```

Note here that if the IER is to be "multicast" to more than one destination, then the destination list should be filled when the IER is created.[40] This destination list should be checked when processing the IER at the reception end.

Proc_Pk State Implementation:

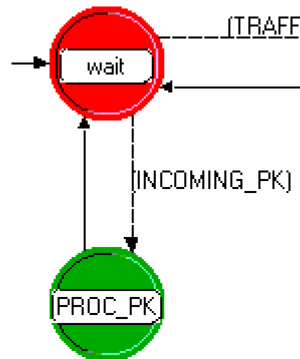


Figure 4-38: Proc_Pk State

The execution reaches this state when the radio end device receives an IER (stream interrupt).

```
#define INCOMING_PK      (intrpt == OPC_INTRPT_STRM)
```

The following factors are to be considered:

1. Determine whether this radio is an intended recipient of the IER.
2. Process the received IER, and send interrupt to the OE in the OPFAC about the received IER. The remote interrupt to the OE should contain the code = NWC_INFORM_SRC_OE_RCVD. The interrupt should also have an ICI associated with it (of format "oe_se") containing information of the IER received.

[40] This function is performed by the OE.

```
/* inform OE of arriving IER */
pkt = op_pk_get (op_intrpt_strm ());

/* get source ier */
op_pk_nfd_get(pkt, "IER", &ierp);

if (strcmp (ierp->ier_id, NWC_App_IER_ID) != 0)
    {
    /* Received an IER packet.  Inform the OE. */
    /* We do not need to do this for application calls. */
    /*  Get the consumer index. */
    cons_idx = nw_consumer_index_get (ierp, pfname);

    if (cons_idx >= 0)
        {
        /* inform the oe */
        oe_ici = op_ici_create("oe_se");

        /*  Make a copy of the IER parameters.  */
        copy_ier_ptr = nw_oe_ier_copy (ierp, sizeof (IER_Parameters));
        op_ici_attr_set(oe_ici, "ier_parameters_ptr", copy_ier_ptr);
        op_ici_attr_set(oe_ici, "consumer_idx", cons_idx);
        op_ici_install(oe_ici);
        op_intrpt_force_remote(NWC_INFORM_SRC_OE_RCVD, ierp->ier_src_oe_id);
        }
    }
else
    {
    /* An application call has completed. */
    if (op_prg_odb_ltrace_active ("cktsw_app") == OPC_TRUE)
        {
        op_prg_odb_print_major ("destination se_sincgars: application call completed.", PRGC_NIL);
        }
    }

nw_oe_ier_destroy (ierp);
/* Discard incoming voice packet */
op_pk_destroy (pkt);
```

4.10.5.7 ICI and Packet Formats

Relevant ICI and packet formats are as follows:

1. “oe_se” ICI is used for interaction between the OE and SE modules.
2. A new packet format for the IER is to be generated by the radio. This packet format has packet format fields for the IER information, a flag to indicate that the packet is from a radio SE, and so forth. For example, the SINCGARS radio creates a packet of format “radio_packet.”

4.10.6 Addressing and Other Issues

For radio devices that have IP devices attached to them (e.g., the wired interface in the radio above may have an IP device such as a router attached to it), autoaddressing modifications are necessary. Please refer to the discussion on autoaddressing changes in Appendix W for further details.

4.10.7 Optimization and Efficiency Considerations

Some high-level efficiency considerations include—

- For the radio model, dynamic receiver groups are an implementation option to modify the list of potential receivers during the course of a simulation.

4.11 SATELLITE TERMINAL GENERIC EXAMPLE

4.11.1 Node Model Contents

A generic satellite terminal, as it is termed in the context of NETWARS, has only a direct mapping of a wired input port of a particular index to an uplink and downlink channel pair of the same index. It does not need to contain any process models that process packets received.

It must have a module to house the antenna aiming process. This plays an important role in pointing a directionalized antenna at the terminal's home satellite, and it plays a role in simulation efficiency. This module should house the process `sat_term_antenna_aim`.

It must have its radio transmitter receiver pair named "sat_tx/rx_0", and it must have its wired input ports named as "uplink_pt/pr_<n>" where the <n> corresponds to the wired input port index and the associated uplink and downlink channel pair index.

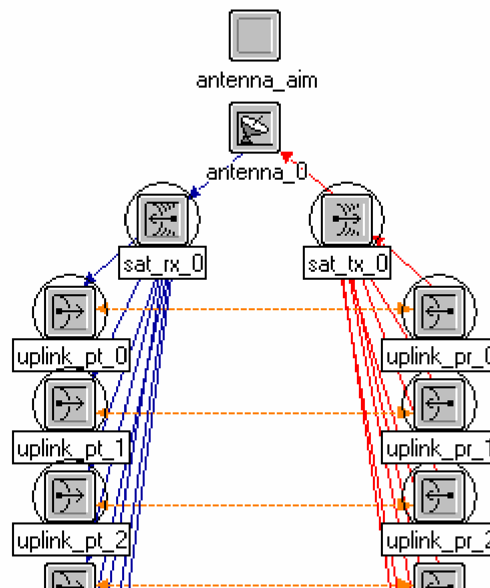


Figure 4-39: Generic Satellite Terminal

4.11.2 Core Self-Description Attributes

- **Nodal Mode** should have the value "Generic."
- **Supported Bands** should have the value "Ku,X,C,Ka."

4.11.3 Additional Attributes

- **Home Satellite (string).** This attribute contains the dotted hierarchical name of the home satellite node in the scenario for this satellite terminal. It should have the initial value "Unspecified," and active attributes should prevent direct user modification.

- **Channel <n> Function (integer).** This helps the Wired Link Deployment Wizard determine what types of links to consider during link deployment. For the *Nodal Mode*

attribute, it should always have the symbol map value "Non-TSSP." The <n> of the attribute name corresponds to a wired port index. A separate instance of this attribute must exist for each wired input port.

- **Port <n> Mapping (compound).** This node model must have N instances of this attribute, where each instance corresponds to a single wired input port. The peer satellite terminal on the other end of the link has values that mirror those on the local device for this attribute.
  - **Input Port (integer).** Corresponds to a wired input port index on this satellite terminal device instance; it should have only one possible value that equals the <n> of the name of its parent compound attribute.
  - **Remote Satellite Terminal (string).** Identifies the peer satellite terminal to which this terminal will connect via the channel index by which it connects.
  - **Remote Input Port (integer).** Corresponds to a wired input port index on the peer satellite terminal. As of version 2006-2, a remote generic terminal can have up to eight wired input ports, so this attribute must support values "0–7".

  Downlink <n> Bandwidth (double, kHz),
  Downlink <n> Data Rate (double, bps),
  Downlink <n> Frequency (double, MHz),
  Uplink <n> Bandwidth (double, kHz),
  Uplink <n> Data Rate (double, bps),
  Uplink <n> Frequency (double, MHz),
  Uplink <n> Power (double, W)

  Together, these attributes define the properties of channel <n>. The node model should include an instance of each of these attributes for every wired input port channel. The user should not have the ability to directly modify them in the Scenario Builder editor. Only the Satellite Link Deployment Wizard should assign these attribute values. Active attribute definitions should prevent the user from modifying them directly.

  Modulation Downlink (string),
  Modulation Uplink (string)

  These attributes define the modulation used for all channels of this satellite terminal in the uplink and downlink directions. The user should not have the ability to directly modify them in the Scenario Builder editor. Instead, only the Satellite Link Deployment Wizard should assign these attributes values. Active attribute definitions should prevent the user from modifying them directly.

4.11.4 Antenna Aim Process

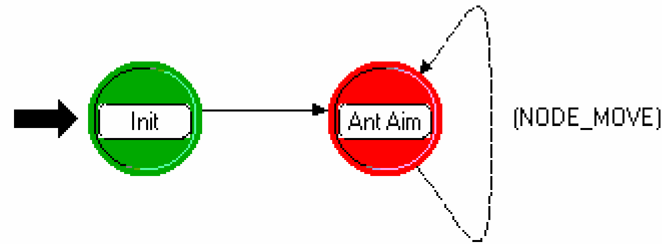
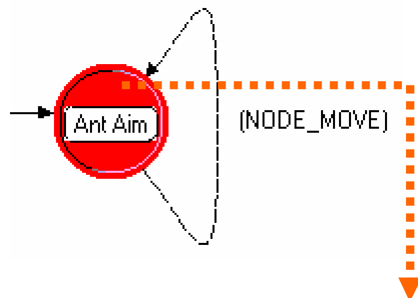


Figure 4-40: Antenna Aim Process

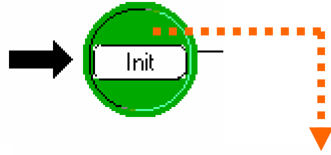
This process serves two purposes. It repoints the satellite terminal's antenna every time the satellite moves. It also sets up simulation efficiency for satellite terminals that do not have any process models besides this one. When running in SATCOM efficiency mode, a simulation-level attribute defined in the *satellite_switch* process model, each satellite has the responsibility of establishing the receiver group of its own channels and those of its home satellite. TSSP satellite terminals, for example, do this in the *tssp* process model, but generic satellite terminals do that in the *sat_term_antenna_aim* process model.

4.11.5 Key Code Snippets from Antenna Aim Process




```
// Ant Move Enter Execs
//
// Point the antenna of the satterm in which this process runs at the location
// coordinates of the home satellite.

op_ima_obj_attr_get (sv_sat_id, "longitude", &sv_lon);
op_ima_obj_attr_get (sv_sat_id, "latitude", &sv_lat);
op_ima_obj_attr_get (sv_sat_id, "altitude", &sv_alt);

op_ima_obj_attr_set (sv_ant_id, "target latitude", sv_lat);
op_ima_obj_attr_set (sv_ant_id, "target longitude", sv_lon);
op_ima_obj_attr_set (sv_ant_id, "target altitude", sv_alt);

if (op_prg_odb_ltrace_active ("sat_term_antenna_aim"))
   {
   op_prg_odb_print_minor ("", OPC_NIL);
   printf ("Antenna aimed at lat=%lf, lon=%lf, alt=%lf\n", sv_lat, sv_lon, sv_alt);
   op_prg_odb_bkpt ("sat_term_antenna_aim");
   }
```

This code executes when the kernel notifies the process of movement on the part of the home satellite device of the satellite terminal via the OPNET kernel procedure `op_ima_obj_pos_notification_register ()`.




```
// uplink channel
//
// Build lists of receiver channels on satellite to include in the rxgroups
// of the satterm transmitter channels.
if (freq == bypass_ul_frequency)
    {
    op_ima_obj_attr_get (row_attr_id, "Transponder", transp_name);
    op_ima_obj_attr_get (row_attr_id, "Channel", &chnl_idx);
    sat_rx_id = op_id_from_name (sv_sat_id, OPC_OBJTYPE_RARX, transp_name);
    if (OPC_OBJTYPE_INVALID != sat_rx_id)
        {
        op_ima_obj_attr_get (sat_rx_id, "channel", &chnl_id);
        chnl_row_id = op_topo_child (chnl_id, OPC_OBJTYPE_RARXCH, chnl_idx);
        rx_group_set [rx_group_set_size] = chnl_row_id;
        rx_group_set_size++;

        if (op_prg_odb_ltrace_active ("sat_term_antenna_aim"))
            {
            op_prg_odb_print_minor ("", OPC_NIL);
            printf ("Added %d (%lf MHz) to satterm rxgroup.\n", (int) chnl_row_id, freq);
            op_prg_odb_bkpt ("sat_term_antenna_aim");
            }
        }
    }

// downlink channel
//
// Add satterm receiver channels to satellite transmitter channel's rxgroup.
if (freq == bypass_dl_frequency)
    {
    op_ima_obj_attr_get (row_attr_id, "Transponder", transp_name);
    op_ima_obj_attr_get (row_attr_id, "Channel", &chnl_idx);
    sat_tx_id = op_id_from_name (sv_sat_id, OPC_OBJTYPE_RATX, transp_name);
    if (OPC_OBJTYPE_INVALID != sat_rx_id)
        {
        op_ima_obj_attr_get (sat_tx_id, "channel", &chnl_id);
        sat_chnl_row_id = op_topo_child (chnl_id, OPC_OBJTYPE_RATXCH, chnl_idx);

        op_ima_obj_attr_get (my_rx_id, "channel", &chnl_id);
        chnl_row_id = op_topo_child (chnl_id, OPC_OBJTYPE_RARXCH, j);

        op_radio_txch_rxch_add (sat_chnl_row_id, chnl_row_id);

        if (op_prg_odb_ltrace_active ("sat_term_antenna_aim"))
            {
            op_prg_odb_print_minor ("", OPC_NIL);
            printf ("Added %d (%lf MHz) to satellite rxgroup\n", (int) chnl_row_id, freq);
            op_prg_odb_bkpt ("sat_term_antenna_aim");
            }
        }
    }
```

This code executes at simulation startup if the simulation runs with the *SATCOM Efficiency Mode* set to “Enabled.” It configures its uplink channels’ rxgroups and its home satellite’s downlink transponders channels’ rxgroups.

4.12 SATELLITE TERMINAL WITH TSSP EXAMPLE

4.12.1 Overview

TSSP serves as a multiplexing scheme used in Super High Frequency (SHF) satellite systems. It performs multiplexing and de-multiplexing at the satellite link endpoints on the terminals. TSSP employs the concept of a nodal terminal versus a non-nodal terminal. A non-nodal terminal simply has one uplink channel for its multiplexed traffic for transmission and a single downlink channel for receiving multiplexed traffic that it decodes and forwards to its wired input ports. A nodal terminal has one uplink signal that it transmits with all of its multiplexed traffic; however, it can support multiple downlink channels where each downlink channel can carry a different multiplexed signal. For more information about TSSP and nodal versus non-nodal, consult Chairman of the Joint Chiefs of Staff Manual (CJCSM) 6231 and Military Standard (MIL-STD)-188-168.

4.12.2 Node Model Contents

A non-nodal TSSP satellite terminal has two wired input ports on the landline side, but the model can accommodate up to eight for those who would like to model it in that manner. Each wired transmitter/receiver pair must have the naming format "input_pt/pr_<n>" where <n> represents the port index. It has exactly one radio interface named "sat_tx/rx_0" that has a single uplink and a single downlink channel. The uplink channel carries the outgoing multiplexed signal, while the downlink channel receives the incoming multiplexed signal. All the interfaces connect to the central processing unit, the module named "tssp." This module performs the multiplexing of the outgoing bitstream and the demultiplexing of the incoming bitstream. Lastly, it has a module named "antenna_aim" that aims the device's directional antenna at the home satellite.

A nodal TSSP satellite has the same properties as its non-nodal counterpart with two exceptions. It must have exactly eight inputs, no more and no less. It also can support up to four incoming bitstreams to demultiplex, which means it has four downlink channels rather than just one.

To develop second- and third-generation TSSP models, simply increase the number of wired input ports to 12, increase the nodal terminal's number of downlink de-multiplexing channels, and make the appropriate data rate values supported on the channel attributes. The subsections below discuss attributes. Their values have a great deal of impact on how the model behaves.

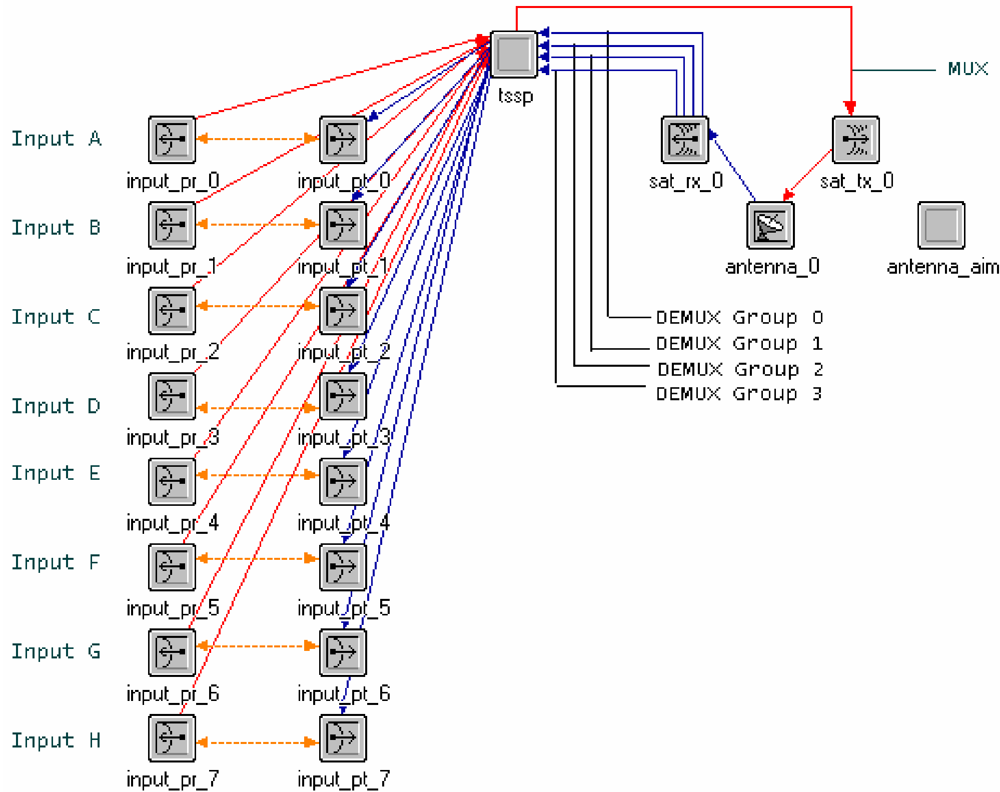


Figure 4-41: TSSP Satellite Terminal

4.12.3 Core Self-Description Attributes

Nodal mode should have the following values under the following conditions:

- "Non-Nodal TSSP" for first-generation non-nodal terminals
- "Nodal TSSP" for first-generation nodal terminals
- "Non-Nodal ETSSP" for second-generation (enhanced) terminals
- "Nodal ETSSP" for second-generation (enhanced) terminals
- "Non-Nodal ETSSP G3" for third-generation non-nodal terminals
- "Nodal ETSSP G3" for third-generation nodal terminals.

Supported bands should have the value "Ku,X,C,Ka".

4.12.4 Additional Attributes

The TSSP module contains several attributes, but how you set the values of some affects which others the model reads during simulation.

- **Nodal Mode.** This attribute plays a pivotal role in how the process reads other attributes. This attribute should have the same value as specified in the *Nodal Mode* Core Self.

- **Description attribute.** The node should always have this attribute promoted, set, and hidden. It should have these values under the following circumstances.

  - “Non-Nodal TSSP” for first-generation non-nodal terminals
  - “Nodal TSSP” for first-generation nodal terminals
  - “Non-Nodal ETSSP” for second-generation (enhanced) terminals
  - “Nodal ETSSP” for second-generation (enhanced) terminals
  - “Non-Nodal ETSSP G3” for third-generation non-nodal terminals
  - “Nodal ETSSP G3” for third-generation nodal terminals.

- **Home Satellite (string).** This attribute contains the dotted hierarchical name of the home satellite node in the scenario for this satellite terminal. It should have the initial value “Unspecified,” and active attributes should prevent direct user modification.

- *Modulation Downlink (string),*
  Modulation Uplink (string)

  These attributes define the modulation used for all channels of this satellite terminal in the uplink and downlink directions. The user should not have the ability to directly modify them in the Scenario Builder editor. Instead, only the Satellite Link Deployment Wizard should assign these attributes values. Active attribute definitions should prevent the user from modifying them directly.

4.12.5 Node Model Specific Configuration

4.12.5.1 General

Each node model that represents a particular generation and nodal or non-nodal implementation requires some attribute characterization. This subsection describes that for each type of terminal.

Each TSSP node model additionally has two compound attributes that must be uniquely configured for each type of TSSP satellite terminal: *Channel Config* and *Groups Memberships*. *Channel Config* has the attributes that characterize a channel, and *Group Memberships* has the attributes that define TSSP group configurations, also referred to as TSSP circuits.

Make these modifications in OPNET Modeler’s or ODK’s Node Model editor. The default attributes’ symbol maps must have the value “Unset.” Scenario Builder’s Satellite Link Deployment Wizard expects to find these attributes set to the symbol map value “Unset” initially. It also expects these attributes to have the correct number of rows. Each row corresponds to the index of an aggregate side radio channel or an input side wired port.

Refer to the Figure 4-42 below for an example of how to configure these attributes of a nodal TSSP satellite terminal.

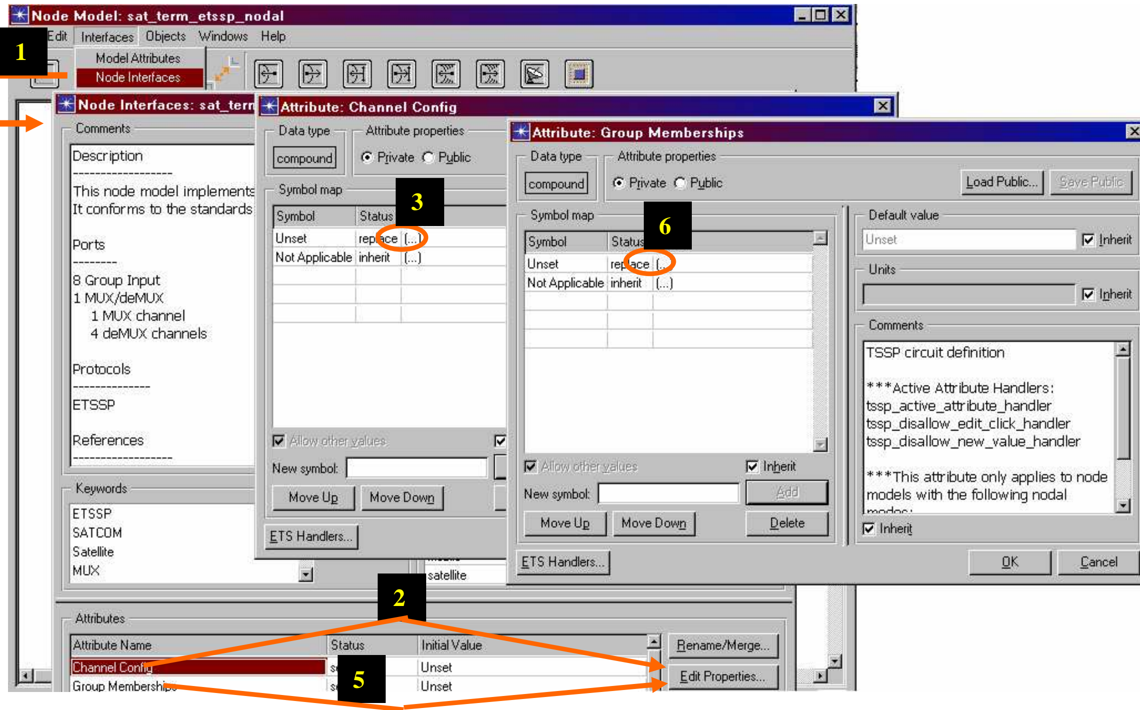
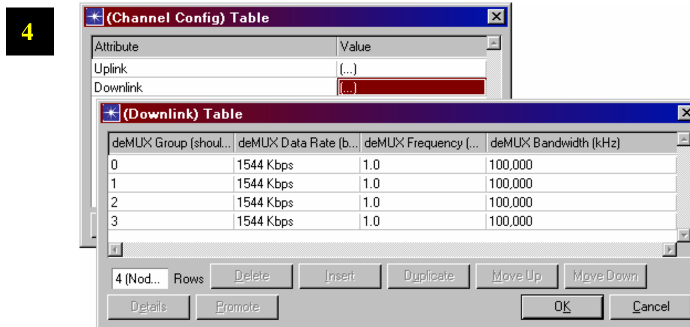


Figure 4-42: Configuration-TSSP Nodal Terminals

Example Configuration: TSSP Nodal Terminals
Channel Config / Downlink (compound)

This should have exactly four rows. Each row corresponds to a deMUX group.



deMUX Group (shoul...	deMUX Data Rate (b...	deMUX Frequency (...	deMUX Bandwidth (kHz)
0	1544 Kbps	1.0	100,000
1	1544 Kbps	1.0	100,000
2	1544 Kbps	1.0	100,000
3	1544 Kbps	1.0	100,000

Figure 4-43: Each Row Corresponding to deMUX Group

Group Configuration (compound)

This should have exactly eight rows. Each row corresponds to an input port group.

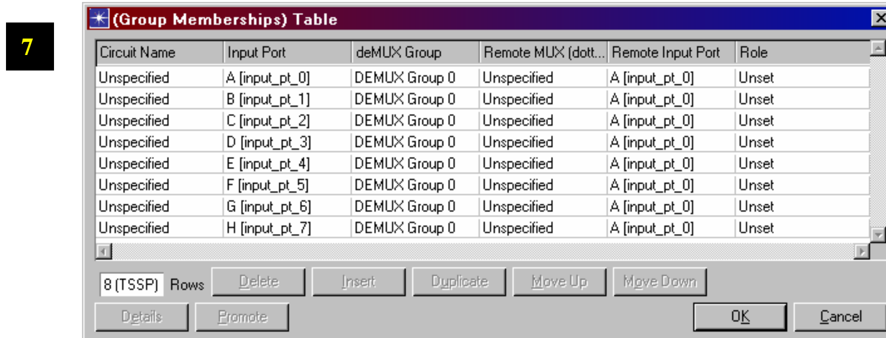


Figure 4-44: Each Row Corresponding to Input Port Group.

Table 4-12: Event Response Table for Radio SE Module

Configuration	Attribute Settings
TSSP Nodal Terminals	*Channel Config | Downlink (compound)* This should have exactly four rows. Each row corresponds to a deMUX group. *Group Configuration (compound)* This should have exactly eight rows. Each row corresponds to an input port group.
TSSP Non-Nodal Terminals (8 Inputs)	*Channel Config | Downlink (compound)* This should have exactly one row. The row corresponds to the single available deMUX group. *Group Configuration (compound)* This should have exactly eight rows. Each row corresponds to an input port group.
ETSSP Nodal Terminals	*Channel Config | Downlink (compound)* This should have exactly six rows. Each row corresponds to a deMUX group. *Group Configuration (compound)* This should have exactly twelve rows. Each row corresponds to an input port group.
ETSSP Non-Nodal Terminals w/ 8 Inputs	*Channel Config | Downlink (compound)* This should have exactly one row. The row corresponds to the single available deMUX group. *Group Configuration (compound)* This should have exactly 12 rows. Each row corresponds to an input port group.

Configuration	Attribute Settings
ETSSP 3G Nodal Terminals	*Channel Config | Downlink (compound)* This should have exactly six rows. Each row corresponds to a deMUX group. *Group Configuration (compound)* This should have exactly 12 rows. Each row corresponds to an input port group.
ETSSP 3G Non-Nodal Terminals w/ 8 Inputs	*Channel Config | Downlink (compound)* This should have exactly one row. The row corresponds to the single available deMUX group. *Group Configuration (compound)* This should have exactly 12 rows. Each row corresponds to an input port group.
All Non-Nodal Terminals w/ 2 Inputs	*Channel Config | Downlink (compound)* This should have exactly one row. The row corresponds to the single available deMUX group. *Group Configuration (compound)* This should have exactly two rows. Each row corresponds to an input port group.

4.12.6 TSSP Process

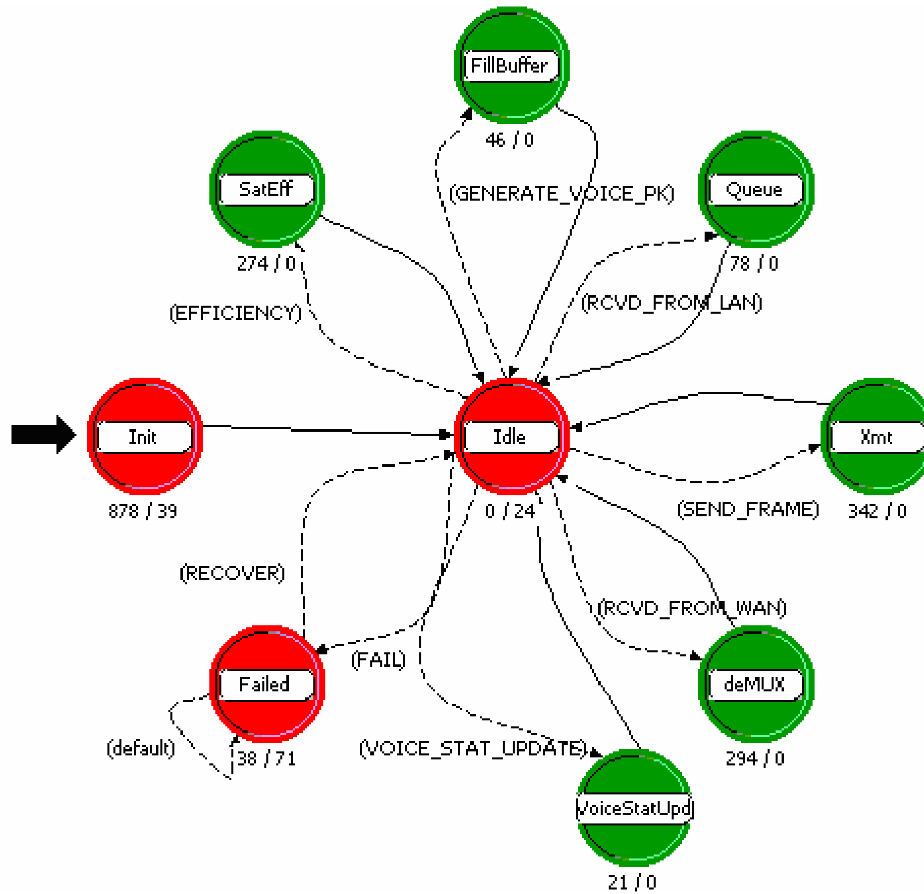


Figure 4-45: TSSP Process Model

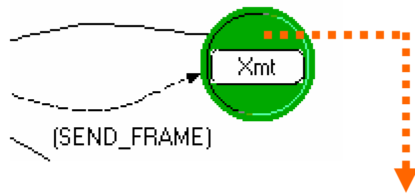
Table 4-13: Events of TSSP Process Model

Current State	Event	Condition	Action	Next State
Init	Simulation start	None	Perform initialization	Idle
Idle	Self Interrupt	Interrupt code = TsspC_Intrpt_SatEf f	Set rxgroups of terminal and satellite channels	SatEff
SatEff	—	None	None	Idle
Idle	Stream Interrupt	Interrupt stream from an input port	Place incoming packet in correct transmission queue	Queue
Queue	—	None	None	Idle
Idle	Self Interrupt	Interrupt code = TsspC_Intrpt_Send Frame	Construct frame with payload of transmission queues and send	Xmt
Xmt	—	None	None	Idle

Current State	Event	Condition	Action	Next State
Idle	Stream Interrupt	Interrupt stream from an radio (satellite) port	Deconstruct the incoming frame, extract payload, and forward it to appropriate inputs	deMUX
deMUX	—	None	None	Idle
Idle	Fail Interrupt	None	Flush queues, cancel all scheduled frame transmissions	Failed
Failed	Recover Interrupt	None	Schedule next frame transmission	Idle
Failed	Stream Interrupt	None	Destroy incoming packet	Failed

4.12.7 Key Code Snippets from TSSP Process

Xmt Enter Execs:

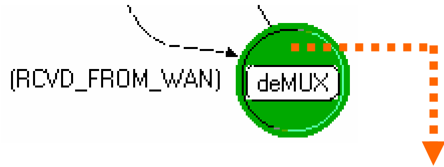



```
// Construct a TSSP frame and transmit it in efficiency mode.
else if (have_data_to_send && sv_efficiency_mode_enabled)
   {
   // If I have efficiency mode turned on, then I won't build the TSSP frame
   // as detailed.  This will use far fewer packets (OPNET packets) and save
   // me a lot of processing time.  In thise mode, I'll just place all the bits
   // of an entire TSSP major frame for one input in a single packet field.
   // Input A's data will go into packet field X.  Input B's data will go into
   // packet field X+1.  Input C's data will go into field X+2, and so forth.

   major_frame_pkptr = op_pk_create_fmt ("tssp_frame");

   // Iterate through each of the inputs and insert data into the TSSP frame
   // that I'll send.
   for (i = 0; i < sv_num_inputs; i++)
       {
       if (!sv_input_port [i].active ||
           0 >= op_sar_buf_size (sv_input_port [i].segbuf_hndl))
           {
           continue;
           }

       have_data_to_send = OPC_TRUE;

       // Calculate the number of bits to put into a single minor frame for
       // this Input.
       seg_size =
           sv_input_port [i].num_bits_subf_1 +           // # bits subframe 1
           (sv_input_port [i].num_bits_subf_2_to_5 * 4); // # bits subframes 2-5

       // We have 60 minor frames in a single TSSP frame, so multiply the
       // number of bits for one frame by that.
       seg_size = seg_size * 60;

       // Get a segement from this Input's SAR buffer.
       seg_pkptr = op_sar_srcbuf_seg_remove (sv_input_port [i].segbuf_hndl,
           seg_size);

       // Insert that segment into the TSSP frame.  Also, I'll offset the
       // packet field index by 10 to avoid conflicting with the formatted
       // fields.
       pkt_fd_idx = i + 10;
       fd_size = seg_size;
       op_pk_fd_set_pkt (major_frame_pkptr, pkt_fd_idx, seg_pkptr, fd_size);
       }
```

This code snippet shows how the TSSP process constructs its frames in efficiency mode. It places data from each input port into a slot index reserved for only one input port. In efficiency mode, each frame slot holds all the data of a TSSP frame with respect to one input port. In

regular mode, the TSSP frame has many slots of smaller size spread out across the entire frame for each input.

deMUX Enter Execs:




```
// Extract the contents of the recieved TSSP frame in efficiency mode
else if (!ignore_this_frame && sv_efficiency_mode_enabled)
    {
    for (i = 0; i < sv_num_inputs; i++)
        {
        if (sv_input_port [i].active &&
            sv_input_port [i].demux_group == demux_idx)
            {
            // Determine the packet field index of this remote MUX's data
            // for this remote Input.  Note that in efficiency mode, it
            // has an offset of 10 to avoid conflict with the standard mode
            // packet fields.
            pkt_fd_idx = sv_input_port [i].neighbor_port + 10;

            // Only continue with this iteration of the loop if the part
            // of the TSSP frame for this input has some data in it to
            // send to this input.
            if (!op_pk_fd_is_set (major_frame_pkptr, pkt_fd_idx))
                {
                continue;
                }

            op_pk_fd_get_pkt (major_frame_pkptr, pkt_fd_idx, &seg_pkptr);

            // Print some trace information.
            if (op_prg_odb_ltrace_active ("tssp"))
                {
                op_prg_odb_print_minor ("", NULL);
                printf ("[deMUX] extracting packet %d of slot %d (efficiency mode)\n",
                    (int) op_pk_id (seg_pkptr), pkt_fd_idx);
                }

            op_sar_rsmbuf_seg_insert (sv_input_port [i].rsmbuf_hndl, seg_pkptr);
            seg_pkptr = NULL;

            // Check the reassembly buffer for fully reassembled packets.
            while (NULL != (grp_input_pkptr = op_sar_rsmbuf_pk_remove (sv_input_port [i].rsmbuf_hndl)))
                {
                // Print some trace information.
                if (op_prg_odb_ltrace_active ("tssp"))
                    {
                    op_prg_odb_print_minor ("", NULL);
                    printf ("[deMUX] sending packet %d to input %d (efficiency mode)\n",
                        (int) op_pk_id (grp_input_pkptr), i);
                    op_prg_odb_bkpt ("tssp");
                    op_prg_odb_bkpt ("tssp_deMUX");
                    }

                // Check the packet format and deal with circuit switch packets if the device is a multiplexer
                op_pk_format(grp_input_pkptr, pk_format);

                if (!strcmp (pk_format, CIRCUIT_SWITCH_PACKET) || !strcmp (pk_format, ISDN_CKSW_PACKET))
                    {
                    // Handle voice packets here before sending out
                    tssp_handle_voice_packets (grp_input_pkptr, i);
                    }

                if (!strcmp (pk_format, "dummy_voice_pk"))
                    {
                    op_pk_destroy (grp_input_pkptr);
                    }
                else
                    {
                    // Send the packet
                    op_pk_send (grp_input_pkptr, sv_input_port [i].strm_to);
                    }
                }
            }
        }
    }
```

This code snippet shows how the TSSP process deconstructs a TSSP frame when running with the global simulation attribute *TSSP Efficiency Mode* set to “Enabled.” Notice how particular parts of the frame apply to different individual landline input ports, also called group members.

4.13 SATELLITE GENERIC EXAMPLE

4.13.1 Overview

This subsection provides an example of how to create a satellite that can support the deployment of bent-pipe links running through it. Creating a satellite node in NETWARS requires following some basic conventions. Before reading this subsection, be sure to read the subsection "Building Wireless Interfaces" in Section 3, Building NETWARS Models.

The following subsection details what a satellite model must have implemented if it is to function with Scenario Builder's functionality, such as its Link Deployment Wizard, and is to interoperate with other device models of the NETWARS Standard Model Library.

4.13.2 Node Model Contents

A satellite device model must have one or more uplink and downlink transponders, each with some number of channels. Each transponder must connect to its own antenna module. Uplink transponders (radio receiver modules) should follow the naming convention `uplink_transponder_rx_<n>` where <n> is an integer that identifies each uplink transponder with a unique index. Similarly, the downlink transponders should follow the naming convention downlink_transponder_tx_<n>. Each transponder's antenna should follow the naming convention antenna_tx/rx_<n>.

The satellite model must have its *equipment_type* attribute set to "Satellite." It can discover the possible ground terminals by checking for devices with an *equipment_type* set to "Satellite terminal."

4.13.3 Additional Attributes

At its most fundamental level, b a satellite model must have some basic attributes that define that model as a satellite node in NETWARS. These attributes further characterize how the satellite device handles the traffic that passes through it.

- **Channel Config (compound).** This compound attribute defines the properties of each channel on the satellite device. Each row of the compound attribute applies to one channel.
  - **Transponder (string)**. Identifies the transponder on which this channel resides; it should have a locked value via active attributes that the user cannot modify in Scenario Builder.
  - **Channel (integer).** Identifies the index of this channel on the transponder; it should have a locked value via active attributes that the user cannot modify in Scenario Builder. Together, the *Transponder* and *Channel* attributes provide a unique way to identify any channel of the satellite.
  - **Frequency (double).** Minimum frequency value assigned to this channel (MHz).
  - **Bandwidth (double).** Bandwidth value assigned to this channel (kHz).
  - **Data Rate (double).** Data rate value assigned to this channel (bps).
  - **Power (double).** Transmission power assigned to this channel (W); only applicable to downlink channels.

- **Switching Table (compound).** This compound attribute defines how the device forwards traffic received on uplink channels to downlink channels. Each row represents a mapping of an uplink channel to a downlink channel.
  - **Uplink Transponder (index with symbol map).** Identifies the transponder of the uplink channel to map to some other downlink transponder; it should have a locked value via active attributes that the user cannot modify in Scenario Builder.
  - **Uplink Chnl Idx (integer).** Identifies the channel index of the uplink channel to map to some other downlink transponder; it should have a locked value via active attributes that the user cannot modify in Scenario Builder.
  - **-- maps to --> (string).** Serves no purpose beyond visualization.
  - **Downlink Transponder (index with symbol map).** Identifies the transponder of the downlink channel to which the satellite forwards all traffic from the uplink channel identified by *Uplink Transponder* and *Uplink Chnl Idx.*
  - **Downlink Chnl Idx (integer).** Identifies the channel index of the downlink channel to which the satellite forwards all traffic from the uplink channel identified by *Uplink Transponder* and *Uplink Chnl Idx.*
- **Current Number of Links (integer).** This integer value represents the current number of links deployed through this satellite. This attribute should always have a value of "0" upon instantiation of this model and an active attribute handler to prevent its direct modification by a user.

  Only Scenario Builder should update this value upon the creation and removal of satellite links running through the satellite.

- *Uplink Modulation (compound),*
  Downlink Modulation (compound)

  The satellite process reads these attributes to determine what modulation to use for each of the uplink and downlink transponders. The satellite switch module maintains these two attributes as extended attributes defined on the module itself.

  Both of these compound attributes have a *Transponder Index (integer)* and a *Modulation Scheme (string)* subattributes. The number of rows in the *Uplink Modulation* and *Downlink Modulation* compound attributes should equal the number of uplink and downlink transponders, respectively.

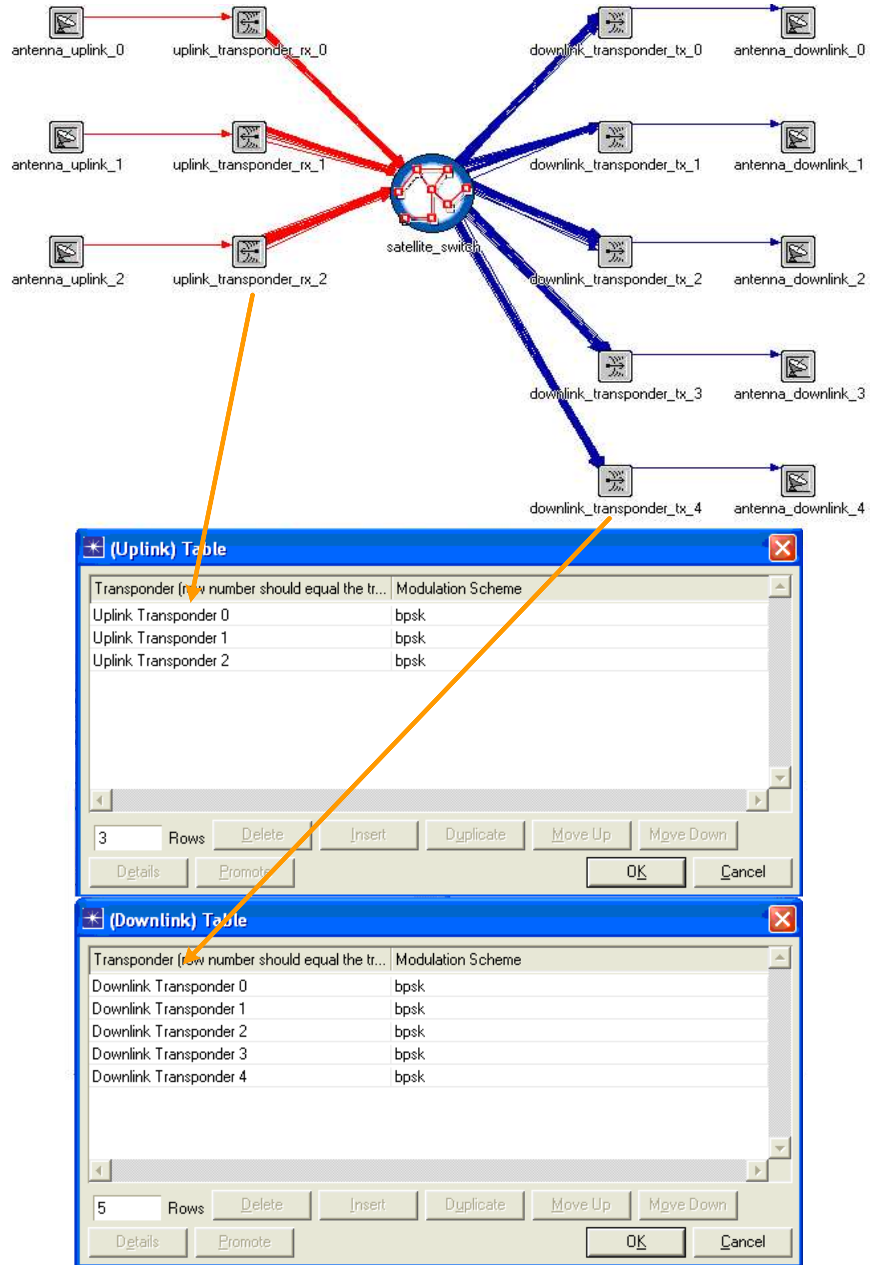


Figure 4-46: Uplink and Downlink Tables

4.13.4 Satellite Switch Process

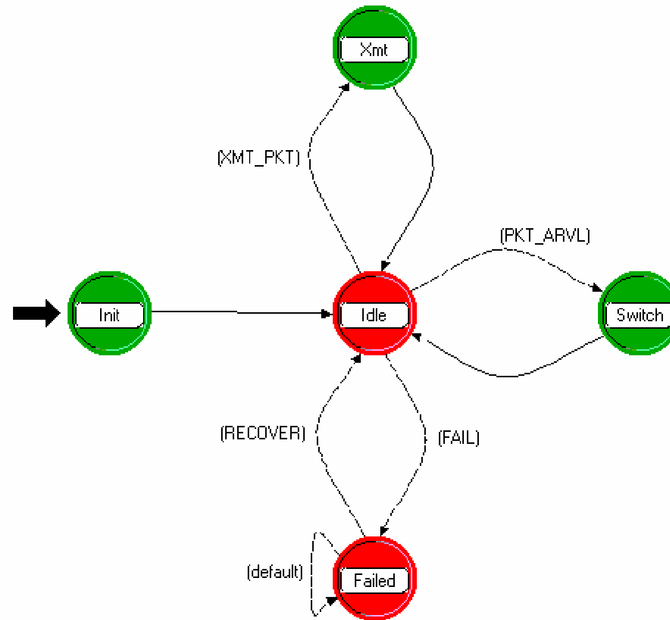


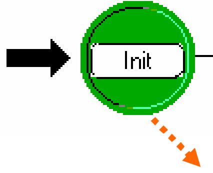
Figure 4-47: Satellite Switch Process Model

Table 4-14: Events of Satellite Switch Process Model

Current State	Logical Event	Condition	Action	Next State
Init	Simulation start	None	Perform initialization.	Idle
Idle	Stream Interrupt	None	None	Switch
Idle	Failure Interrupt	None	Flush transmission and receiver queues	Failed
Idle	Self Interrupt	None	None	Xmt
Switch	N/A	None	Transmit immediately **or** En queue the packet in the service queue, which depends on the packet switching rate having INFINITE for its value	Idle
Xmt	Self Interrupt	None	Transmit next packet in transmit queue	Idle
Failed	Recover Interrupt	None	None	Idle
Failed	Stream Interrupt	None	Destroy incoming packet	Failed
Failed	Fail Interrupt	None	None	Failed

Key Code Snippets from Satellite Switch Process

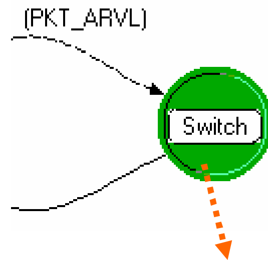
Init Enter Execs:




```
// Read the switching table from the local node's attributes
op_ima_obj_attr_get (sv_module_id, "Switching Table", &comp_attr_id);
n = op_topo_child_count (comp_attr_id, OPC_OBJTYPE_GENERIC);
for (i = 0; i < n; i++)
    {
    // Get the next row of the switching table attribute.
    row_attr_id = op_topo_child (comp_attr_id, OPC_OBJTYPE_GENERIC, i);

    // The incoming stream index from the radio receiver object should equal
    // the uplink transponder index + the channel index.
    op_ima_obj_attr_get (row_attr_id, "Uplink Transponder", &sw_tbl_entry.ul_transp);
    op_ima_obj_attr_get (row_attr_id, "Uplink Chnl Idx", &sw_tbl_entry.ul_chnl);

    // The outgoing stream index to the radio transmitter object should equal
    // the downlinkg transponder index + the channel index.
    op_ima_obj_attr_get (row_attr_id, "Downlink Transponder", &sw_tbl_entry.dl_transp);
    op_ima_obj_attr_get (row_attr_id, "Downlink Chnl Idx", &sw_tbl_entry.dl_chnl);

    // Get the uplink incoming stream.
    strcpy (transp_name, "uplink_transponder_rx_");
    _itoa (sw_tbl_entry.ul_transp, idx_str, 10);
    strcat (transp_name, idx_str);
    ra_mod_id = op_id_from_name (sv_node_id, OPC_OBJTYPE_RARX, transp_name);
    strm_id = op_topo_assoc (ra_mod_id, OPC_TOPO_ASSOC_OUT, OPC_OBJTYPE_STRM, sw_tbl_entry.ul_chnl);
    op_ima_obj_attr_get (strm_id, "dest stream", &sw_tbl_entry.ul_strm);

    // Get the uplink channle Objid.
    op_ima_obj_attr_get (ra_mod_id, "channel", &chnl_id);
    chnl_row_id = op_topo_child (chnl_id, OPC_OBJMTYPE_ALL, sw_tbl_entry.ul_chnl);
    op_ima_obj_attr_get (chnl_row_id, "min frequency", &sw_tbl_entry.ul_freq);

    // Get the downlink outgoing stream.
    strcpy (transp_name, "downlink_transponder_tx_");
    _itoa (sw_tbl_entry.dl_transp, idx_str, 10);
    strcat (transp_name, idx_str);
    ra_mod_id = op_id_from_name (sv_node_id, OPC_OBJTYPE_RATX, transp_name);
    strm_id = op_topo_assoc (ra_mod_id, OPC_TOPO_ASSOC_IN, OPC_OBJTYPE_STRM, sw_tbl_entry.dl_chnl);
    op_ima_obj_attr_get (strm_id, "src stream", &sw_tbl_entry.dl_strm);

    // Get the uplink channle Objid.
    op_ima_obj_attr_get (ra_mod_id, "channel", &chnl_id);
    chnl_row_id = op_topo_child (chnl_id, OPC_OBJMTYPE_ALL, sw_tbl_entry.dl_chnl);
    op_ima_obj_attr_get (chnl_row_id, "min frequency", &sw_tbl_entry.dl_freq);

    // The index of this switching table entry in the switching table, an array,
    // will depend on two things: (1) the uplink transponder stream index and
    // (2) the channel index.
    sw_tbl_idx = (sw_tbl_entry.ul_transp * NUM_CHNLS_PER_TRANSP) + sw_tbl_entry.ul_chnl;
    if (UNSET == sv_switching_table [sw_tbl_idx].ul_transp)
        {
        // Add this entry to the switching table.
        sv_switching_table [sw_tbl_idx] = sw_tbl_entry;
```

This code from the **Init** state reads the *Switching Table* attribute to determine how an uplink channel maps to a downlink channel. A two-dimensional array defines the switching table in such a manner that any packet received on any single uplink frequency has a predetermined downlink frequency on which the satellite transmits it.

Switch Enter Execs:




```
pkptr = op_pk_get (intrpt_strm = op_intrpt_strm ());
op_pk_stamp (pkptr);

// Get the switching table index of this uplink channel based on the
// incoming stream.
sw_tbl_idx = sv_strm_map [intrpt_strm].sw_tbl_idx;

if (0.0 == sv_processing_delay_per_frame)
    {
    // I have infinite packet switching rate, so just send the packet.

    pkt_size = op_pk_total_size_get (pkptr);

    // Write uplink channel stats.
    stat_idx = sv_switching_table [sw_tbl_idx].ul_strm;
    op_stat_write (sv_ul_bps_stathndl [stat_idx], pkt_size);
    op_stat_write (sv_ul_bps_stathndl [stat_idx], 0.0);

    // Write downlink channel stats.
    stat_idx = sv_switching_table [sw_tbl_idx].dl_strm;
    op_stat_write (sv_dl_bps_stathndl [stat_idx], pkt_size);
    op_stat_write (sv_dl_bps_stathndl [stat_idx], 0.0);

    // Print some trace information.
    if (op_prg_odb_ltrace_active ("satcom"))
        {
        op_prg_odb_print_minor ("", OPC_NIL);
        printf ("[Switch] ul transp %d ch %d (%.1lfMHz) --> packet %d --> dl trasnp %d ch %d (%.1lfMHz)\n",
            sv_switching_table [sw_tbl_idx].ul_transp,
            sv_switching_table [sw_tbl_idx].ul_chnl,
            sv_switching_table [sw_tbl_idx].ul_freq,
            (int) op_pk_id (pkptr),
            sv_switching_table [sw_tbl_idx].dl_transp,
            sv_switching_table [sw_tbl_idx].dl_chnl,
            sv_switching_table [sw_tbl_idx].dl_freq);
        op_prg_odb_bkpt ("satcom");
        }

    op_pk_send (pkptr, sv_strm_map [intrpt_strm].dl_strm);
    }
```

In the packet arrival state, the process reads the switching table to determine to which downlink stream to forward the received uplink packet. This snippet shows the process set to an infinite switching speed, whereby it sends the packet immediately upon receiving it rather than storing it in a queue and sending it at a specified rate. The infinite switching speed setting defines a more realistic scenario because bent pipe links typically have circuits running through them, which means it never needs to store and forward bits; it just sends them without waiting to detect the trailing edge of a packet. Also, note how the interrupt stream value and the first dimension indexes of the switching table correspond.

4.14 LINK MODEL EXAMPLE

4.14.1 Overview

This subsection explains the construction of a link model using an example. The example link considered is a duplex link with two channels, each at 1 Mbps. The link also has an additional signaling overhead. The delay due to the signaling overhead is specified as a model attribute.

4.14.2 Steps

Step 1: Because this is a duplex link, in the Link Types field, set *ptdup* as the supported link type. In a new link editor window, set the link type option *ptdup* as “yes” and leave the other options as “no.”

Step 2: In the Attributes field, specify *channel count* as 2. There are two channels supporting data rates of 1 Mbps each. Therefore, set the *data rate* as 2,000,000.

Step 3: On the Link menu, choose Model Attributes. In the New Attribute field, enter “signaling overhead” and click Add. The *type* for this attribute is specified as “double.”

Step 4: Save the link model.

4.14.3 Pipeline Stage: txdel

The newly created link has a model attribute called *signaling overhead*. The signaling overhead for a packet causes a delay in the packet transmission. To account for this, the transmission delay pipeline stage must be customized.

Sample code for this customization is provided below (this code is derived from `dpt_txdel.ps.c`):

```
/** Compute transmission delay associated with a    **/
/** packet transmission on a point-to-point link.   **/
FIN_MT (dpt_txdel (pkptr));

/* Obtain object id of transmitter channel forwarding transmission. */
tx_ch_obid = op_td_get_int (pkptr, OPC_TDA_PT_TX_CH_OBJID);

/* Obtain the transmission rate of that channel. */
if (op_ima_obj_attr_get (tx_ch_obid, "data rate", &tx_drate) == OPC_COMPCODE_FAILURE)
    op_sim_end ("Error in point-to-point transmission delay pipeline stage (dpt_txdel):",
        "Unable to get transmission rate from channel attribute.", OPC_NIL, OPC_NIL);

/* Obtain length of packet. */
pklen = op_pk_total_size_get (pkptr);

/* Compute time required to complete transmission of packet. */
tx_delay = pklen / tx_drate;

/* Place transmission delay in packet transmission data attribute. */
op_td_set_dbl (pkptr, OPC_TDA_PT_TX_DELAY, tx_delay);

FOUT
}
```

Figure 4-48: Code 3-Adding Signaling Overhead to Transmission Delay

Please refer to the pipeline stage dpt_txdel.ps.c in the OPNET\<rel_dir>\models\ std\links folder for more information. FIN/FOUT/FRET (FIN and FOUT are used in the sample code above) are macros representing Function-IN, Function-OUT, and Function-RETurn. OPNET recommends that developers incorporate these macros in their code. This is useful while generating stack traces and function profiling. Further information on this can be found in the OPNET Online Documentation → Programmers Reference → Discrete Event Simulation → Introduction → Kernel Procedure Names.

4.15 OE NODE EXAMPLE

4.15.1 Overview

This subsection explains how to build an OE model using an example. The OE node does not have to be connected to other devices via links. Thus, it does not need to have any physical interfaces. It contains one module that performs all the necessary functions. The SDF file is parsed and the IER, movement, and failure/recovery information pertaining to this OPFAC are obtained in the *init* state. Depending on what the next event is, the process model transitions to the appropriate state—movement or IER transmission.

Important: a justified change to the OE Node is rare and should be examined closely, due to its effects on the entire OPFAC. Be advised, before modifying the OE Node it is highly suggested that you contact the NETWARS Office to determine if there is a better method of accomplishing your objective. After talking with the NETWARS Office, if it is still appropriate to modify the OE Node, then be careful to make backups and have a plan in place to rollback any changes you are about to make.

The OE node model requires a single module in the node.

4.15.2 Steps

Step 1.

- In the node editor, click the “create processor” button.
- Click the workspace to place the processor module.

Step 2.

- Left-click the module and right-click to edit the attributes.
- Choose the “process module” attribute and change it to the process model name as created below.
- Choose the “name” attribute and name the module as “oe.”

4.15.3 Process Model

Step 1. The functions of the OE are defined using a data flow diagram.

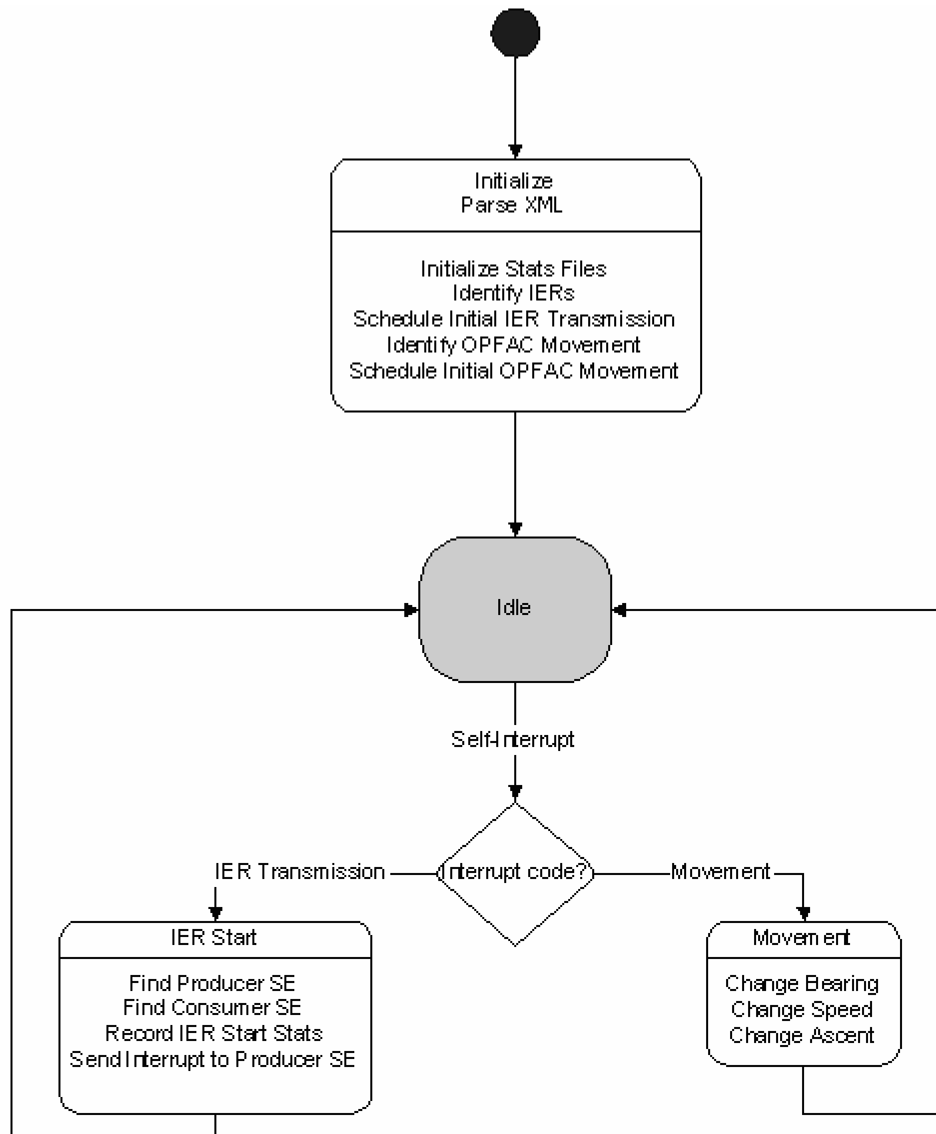


Figure 4-49: Functions of OE Process Model

Step 2. The process model must be built.

The OE process model consists of the states shown in Figure 4-50. The *init* state parses the SDF file to obtain the IER and movement information. Then the process model transitions to the *idle* state. If the OE is required to send an IER, it transitions to the *Device Find* state. If the device is found, it generates the IER in the *Tx IER* state.

If the OE is required to move, it transitions to the *movement* state and sets the new *bearing* and *ground speed* values for the OPFAC.

The process model for the OE looks like Figure 4-50.

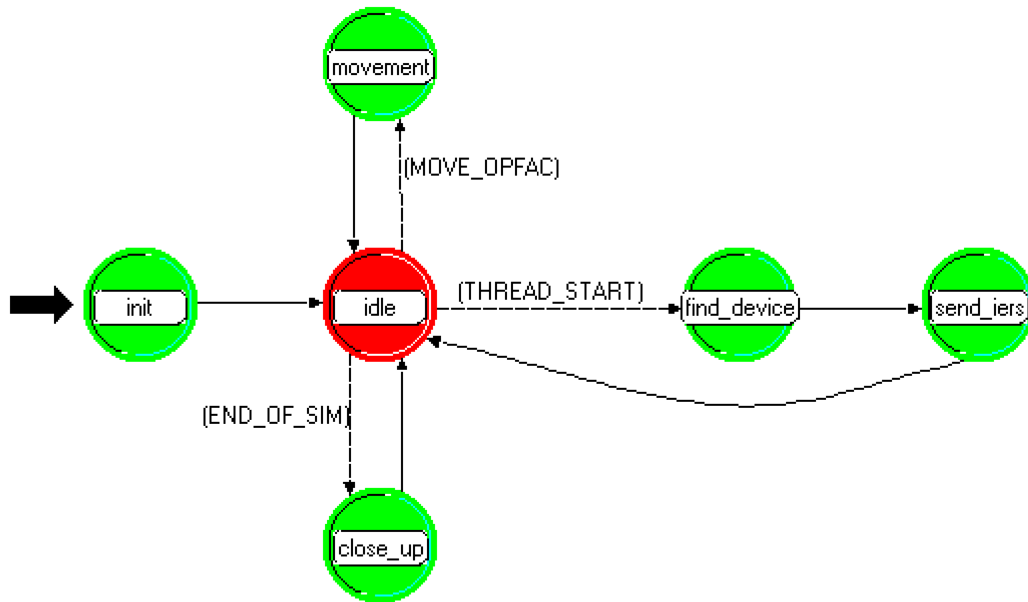


Figure 4-50: OE Process Model

4.16 UTILITY NODE EXAMPLE

4.16.1 Overview

This subsection explains the construction of a Utility Node using an example. The example utility model is the Promina Configuration Utility Node, which is used to define circuits between Promina nodes in a network. Only a high-level overview is given below. For further details, consult the NETWARS Standard model called *pro_portmap_utility.nd.m* and its process model *pro_portmap_process.pr.m*.

4.16.2 Details

Because Utility Nodes are highly specific, begin with a new node model. Because the object will be a repository of information, a single processor module is all that is needed. This processor requires a custom process model that performs the following functions:

- Read in attribute values
- Parse information
- Publish information.

Once the node model is created and a processor module added, the node model looks like Figure 4-51 below.

Figure 4-51: Promina Configuration Utility Node-Node Model

The model attributes for the node model must contain the attribute below. The other detailed configuration attributes can be part of the processor itself.

Table 4-15: Utility Node-Model Attributes

Attribute Name	Attribute Type
utility_technologies	String

4.16.3 Process Model

The Utility Node reads in attributes, parses them, and then publishes them, making the information available to other models. The Promina Configuration utility does all of this using a single *BEGSIM* interrupt.

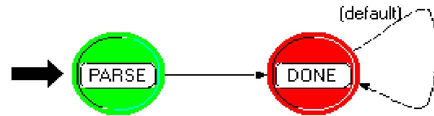


Figure 4-52: Promina Configuration Object-Process Model

The “DONE” state is used to ensure errors are not incurred if this device sees an event. The code in the Enter Executives of the “PARSE” state perform all of the actions of this object, as seen in the following code sample:

```
/* Obtain commonly used identification information and store them into the         */
/* corresponding state variables.                                                   */
my_objid        = op_id_self();
op_ima_obj_attr_get(my_objid, "process model", proc_model_name);
own_node_objid = op_topo_parent(my_objid);
own_subnet_objid = op_topo_parent(own_node_objid);

/* Procedure to get to the top subnet. */
top_subnet_objid = nw_sup_top_id_get(own_subnet_objid);

/* Parse the process model attributes. */
pro_portmap_parse();

/* Print out some basic trace statements. */
if (op_prg_odb_ltrace_active ("pro_portmap") || TRACE_PROMINA)
    {
    sprintf (msg0, "Total number of portmap entries    : (%d)", op_prg_list_size (portm
    sprintf (msg1, "Total number of selected path entries    : (%d)", op_prg_list_size

    op_prg_odb_print_major ("Promina Circuit Provision has finished parsing port map us
    }

/* Publish the result through process registry so the Promina processes can access them
pro_portmap_publish();
```

Figure 4-53: Promina Configuration Object-Sample Code

4.17 CONVERTING A DEVICE MODEL FROM THE OPNET STANDARD MODEL LIBRARY

4.17.1 Overview

The OPNET Standard Model Library contains the node model *wlan_server_adv*. The following example demonstrates how to make this model function in OPNET COTS products such that it is compliant with this guide.

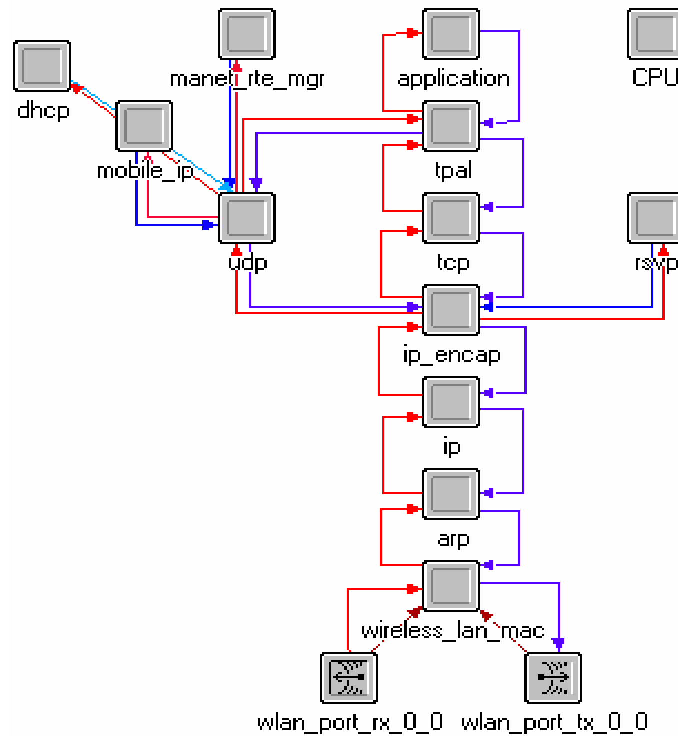


Figure 4-54: Sample Node Model

4.17.2 Details

Step 1. Determine which subsections of Section 3 apply to this device model.

This has the application layer, so it has the characteristics of an end system. It has a radio transmitter and receiver pair, so it also has the characteristic of wireless interfaces.

Step 2. Add required attributes *classification*, *equipment_type*, and *availability_status*. Use public attribute definitions for each. Because it is an end-system with the full stack, select “Computer” for *equpment_type*.

[End-System Compliance]

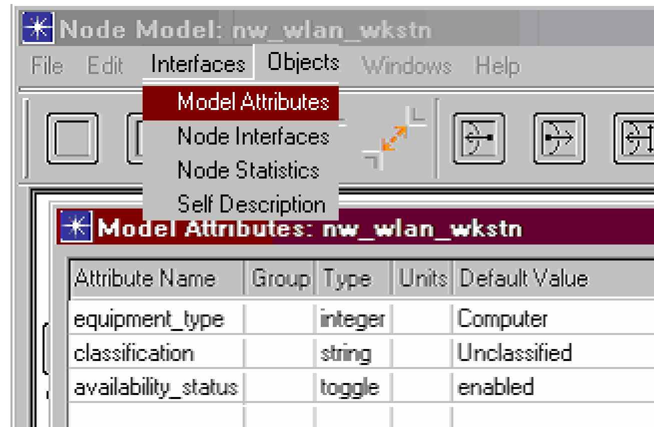


Figure 4-55: Selecting "Computer" for equipment_type

Step 3. Give it the functionality of firing TCP and UDP IERs by adding *se* modules to generate traffic via TCP and another to generate traffic via UDP (*se_tcp* and *se_udp*).

[End-System Compliance]

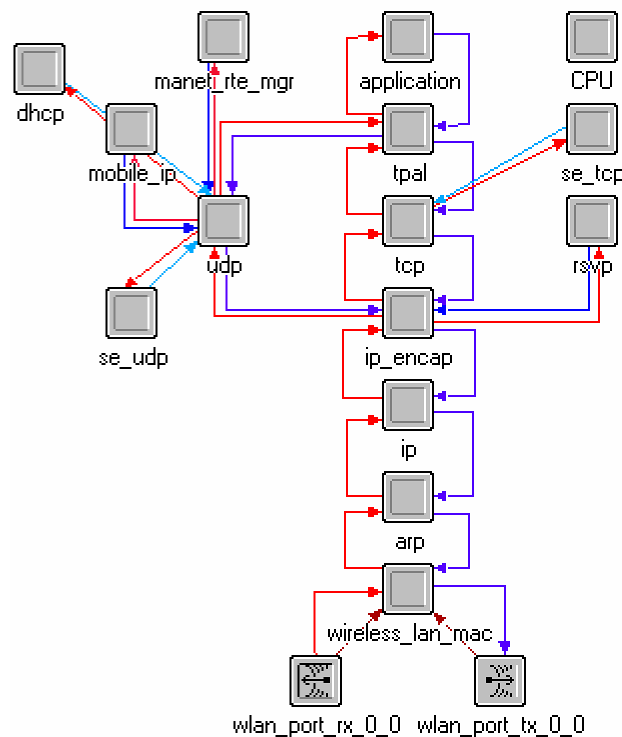


Figure 4-56: Adding se_tcp and se_udp

Step 4. Promote the radio channel properties on the transmitter and receiver and add the *net_id* extended attribute so that the broadcast network object can interface with it.

[Wireless Interface Compliance]

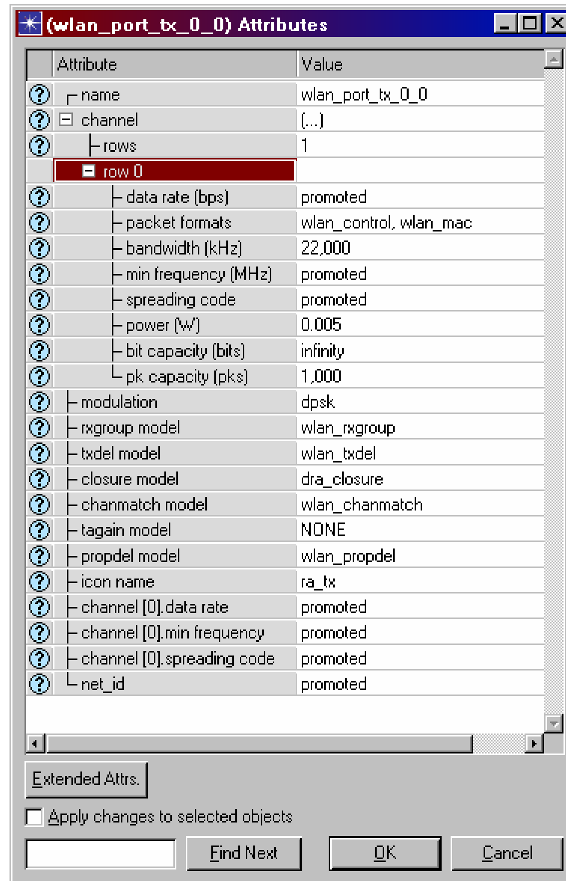


Figure 4-57: Adding *net_id* Extended Attribute

Step 5. Remove the lines of code that set the channel frequency. This now happens via the broadcast network object.

[Wireless Interface Compliance]

wlan_mac Function Block

```
static void
wlan_transceiver_channel_init (void)
    {
    ...
    /* Configure the transmitter channel based on selected/assigned     */
    /* frequency band.                                                  */
    op_ima_obj_attr_set (txch_objid, "bandwidth", bandwidth_mhz * 1000.0);
    //op_ima_obj_attr_set (txch_objid, "min frequency", frequency);

    /* Similarly configure the receiver channel.                        */
    op_ima_obj_attr_set (rxch_objid, "bandwidth", bandwidth_mhz * 1000.0);
    //op_ima_obj_attr_set (rxch_objid, "min frequency", frequency);

    FOUT;
    }
```

wlan_mac_hcf Function Block

```
static void
wlan_hcf_transceiver_channel_init (void)
    {

    ...

    /* Configure the transmitter channel based on selected/assigned     */
    /* frequency band.                                                  */
    op_ima_obj_attr_set (txch_objid, "bandwidth", bandwidth_mhz * 1000.0);
    //op_ima_obj_attr_set (txch_objid, "min frequency", frequency);

    /* Similarly configure the receiver channel.                        */
    op_ima_obj_attr_set (rxch_objid, "bandwidth", bandwidth_mhz * 1000.0);
    //op_ima_obj_attr_set (rxch_objid, "min frequency", frequency);

    FOUT;
    }
```

Step 6. Add a line to the net_configs file to have a Wireless Local Area Network (WLAN) entry.

```
WLAN;Unclassified;11000;2401;"Include";
5000,3000,2000,1000;wlan_control, wlan_mac;wlan_control,wlan_mac
```

4.18 CP MODEL EXAMPLE

4.18.1 Overview

This subsection provides an example on the implementation of a CP compliance model. As mentioned, NETWARS applies analytical techniques to rapidly determine the bandwidth requirements to support specific traffic profiles and patterns. NETWARS will require three basic attributes from the model to determine the CP layer of a specific device: equipment type, interface class, and machine type. These attributes occur in specific locations within the model.

4.18.2 CP Implementation

In order to use the CP function in NETWARS, model developers do not have to insert or modify any code within the node model. It is vital, however, to add the three required attributes into the device model to their associated location. The following subsections will describe the location by using the SINCGARS INC radio model in NETWARS.

4.18.2.1 Equipment Type Attribute

First, the equipment type attribute is used to define the type of the device, such as radio, computer, and router. Figure 4-58 shows the location of the attribute and a list of available types. Model developers should define the equipment_type attribute in the model attributes windows as show in the Figure 4-58.

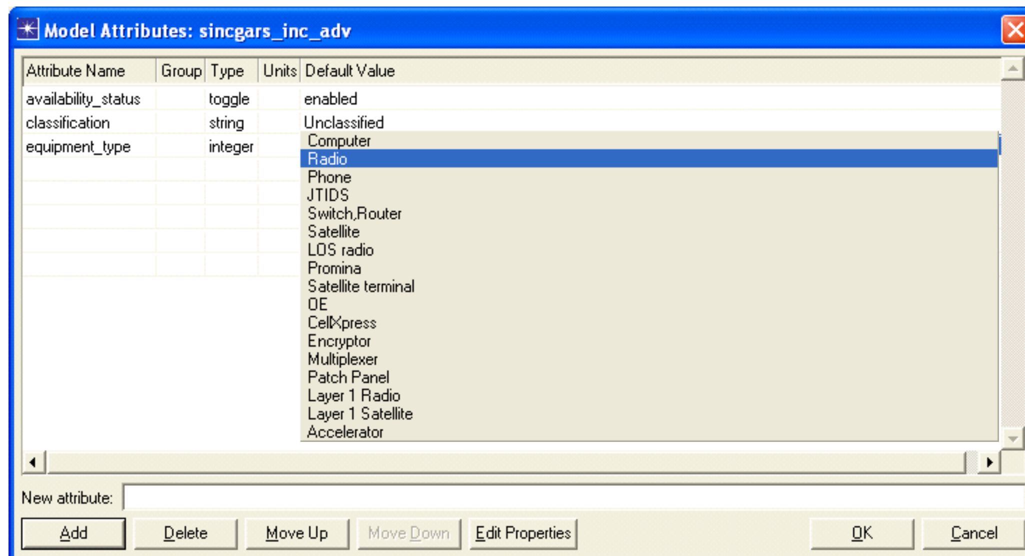


Figure 4-58: Equipment type attribute location

4.18.2.2 Interface Class and Machine Type Attributes

The interface class and machine type attributes are both located in the self-description section of the device model as shown in the Figure 4-59.

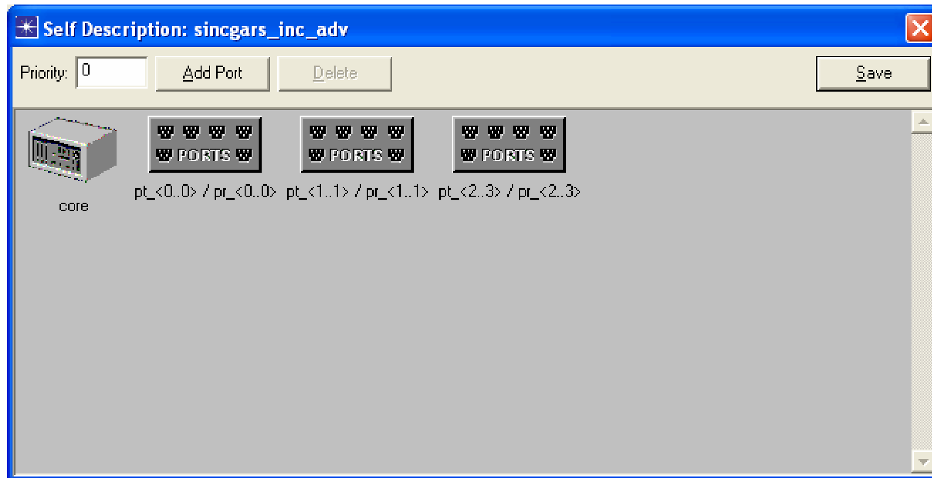


Figure 4-59: Interface Class and Machine type attribute locations

The interface class is defined within the ports description as shown in the following Figure 4-60. In this example, the interface class of the SINCGARS INC device model is IP.

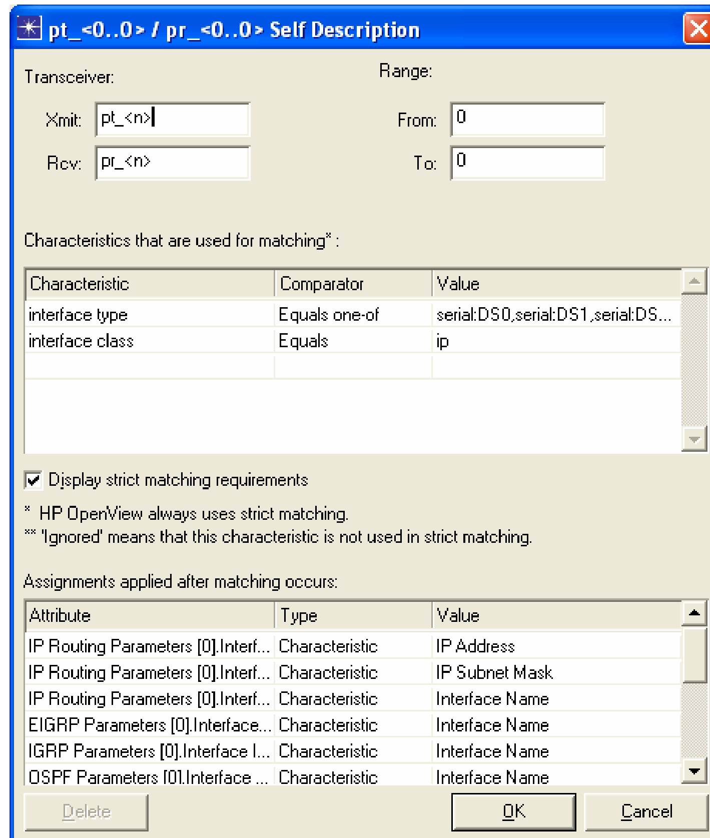


Figure 4-60: Interface Class attribute

Finally, the Machine type attribute is defined within the core section of the self-description. The following Figure 4-61 shows the example of the machine type that is assigned to the SINCGARS INC device model, and the value is router.

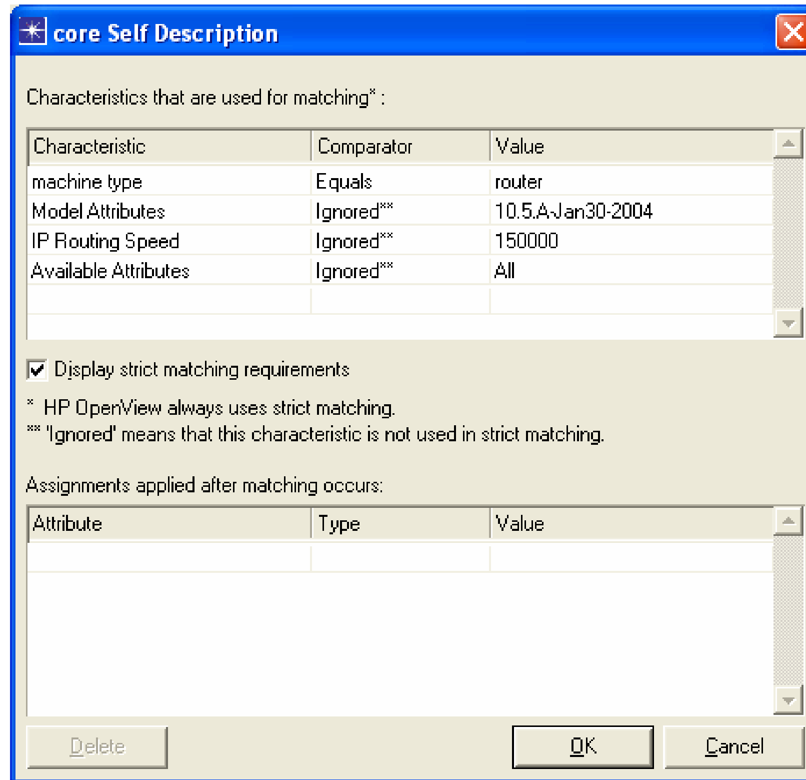


Figure 4-61: Machine type attribute

5 VERIFICATION AND VALIDATION

Verification and Validation (V&V) is important to the creditability of a model that is being used to solve a real world problem. Without that credability, it is extremely difficult to gain buy-in on simulation results.

The importance of V&V is recognized by the Department of Defense in DoD Directive (DoDD) 5000.59 and DoD Instruction (DoDI) 5000.61. These policies describe Verification, Validation, and Accreditation (VV&A) from the standpoint of Policy, Roles, Responsibilities, Processes and Procedures. DoDI 5000.61 established the Defense Modeling Simulation Office (DMSO) as the "DoD VV&A focal point" and the central source of DoD VV&A information. Most of the information from DMSO is addressed in it's VV&A Recommended Practices Guide (RPG), Build 3.0 dated September 2006. There is also a DoD VV&A Documentation Tool that is being developed to assist Model Developers. These references can be found at:

- DoDD 5000.59 – DoD Modeling and Simulation (M&S) Management **http://www.dtic.mil/whs/directives/corres/html/500059.htm**
- DoDI 5000.61 – DoD Modeling and Simulation (M&S) Verification, Validation, and Accreditation (VV&A) **http://www.dtic.mil/whs/directives/corres/html/500061.htm**
- VV&A Recommended Practices Guide – Build 3.0 / September 2006 **http://vva.dmso.mil/**

The accreditation portion may or may not be significant for the NETWARS Model Developer. According to the DoD Policy, all models should go through V&V, however, not all models need to be accredited. DoDD 5000.59 discusses two primary instances where accreditation is required; when the model is going to be reused by an external organization, or results will be used in the acquisition process. In addition, all DoD Components should have their own set of policies and procedures that a Model Developer should adhere to in their development process.

The RPG provides guidance on VV&A for general purpose M&S. VV&A is about establishing the relationship between the problem and the model being used to solve that problem. There are not any definitive steps that apply to V&V, since V&V needs to be tailored to match the nature of the problem that is being addressed by the M&S application. Some of the factors involved in tailoring V&V to a general purpose M&S application are:

- Situations being simulated
- Types of decisions driving the employment of the simulation
- Nature of the simulation
- Level of risk
- Technical or resource limitations

The scope of the following discussion within the Model Development Guide (MDG) will limit itself to V&V of NETWARS-compliant models within the NETWARS product environment. The focus is to provide high level guidance for V&V of the design and functions of a model and for ensuring the newly developed model will integrate into NETWARS. Since accreditation may or may not be required, dependent on the specific DoD Component policies, the MDG will not discuss accreditation any further.

The following sub-sections are grouped into two primary V&V objectives: first is to V&V the functionality of the models, and second is to V&V the model that can be integrated to NETWARS.

5.1 MODEL FUNCTIONAL V&V

This section will focus on introducing the basic V&V steps and references to test and examine the basic required functionalities and accuracies of the model.

5.1.1 Objectives

The primary objective for V&V on models is to provide credibility and believability to the results that those models generate, so that the results may be used in solving real world problems. It is also important to note that the data used to drive the model should be evaluated together with the model, as the model depends on the data to provide realistic simulation. Data V&V is well documented in the DMSO RPG.

The definitions for verification and validation are often confused:

- **Verification** - The process of determining that a model implementation and its associated data accurately represent the developer's conceptual description and specifications.
- **Validation** - The process of determining the degree to which a model and its associated data provide an accurate representation of the real world from the perspective of the intended uses of the model.

Verification seeks to answer the question, "Did I build the thing right?" while validation seeks to answer the question "Did I build the right thing?" Answering these questions positively with sufficient explanation will create believability in the results generated or the validity of the model for those seeking to reuse it.

5.1.2 Steps

The Model Developer should follow the applicable DoD Component's policies and procedures in accordance with DoD Directives and Instructions. The RPG has very detailed guidelines regarding V&V for new models, modification of models (legacy), and federated models by the different types of user views. The following RPG Problem Solving Process demonstrates the standpoint that VV&A is an integral part of the M&S development process. The focus in the MDG will be on the box entitled "Perform V&V Activities appropriate for M&S Category".

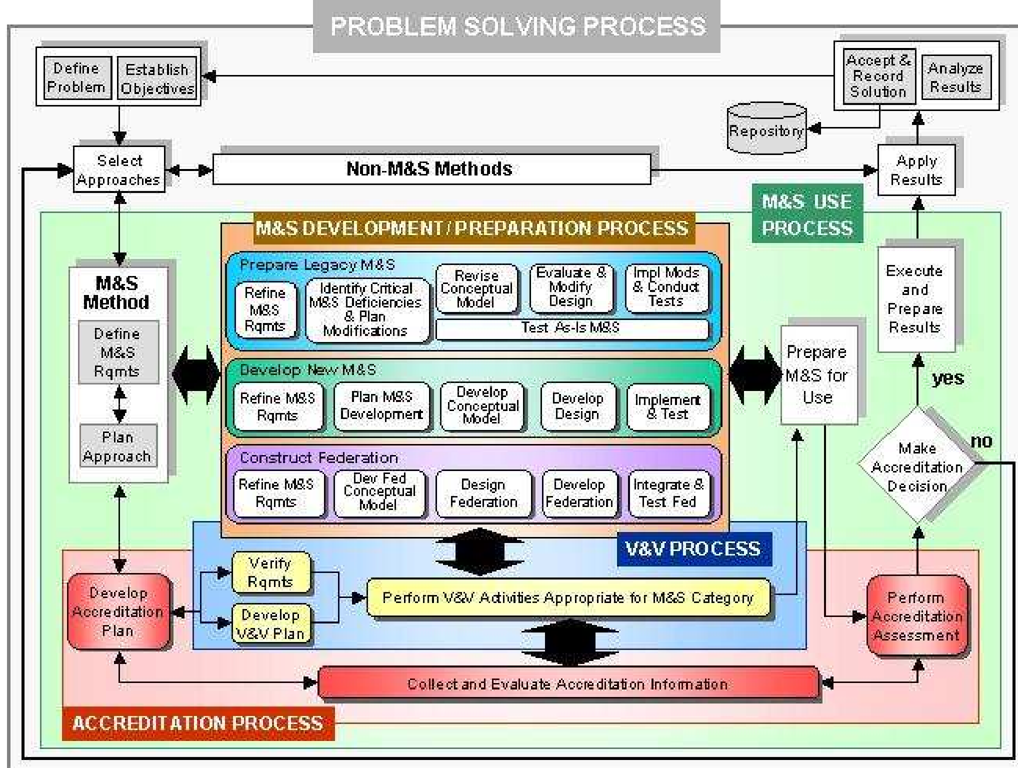


Figure 5-1: M&S Overall Problem Solving Process

The steps to augment the Model Developer's DoD Component's policies and procedures specific to NETWARS Compliance V&V are included in Appendix X: NETWARS Model Development Guide Checklist. The steps that will be discussed in further detail in the next section are:

- Following the NETWARS Model Development Guide Checklist
- Static Testing
- Equipment String
- Capacity Planner

An important reference regarding V&V is the "NETWARS Communications Model Verification and Validation Plan." This document defines the NETWARS structured, repeatable process for ensuring that all communications device models included in NETWARS are reasonable representations of the intended actual systems. This includes constraints on how those modules should be employed. The document describes several phases, of which the final phase focuses on model integration into NETWARS.

Another document that can be referenced is the DoD VV&A Documentation Tool developed by Space & Naval Warfare Systems Command (SPAWAR).

It is a good practice to add a brief validation time stamp and the model development Point of Contact information in the self-description of the model so that users can contact the model developers or corresponding individual to resolve any issues.

5.2 NETWARS COMPLIANCE V&V

The primary objective of this NETWARS MDG is to ensure that newly developed models can be integrated to NETWARS and shared with the NETWARS community. The NETWARS compliance V&V is important, therefore, to both the model developers and the model users. This section will introduce the resources that can be utilized by the developer to perform NETWARS compliance V&V. These resources include the NETWARS Model Development Checklist, NETWARS Static Testing, NETWARS Equipment String, and Capacity Planner Attributes.

5.2.1 NETWARS Model Development Checklist

The NETWARS Model Development Guide Checklist is the first tool to ensure that newly developed models can be integrated to the NETWARS standard model library. The Checklist can be found in Appendix X. The checklist is used to provide a basic development check for the developers to ensure NETWARS compliance; however, the checklist cannot provide full coverage to ensure the compliance.

The checklist can be used for new development or modification of existing OPNET COTS models for NETWARS Compliance, and covers the following areas:

- General Questions regarding the model goals and attributes
- Traffic-generation mechanisms
- Static Testing
- Equipment Strings
- Capacity Planner
- Model Documentation
- Model interfaces to the NETWARS standard pallet of devices
- Model node modules and port conventions
- Model modules included for end systems
- Model attributes for radio broadcast and point-to-point operations
- Model custom links

If the user submits a model for development, the developer should leave contact information inside the self description, such that other organizations may contact them for more information about the model they have developed.

5.2.2 NETWARS Static Testing

The NETWARS Static Testing Tool comes with NETWARS. Static Testing will perform checks of the syntax of a model. The Static Testing documentation should be consulted for further detail on its functionality. Some of the items that will be checked by Static Testing include:

- Minimum Attributes Test

- Check for Tx/Rx naming conventions
- Check for the presence of required modules
- Check for supported packet formats
- Check for interface capability with other equipment
- Check for handling of failure and recovery
- Check for pipeline stage transmitter attributes
- Check for pipeline stage receiver attributes

If a model fails Static Testing, then those points of failure should raise flags. It is important that those flags be addressed even though they do not necessarily by themselves indicate that a model is not NETWARS compliant. The important questions to answer are; "Does the Model Developer care about the raised flag?" and "What are the consequences of the raised flag?" It is possible that mitigation of a raised flag might have to do with different attributes for different equipment types.

Refer to the "NETWARS 2006-2 Communications Device Model Validation and Verification Plan" for further information.

5.2.3 NETWARS Equipment String

In order to ensure that new models are NETWARS-compliant, they should be tested using some basic equipment strings that are relevant to the model that was developed. NETWARS Program Management Office (DISA GE344) has a "NETWARS Equipment Strings Version 1.1, June 2006" document. This living document contains valid equipment strings that involve NETWARS models. This document breaks the equipment strings down into the following categories:

- **Transmission Network**
  - Pure Transmission Devices
  - Prominas
  - Other Multiplexers
- **Routers** – devices that can go over any of the transmission network devices
- **Circuit Switched Voice** – voice circuits that go over all the transmission network devices and can flow over IP or ATM network
- **Layer-1 Encryptors** – paired up on either WAN or LAN side, if follows a router, then decryption must occur before the next router
- **Tactical Radios** – include havequick, jtids, sincgars and eplrs
- **Invalid Equipment Strings** – illogical and unsupported

Another important reference is the "NETWARS 2006-2 Equipment Strings Final Test Plan, OPNET 3.4.4, delivered August 25, 2006." This document provides tests for NETWARS model feature requirements. Some examples of test procedures provided are; SATCOM device equipment strings, Terrestrial Radio equipment strings, Promina to Promina equipment strings, etc.

Most important, developers should determine the equipment strings associated with their models and develop corresponding testing of the strings with the models in the NETWARS standard library.

5.2.4 Capacity Planner

The NETWARS network analytical engine is important for providing network capacity planning support to the network planner. It has the ability to generate shortest-hop routing, calculations of link and circuit utilization, and bandwidth requirements for support of specific traffic profiles and patterns. CP is a NETWARS-specific capacity, therefore models developed for OPNET Modeler and IT-Guru cannot be applied in CP. In order to ensure models are NETWARS-compliant with regards to CP and routing, device attributes and properties should be correctly developed. They include:

- Equipment Type
- Self Description

The static test software is a good tool for verifying that all attributes used by CP are available in the model for the specific model type under the minimum attributes test in the static test software.

The static test may be run from the NETWARS Console. In order to open the NETWARS Console; Go to "**Start**" → "**All Programs**" → "**NETWARS**" → "**NETWARS Console**". Once the NETWARS Console is opened, from the "C:\>" prompt, type in "**mdgtest**"

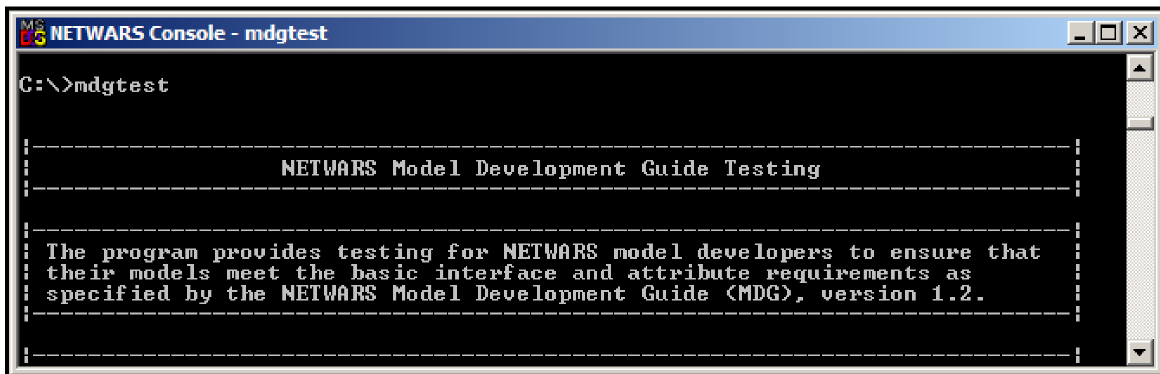


Figure 5-2: Initiate a static test

After providing some helpful text, the "mdgtest" program will stop and wait for the name of the device model that should be validated. In the example, "nw_ethernet_wkstn" was typed in following the prompt for "Model name: ".

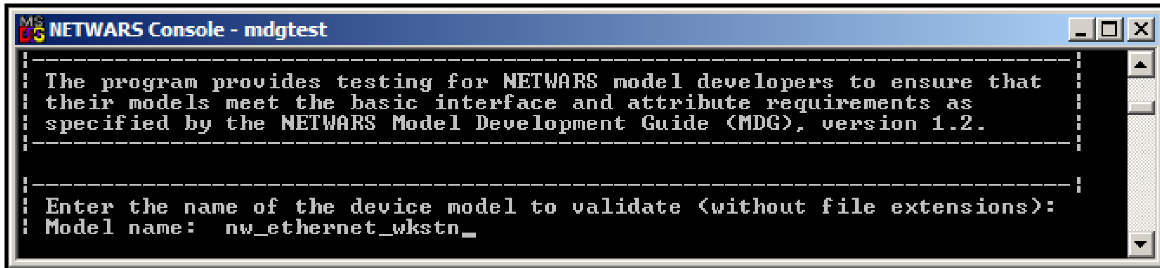


Figure 5-3: Execute a static test for the nw_ethernet_wkstn device

The "mdgtest" program will stop and wait for the component class name. In the example, "4" was typed in following the prompt for "Component Class: " indicated the "End System".

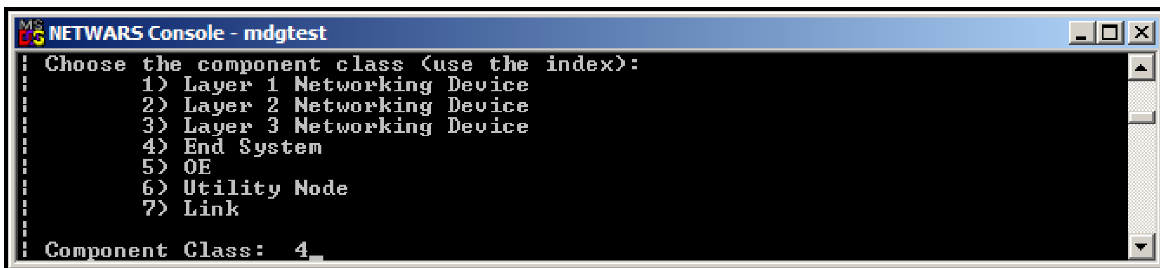


Figure 5-4: Select component class for static test

The "mdgtest" program will stop and wait for the model options. In the example, the <Enter> key was hit for N/A.

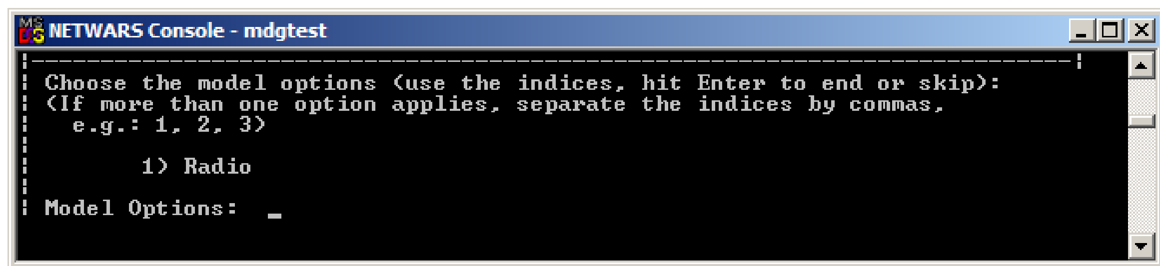


Figure 5-5: Select model options for static test

The "mdgtest" program will stop and wait for the protocols. In the example, "1, 2, 4" was typed in following the prompt for "Protocols: " indicating the TCP, UDP, and Ethernet protocols are to be used.

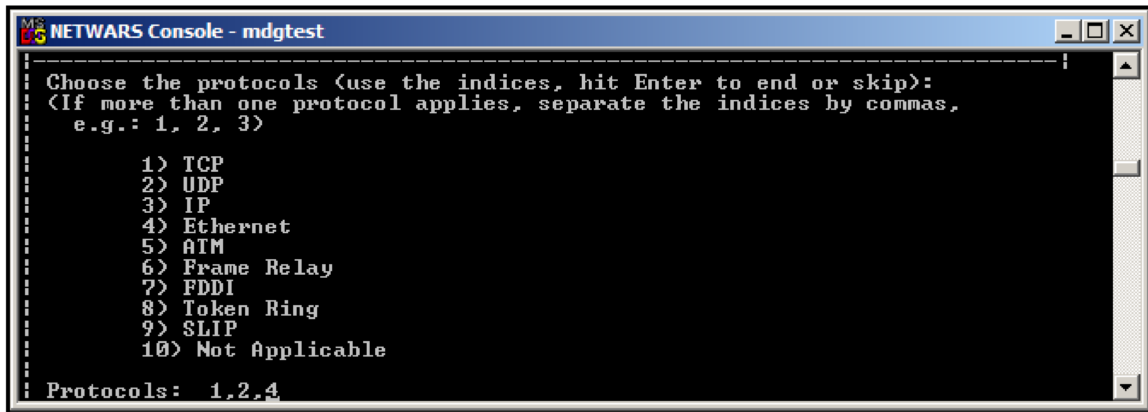


Figure 5-6: Select protocols for static test

The "mdgtest" program will stop and wait for the name of the report file. In the example, "C:\test.txt" was typed in following the prompt for "Report File: " indicating that is where the output from the static test will be placed. After entering the report file name, the question to confirm the answers is asked. If the answers are correct and you want to continue, then a "c" may be typed in.

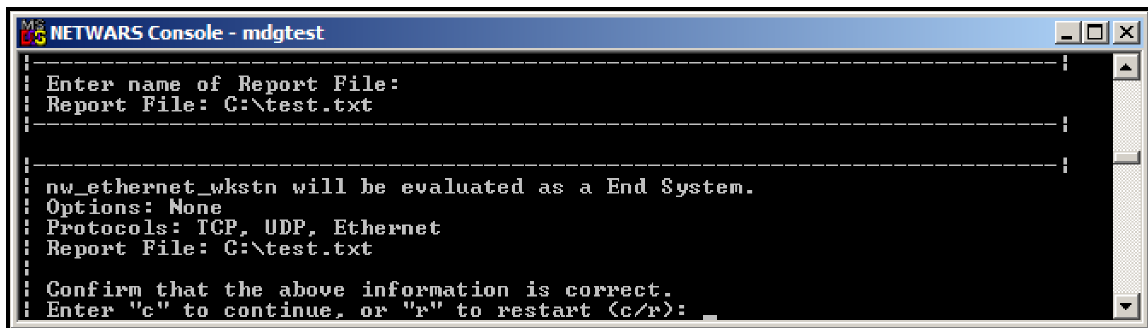


Figure 5-7: Select report file name and confirm answers for static test

The "mdgtest" program will finish with a summary and a completion message.

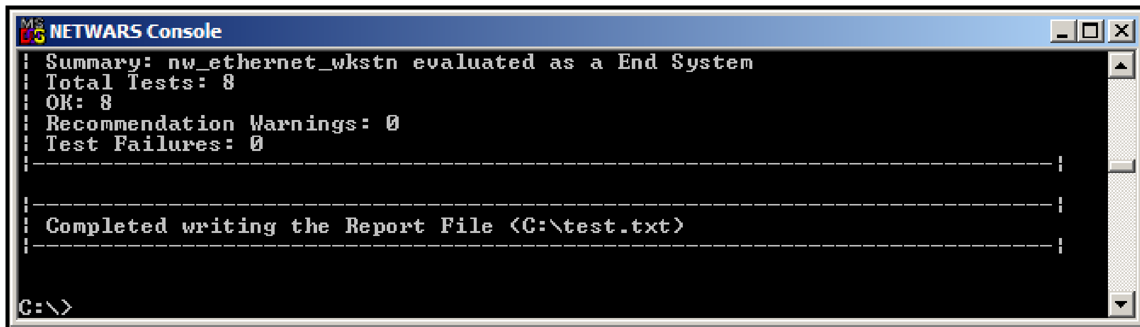
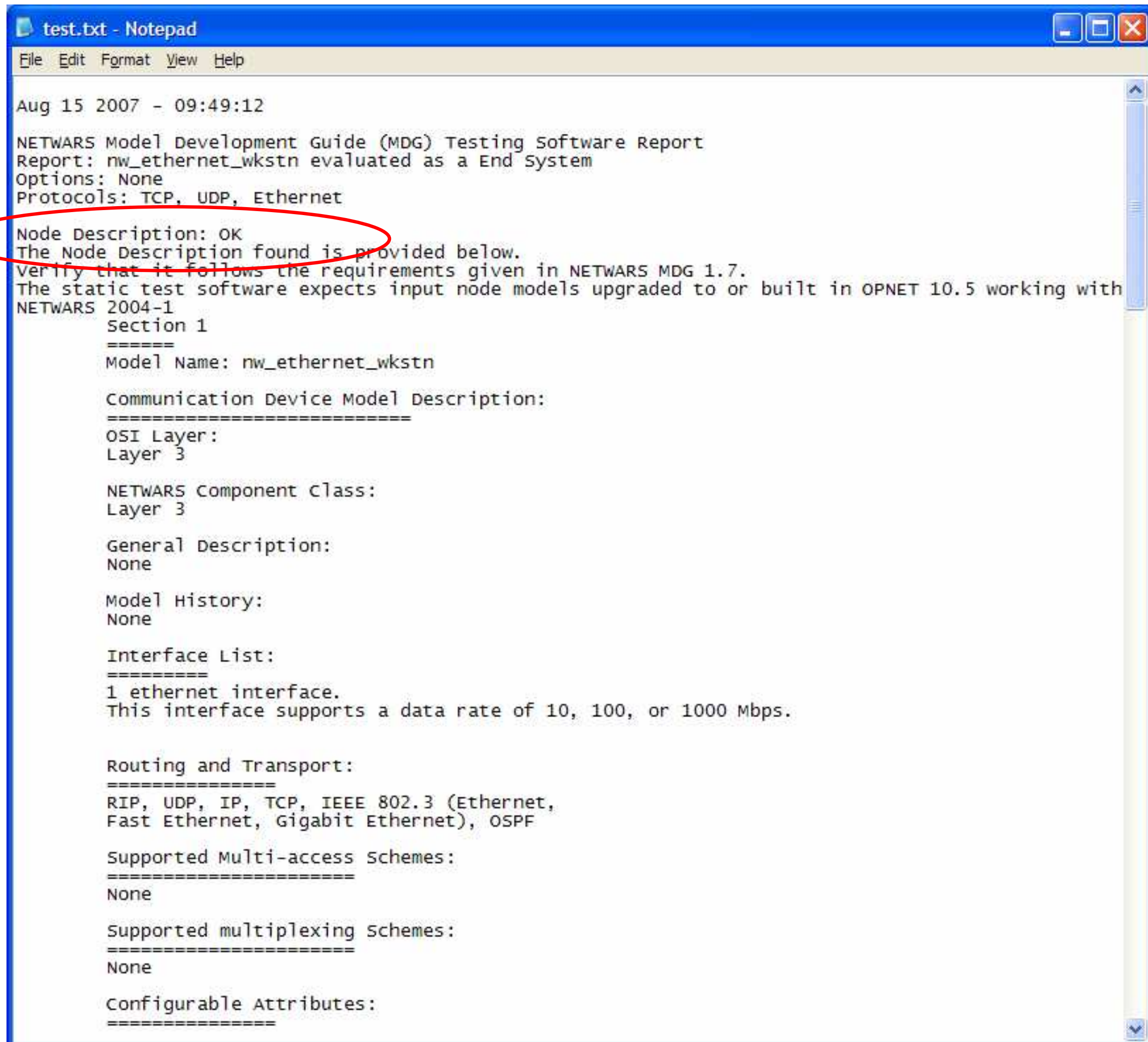


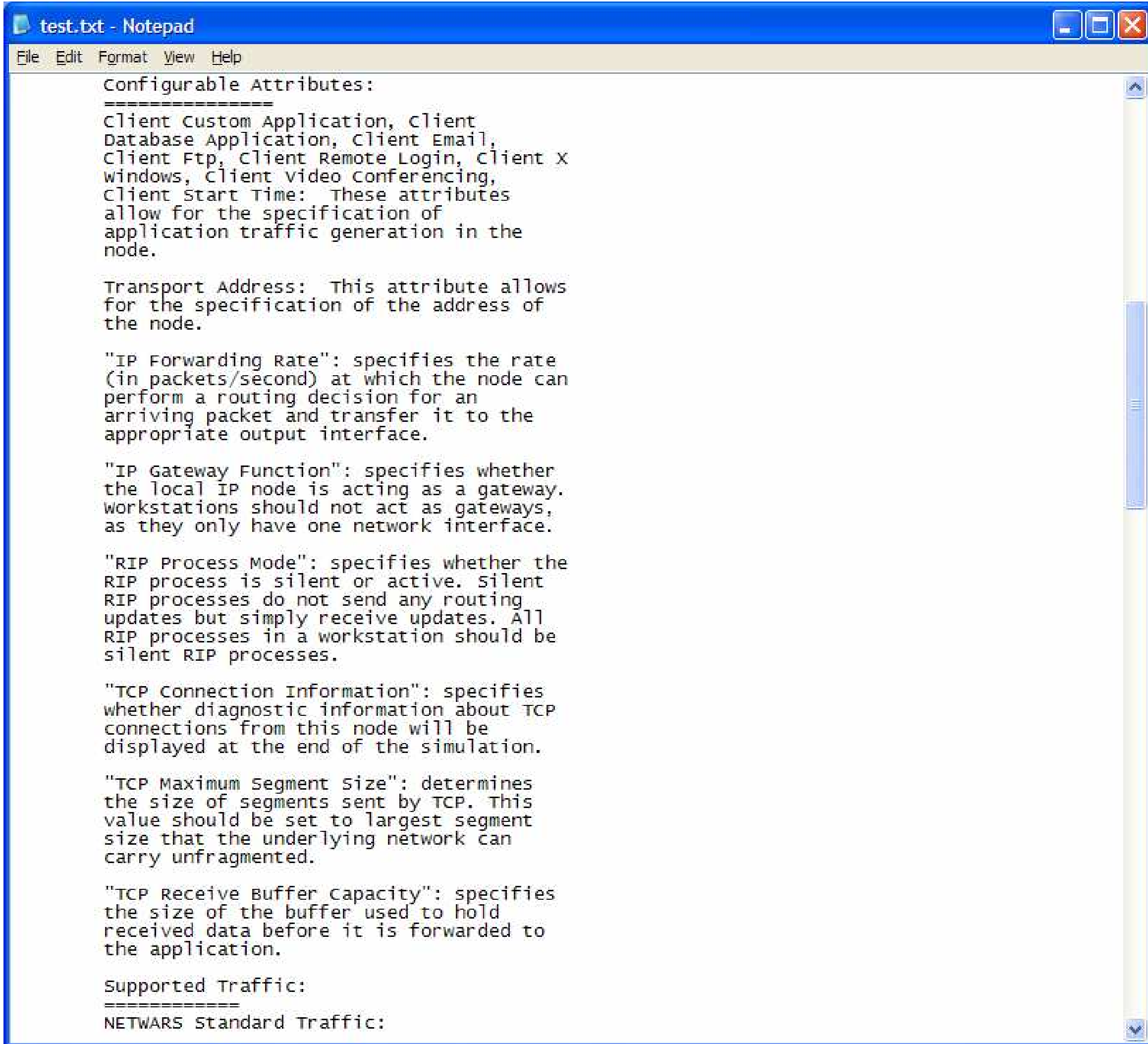
Figure 5-8: Summary and completion message for static test

In order to see the static test report generated from this example, type in “notepad C:\test.txt” at the “C:\>” prompt. The “notepad” program will display the static test report generated by the “mdgtest” program.




```
Aug 15 2007 - 09:49:12

NETWARS Model Development Guide (MDG) Testing Software Report
Report: nw_ethernet_wkstn evaluated as a End System
Options: None
Protocols: TCP, UDP, Ethernet

Node Description: OK
The Node Description found is provided below.
Verify that it follows the requirements given in NETWARS MDG 1.7.
The static test software expects input node models upgraded to or built in OPNET 10.5 working with
NETWARS 2004-1
        Section 1
        ======
        Model Name: nw_ethernet_wkstn

        Communication Device Model Description:
        ===========================
        OSI Layer:
        Layer 3

        NETWARS Component Class:
        Layer 3

        General Description:
        None

        Model History:
        None

        Interface List:
        =========
        1 ethernet interface.
        This interface supports a data rate of 10, 100, or 1000 Mbps.


        Routing and Transport:
        ===============
        RIP, UDP, IP, TCP, IEEE 802.3 (Ethernet,
        Fast Ethernet, Gigabit Ethernet), OSPF

        Supported Multi-access Schemes:
        ======================
        None

        Supported multiplexing Schemes:
        ======================
        None

        Configurable Attributes:
        ===============
```

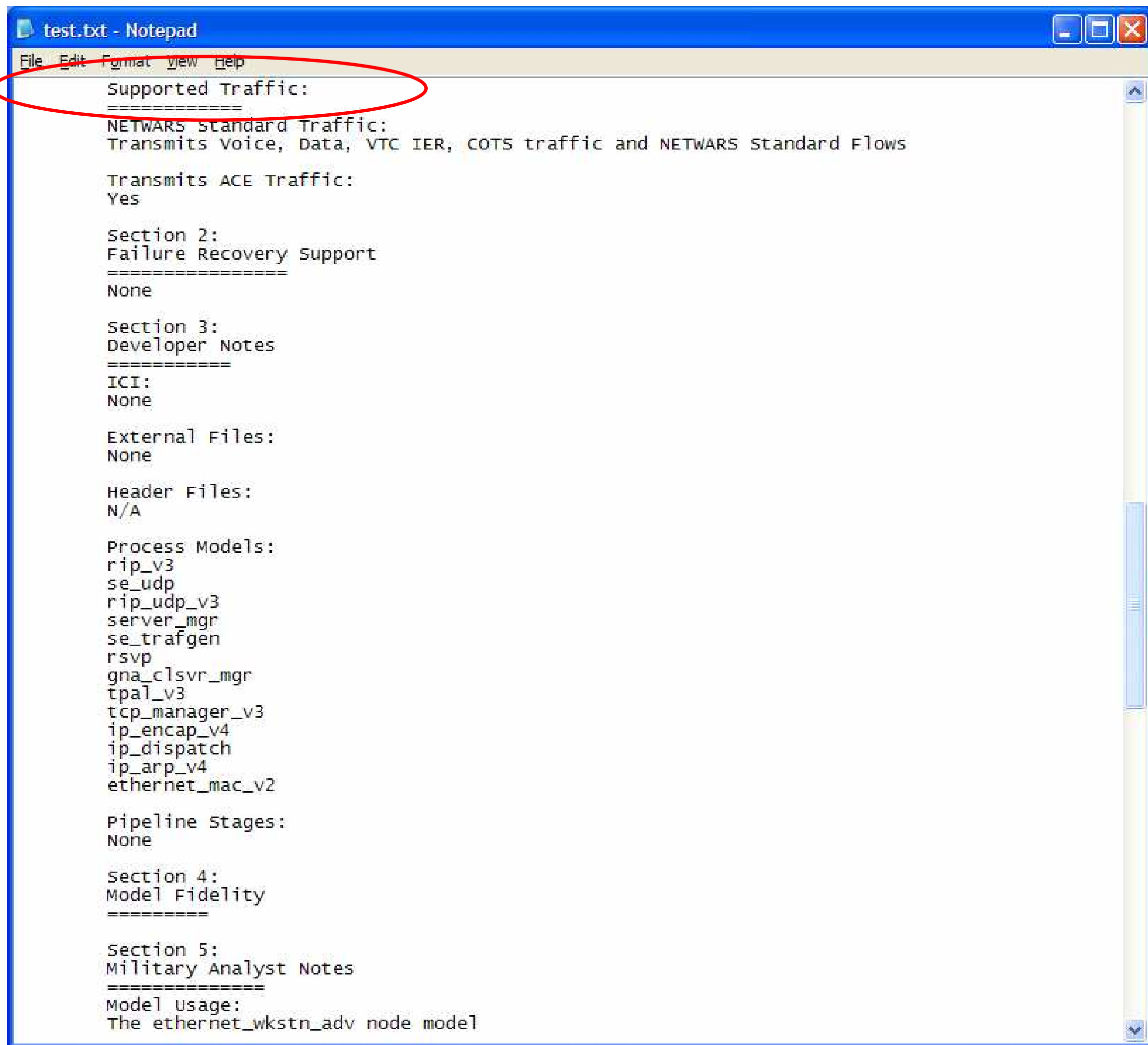
Figure 5-9: Static test report


test.txt - Notepad

File Edit Format View Help

```
Configurable Attributes:
===============
Client Custom Application, Client
Database Application, Client Email,
Client Ftp, Client Remote Login, Client X
Windows, Client Video Conferencing,
Client Start Time:  These attributes
allow for the specification of
application traffic generation in the
node.

Transport Address:  This attribute allows
for the specification of the address of
the node.

"IP Forwarding Rate": specifies the rate
(in packets/second) at which the node can
perform a routing decision for an
arriving packet and transfer it to the
appropriate output interface.

"IP Gateway Function": specifies whether
the local IP node is acting as a gateway.
Workstations should not act as gateways,
as they only have one network interface.

"RIP Process Mode": specifies whether the
RIP process is silent or active. Silent
RIP processes do not send any routing
updates but simply receive updates. All
RIP processes in a workstation should be
silent RIP processes.

"TCP Connection Information": specifies
whether diagnostic information about TCP
connections from this node will be
displayed at the end of the simulation.

"TCP Maximum Segment Size": determines
the size of segments sent by TCP. This
value should be set to largest segment
size that the underlying network can
carry unfragmented.

"TCP Receive Buffer Capacity": specifies
the size of the buffer used to hold
received data before it is forwarded to
the application.

Supported Traffic:
============
NETWARS Standard Traffic:
```

Figure 5-10: Static test report 2


Supported Traffic:
============
NETWARS Standard Traffic:
Transmits Voice, Data, VTC IER, COTS traffic and NETWARS Standard Flows

Transmits ACE Traffic:
Yes

Section 2:
Failure Recovery Support
================
None

Section 3:
Developer Notes
===========
ICI:
None

External Files:
None

Header Files:
N/A

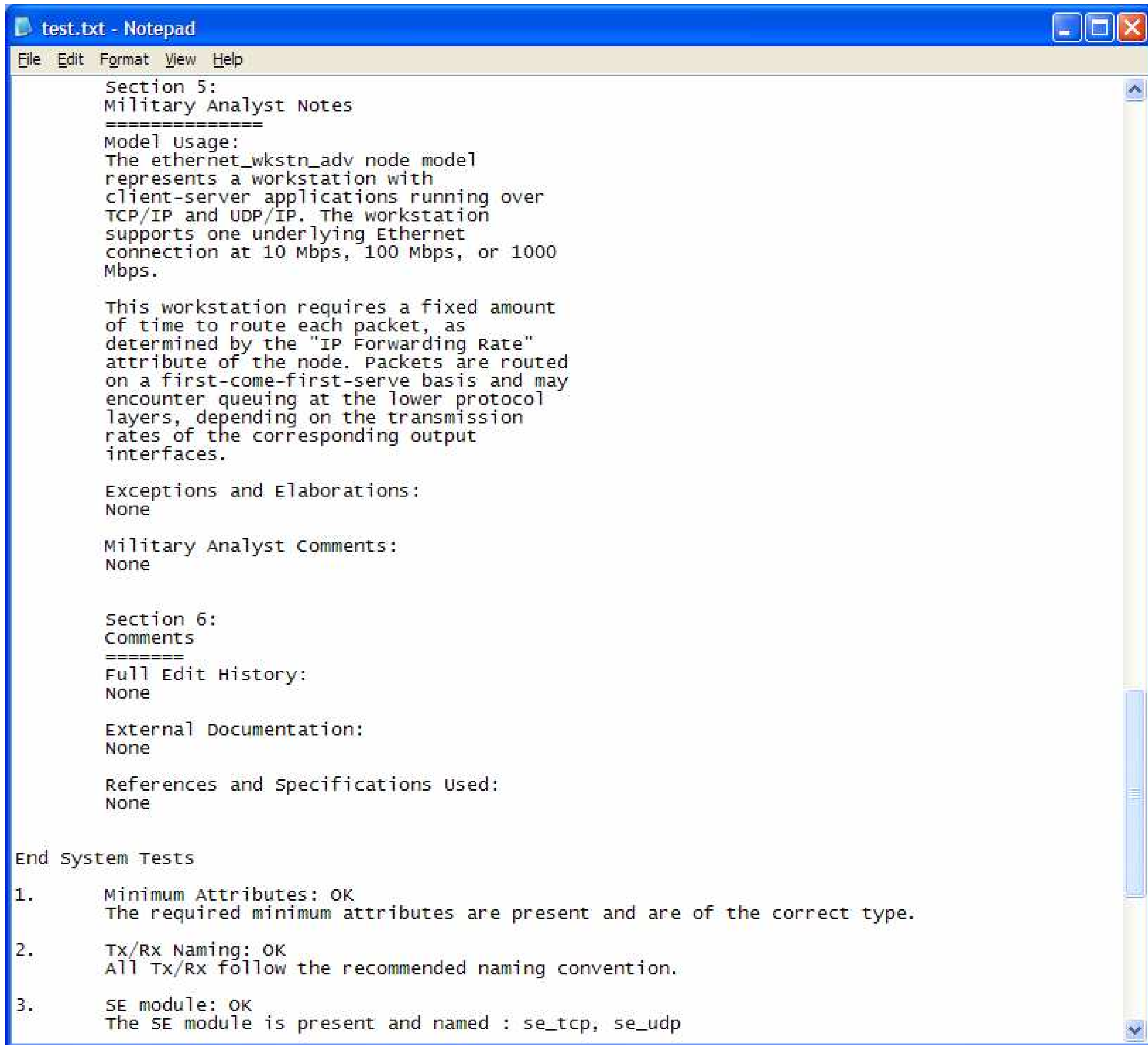
Process Models:
rip_v3
se_udp
rip_udp_v3
server_mgr
se_trafgen
rsvp
gna_clsvr_mgr
tpal_v3
tcp_manager_v3
ip_encap_v4
ip_dispatch
ip_arp_v4
ethernet_mac_v2

Pipeline Stages:
None

Section 4:
Model Fidelity
=========

Section 5:
Military Analyst Notes
==============
Model Usage:
The ethernet_wkstn_adv node model

Figure 5-11: Static test report 3



```
test.txt - Notepad
File Edit Format View Help
        Section 5:
        Military Analyst Notes
        ==============
        Model Usage:
        The ethernet_wkstn_adv node model
        represents a workstation with
        client-server applications running over
        TCP/IP and UDP/IP. The workstation
        supports one underlying Ethernet
        connection at 10 Mbps, 100 Mbps, or 1000
        Mbps.

        This workstation requires a fixed amount
        of time to route each packet, as
        determined by the "IP Forwarding Rate"
        attribute of the node. Packets are routed
        on a first-come-first-serve basis and may
        encounter queuing at the lower protocol
        layers, depending on the transmission
        rates of the corresponding output
        interfaces.

        Exceptions and Elaborations:
        None

        Military Analyst Comments:
        None


        Section 6:
        Comments
        =======
        Full Edit History:
        None

        External Documentation:
        None

        References and Specifications Used:
        None

End System Tests

1.      Minimum Attributes: OK
        The required minimum attributes are present and are of the correct type.

2.      Tx/Rx Naming: OK
        All Tx/Rx follow the recommended naming convention.

3.      SE module: OK
        The SE module is present and named : se_tcp, se_udp
```

Figure 5-12: Static test report 4

```
test.txt - Notepad
File  Edit  Format  View  Help
        Comments
        =======
        Full Edit History:
        None

        External Documentation:
        None

        References and Specifications Used:
        None

End System Tests
1.      Minimum Attributes: OK
        The required minimum attributes are present and are of the correct type.

2.      Tx/Rx Naming: OK
        All Tx/Rx follow the recommended naming convention.

3.      SE module: OK
        The SE module is present and named : se_tcp, se_udp

4.      Required Modules: OK
        All required modules are present.
        TCP: OK
        UDP: OK
        Ethernet: OK

5.      Supported Packet Formats: OK
        All required packet formats are present.
        TCP: OK
        UDP: OK
        Ethernet: OK

6.      Interfacing with Networking Equipment: OK
        All Tx/Rx have data rate set to unspecified, as recommended by the
        NETWARS MDG 1.7 (for auto-sensing ports).

7.      Handling Failure/Recovery: OK
        At least one of the modules in the device is set up for explicit handling of
Failure/Recovery.


Test Summary

Total Tests: 8
OK: 8
Recommendation Warnings: 0
Test Failures: 0
```

Figure 5-13: Static test report 5

The preceding was an example of how to perform a static test, both the input for the static test and the output that can be expected from running the static test.

The developer should test their models in CP to ensure the required model attributes and CP APIs are implemented into their models. For further information, please see the individual sections on model development that relate to CP in this document.

5.2.5 DoD/Joint VV&A Documentation Tool (DVDT/JVDT)

DVDT/JVDT is a tool that assists the user in creating and maintaining four major documents required in the VV&A process:

- Accreditation Plan
- VV&A Plan
- VV&A Report
- Accreditation Report.

This tool is not part of OPNET, nor is it part of NETWARS, however, it is being presented here as a reference to assist in MDG Developers task of VV&A.

REFERENCES:

1. DoD Standard Practice: Documentation of Verification, Validation and Accreditation (VV&A) for Models and Simulations. (MIL-STD-XXX002, Draft of 5 December 2006). It is headed by this caveat:

   NOTE: This draft, dated 5 December 2006, prepared by the Defense Modeling and Simulation Coordination Office, has not been approved and is subject to modification. DO NOT USE PRIOR TO APPROVAL (Project MSSM-2005-002)

APPENDIX A: ACRONYMS

Acronym	Definition
ACE	Applications Characterization Environment
ACK	ACK
AEHF	Advanced Extremely High Frequency
ALE	Automatic Link Establishment
AODV	Ad Hoc on Demand Distance Vector
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BER	Bit Error Rate
BGP	Border Gateway Protocol
C4I	Joint Command, Control, Communications, Computers and Intelligence
CJCSM	Chairman of the Joint Chiefs of Staff Manual
CLEO	Cisco Router Low Earth Orbit
CM	Configuration Management
CNR	Combat Network Radio
COE	Common Operating Environment
COI	Community of Interest
COTS	Commercial Off-the-Shelf
CP	Capacity Planner
CPC	Communications Planning Coordinator
CPU	Central Processing Unit
CSV	Comma Separated Value
DAMA	Demand-Assigned Multiple Access
DE	Deployment Editor
DES	Discrete Event Simulation
DHCP	Dynamic Host Configuration Protocol
DISA	Defense Information Systems Agency
DMSO	Defense Modeling Simulation Office
DNVT	Digital Non-Secure Voice Terminal
DoD	Department of Defense
DoDAF	DoD Architecture Framework
DoDD	DoD Directive
DoDI	DoD Instruction
DSL	Digital Subscriber Line
DSR	Dynamic Source Routing
DTED	Digital Terrain Elevation Data
DTG	Digital Transmission Group
DVDT	DoD VV&A Documentation Tool
ECC	Error Correction Calculation
EIGRP	Extended Interior Gateway Routing Protocol
EMA	External Model Access
EPLRS	Enhanced Position Location Reporting System
ETSSP	Enhanced TSSP
FAQ	Frequently Asked Questions

Acronym	Definition
FCC	Federal Communication Commission
FDDI	Fiber Distributed Data Interface
FDMA	Frequency Division Mutiple Access
FR	Frame Relay
FTP	File Transfer Protocol
GBS	Global Broadcast Service
GOE	Generic Organization Editor
GOTS	Government Off-the-Shelf
GUI	Graphical User Interface
HDR	High Data Rate
HF	High Frequency
HLA	High-Level Architecture (IEEE Standard 1516)
HTTP	Hypertext Transport Protocol
ICI	Interface Control Information
IEEE	Institute of Electrical and Electronics Engineers
IER	Information Exchange Requirement
IGRP	Interior Gateway Routing Protocol
IMEP	Internet MANET Encapsulation Protocol
INC	Internet Controller
INE	Inline Network Encryptor
IP	Internet Protocol
ISDN	Integrated Services Digital Network
JTIDS	Joint Tactical Information Distribution System
JVDT	Joint VV&V Documentation Tool
KP	Kernel Process
LAN	Local Area Network
LDR	Low Data Rate
LDW	Link Deployment Wizard
LOS	Line of Site
M&S	Modeling and Simulation
MAC	Medium Access Control
MANET	Mobile Ad Hoc Network
MDG	Model Development Guide
MILSAR	Military Strategic, Tactical and Relay
MIL-STD	Military Standard
MOP	Measure of Performance
MPLS	Multiprotocol Label Switching
MSE	Mobile Subscriber Equipment
NACK	Negative Acknowledgment
NCES	Net-Centric Enterprise Service
NCS	Network Control Center
NETWARS	Network Warfare Simulation
NPG	Network Participation Group
OE	Operational Element
OLSR	Optimized Link State Routing

Acronym	Definition
OMS	OPNET Model Support
OPFAC	Operational Facility
OPSIT	Operational Scenario in Time
Org	Organization
OSI	Open Systems Interconnect
OSPF	Open Shortest Path Forwarding
OV	Output Vector
PNNI	Private Network-to-Network Interface
POTS	Plain Old Telephone Service
PPP	Point-to-Point Protocol
QAE	Quality Assurance Engineer
QDR	Quadrennial Defense Review
QoS	Quality of Service
RF	Radio Frequency
RIP	Routing Information Protocol
RP	Resource Planner
RPG	Recommended Practices Guide
RSVP	Resource Reservation Protocol
SATCOM	Satellite Communications
SB	Scenario Builder
SCM	Scenario Conversion Module
SDF	Simulation Description File
SE	System Element
SHF	Super High Frequency
SINCGARS	Single-Channel Ground and Airborne Radio System
SLIP	Serial Line Internet Protocol
SME	Subject Matter Expert
SMU	Switch Multiplexer Unit
SNR	Signal-to-Noise Ratio
SOA	Service-Oriented Architecture
SPAWAR	Space & Naval Warfare Systems Command
STD	State Transition Diagram
STEP	Standardized Tactical Entry Point
STU-III	Secure Telephone Units III
T&E	Testing and Evaluation
TACSIT	Tactical Situation
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TIREM	Terrain Integrated Rough Earth Model
TMM	Terrain Modeling Module
TORA	Temporally Oriented Routing Algorithm
TPAL	Transport Protocol Adaptation Layer
TRC	Transmission Release Code
Troposcatter	Tropospheric Scatter
TSSP	Tactical Satellite Signal Processing

Acronym	Definition
UDP	User Datagram Protocol
UHF	Ultra High Frequency
UML	Unified Modeling Language
USGS DEM	United States Geological Survey Digital Elevation Model
V&V	Validation and Verification
VHF	Very High Frequency
VOACAP	Voice of America Communications, Analysis, and Prediction
VTC	Video Teleconferencing
VV&A	Verification, Validation, and Accreditation
WAN	Wide Area Network
WiFi	Wireless Fidelity
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network
XML	Extended Markup Language

APPENDIX B: GLOSSARY

Deployment Editor: A tool used by the study analyst to deploy organizations in scenario and run simulations. This has been replaced by what is now called Scenario Builder in NETWARS.

Generic organization: This is a hierarchical collection of OPFACs, organizations, and communications infrastructure. It can be thought of as a template organization that can be instantiated in a scenario. For example, the study analyst can create a generic organization called "Platoon" and use this in another organization called "Company" or in a scenario.

Kernel procedure: An OPNET-provided function that supports the development of protocols and algorithms. All kernel procedures start with op_.

mod_dirs: An environment attribute that tells OPNET in which folders to look for locating files. The mod_dirs attribute is found under Edit->Preferences.

Online documentation: An Adobe Acrobat manual that has information about the OPNET models, kernel procedures, modeling concepts, etc. The manual can be launched from Modeler by choosing the Online Documentation option under the Help menu.

Process registry: This is a model-wide registry where any process mode can register itself and any process model can obtain information about other process models that are registered. For a list of kernel procedures available for using the process registry, refer to the OPNET Modeler online documentation, General Models manual, "OPNET Model Support" chapter, and "Process Registry" section.

Scenario: This is a collection of organizations and communications infrastructure. The organizations in a scenario have trajectories and positions assigned to them. After a scenario has been created, the study analyst can run simulations on it.

Scenario Builder: A tool used by the study analyst to deploy organizations in a scenario and run simulations.

Scenario Builder GUI: This provides a means of creating libraries of OPFACs and organization, importing from these libraries, and importing IERs from the IER database.

Simulation domain: This consists of the Simulation Engine and the Scenario Conversion Module.

Simulation Engine: The COTS OPNET Modeler tool. It takes the scenario representation produced by the Scenario Conversion Module, processes the simulation events, and provides the output to the Results Analyzer.

Unified Modeling Language: UML is an industry standard set of graphical notations to describe a system from an object-oriented approach. Diagrams include a set of static notations (class diagrams and use case diagrams) and a set of dynamic notations (state diagrams and sequence diagrams). UML does not require a specific design process and does not require implementation

with any specific object-oriented languages or tools. The state diagrams, for example, are consistent with OPNET Modeler's process model notation.

APPENDIX C: ENUMERATED VALUES

The enumerated data types in Table C-1 are provided in NETWARS as public attribute definitions. This provides a mechanism for sharing any changes (additions) to enumerated values that are used as attributes.

Table C-1: Attributes for Enumerated Data Types

Attribute	Values
equipment_type	Computer Radio Phone JTIDS Switch, router Satellite LOS radio Promina Satellite terminal OE CellXpress Encryptor Multiplexer Patch Panel Layer 1Radio Layer 1 Satellite Accelerator Generic Device VTC Terminal Media Gateway
traffic type	Voice Data VTC
transport protocol	TCP UDP AAL5 None (if no transport protocol is used) Other (user specified)

APPENDIX D: PACKET FORMATS

Table D-1 lists the packet formats used by the NETWARS Standard models. These packet formats may be required for interoperability with the NETWARS Standard models and protocols.

In order to examine the contents of the packet format, you will need to open the *.pk.m files using OPNET Modeler. It is easy to perform a search on the directory structure for NETWARS to locate the Packet Format files. However, if you open these files up using a text editor like Wordpad or Notepad, you will quickly discover that they contain binary information that will make it difficult to read.

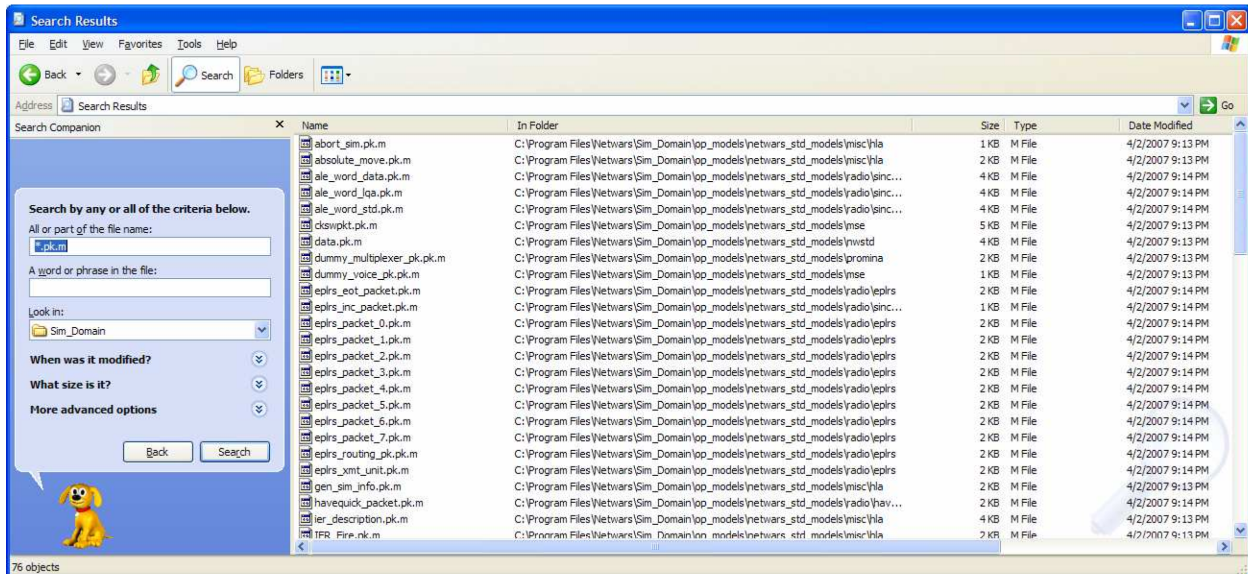


Figure D-1: Packet Format Files

A better way to look at these files is through OPNET Modeler. Select File and then Open to get to the Open Dialog box. Set the "Files of type:" field to "Packet Format Files (*.pk.m)." The example below shows the Open Dialog box for the NETWARS folder of "eplrs".

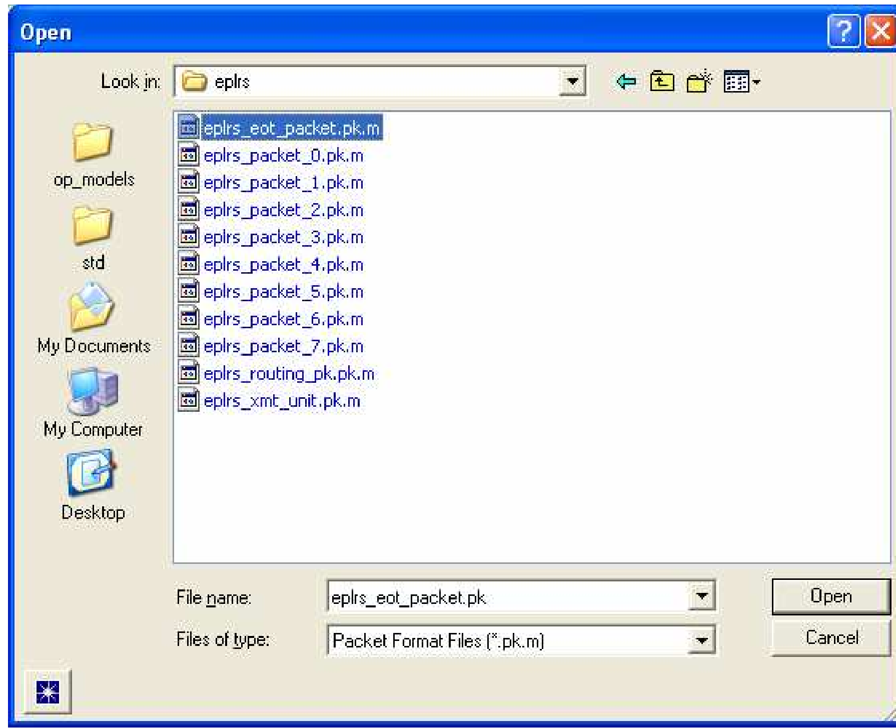


Figure D-2: Open Packet File

Select "eplrs_packet_0." A dialog box is displayed with the format.

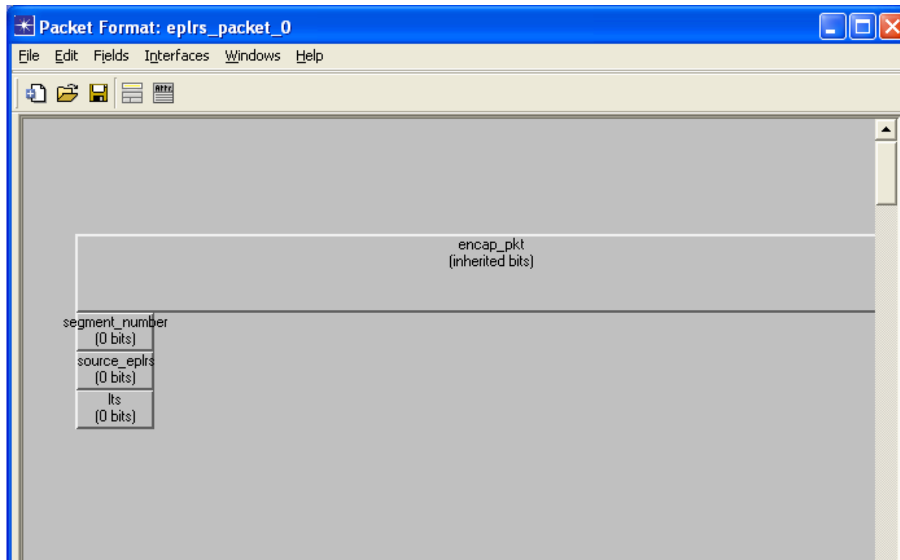


Figure D-3: Packet Format Layout

By right clicking on the field and selecting "Edit Attributes," you can list out the attributes for the particular field, as is the case below for "encap_pkt."

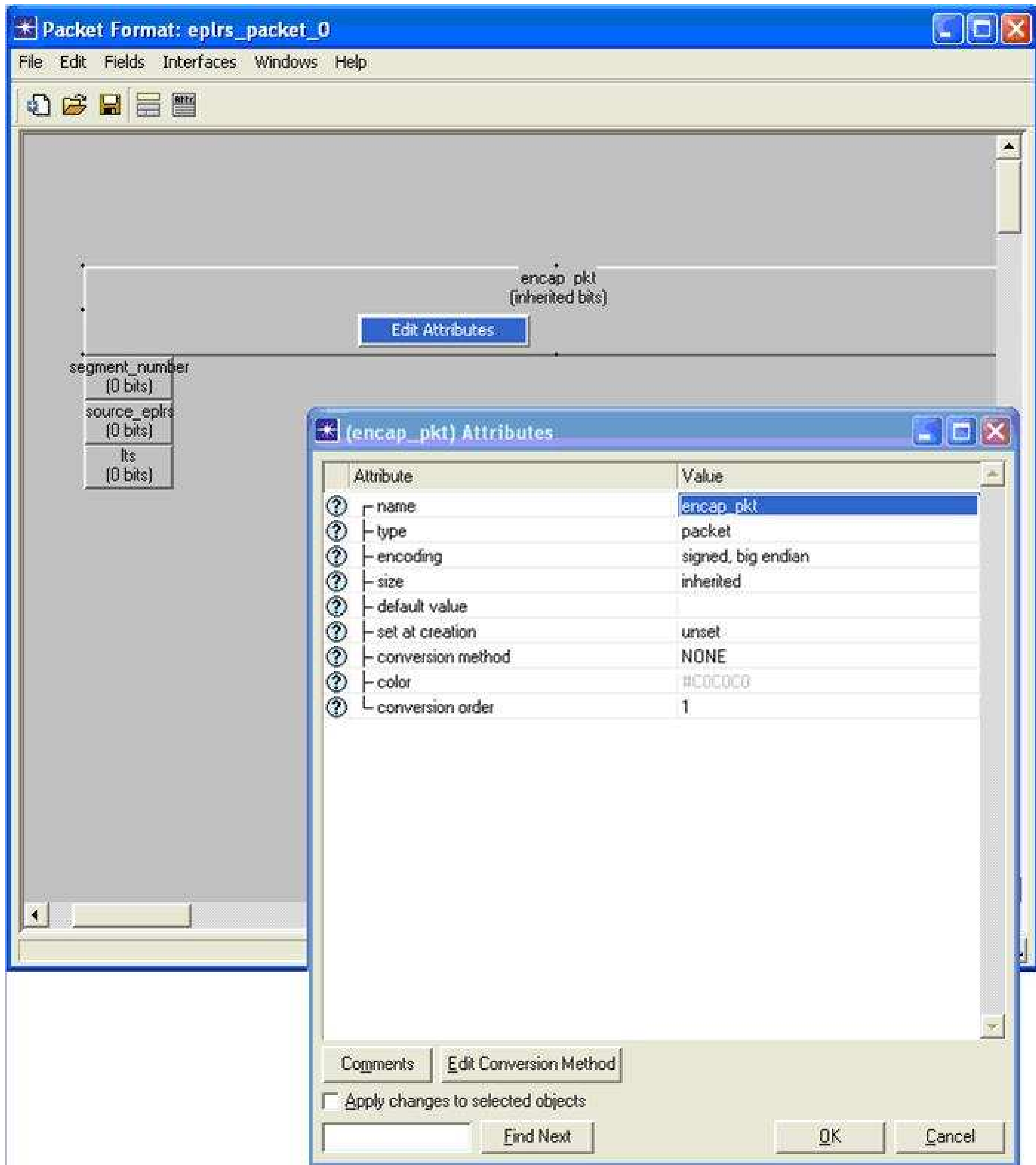


Figure D-4: Packet Format Attribute Editing

Appendix E: Interfaces and Packet Formats contains a list of MAC technologies currently supported by OPNET Modeler and the corresponding packet formats. Use Appendix E to supplement the information found here in Appendix D.

Table D-1: Packet Formats

Packet Format	Description
abort_sim	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
absolute_move	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
ale_word_data	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
ale_word_lqa	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
ale_word_std	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
ckswpkt	Netwars\Sim_Domain\op_models\netwars_std_models\mse
data	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
dummy_multiplexer_pk	Netwars\Sim_Domain\op_models\netwars_std_models\promina
dummy_voice_pk	Netwars\Sim_Domain\op_models\netwars_std_models\mse
eplrs_eot_packet	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_inc_packet	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
eplrs_packet_0	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_1	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_2	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_3	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_4	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_5	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_6	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_packet_7	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_routing_pk	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_xmt_unit	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
gen_sim_info	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
havequick_packet	Netwars\Sim_Domain\op_models\netwars_std_models\radio\havequick
ier_description	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
IER_Fire	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
ier_info	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
ip_dgram_v4	Netwars\Sim_Domain\op_models\modified_opnet_std_models\ip
isdn_packet	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
JREAP_application_free_text_encoded	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JREAP_application_free_text_uncoded	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JREAP_application_header	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JREAP_application_J_series	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JREAP_full_stack_message_group	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JREAP_full_stack_transmission_block	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JREAP_mgmt_message	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
JTIDS_packed_frame	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
jtids_pk	Netwars\Sim_Domain\op_models\netwars_std_models\radio\jtids
KG194_19	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG84_7	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
layer_1_circuit_data	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
Link_16_free_text_message	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
Link_16_J_series_message	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
link_info	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
link11_data	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
MIL_STD_1553_packet	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
mop_data	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
mop_info	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
move_opfac_by	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
move_opfac_by_bearing	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
move_opfac_to	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
mse_data_packet	Netwars\Sim_Domain\op_models\netwars_std_models\mse
mse_hello_packet	Netwars\Sim_Domain\op_models\netwars_std_models\mse
new_ier_description	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
NIMA	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
nw_voatm_hello_pkt	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway
nw_voip_hello_pkt	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway

opfac_damage	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
opfac_init	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
opfac_repair	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
phone_switch	Netwars\Sim_Domain\op_models\netwars_std_models\mse
positional_move	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
pro_cx_pk	Netwars\Sim_Domain\op_models\netwars_std_models\promina
pro_hello_pk	Netwars\Sim_Domain\op_models\netwars_std_models\promina
pro_wan_pk	Netwars\Sim_Domain\op_models\netwars_std_models\promina
radio_packet	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
satellite_pk	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sincgars_inc_packet	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
SRAP_application	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
SRAP_application_v2	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
trigger_ier	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
trigger_new_ier	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
tssp_frame	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Sat_Packet	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USMTF	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
vector_move	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
vtc_packet	Netwars\Sim_Domain\op_models\netwars_std_models\vtc

APPENDIX E: INTERFACES AND PACKET FORMATS

This section provides a list of MAC technologies currently supported by OPNET Modeler and the corresponding packet formats (see Table E-1). This is merely a list of OPNET Standard (COTS) MAC-level packet formats. Users can, and should, use their own packet formats for implementing other interface technologies.

Please refer to Appendix D: Packet Formats for more details regarding the Supported Packet Formats listed below.

Table E-1: Interfaces and Packet Formats

Interface Technology	Supported Packet Formats
Ethernet	ethernet_v2
ATM	ams_atm_cell
FDDI	fddi_llc_fr, fddi_mac_fr, fddi_mac_tk
SLIP (DSL, ISDN)	ip_dgram_v4
Frame Relay	frms_admin_frame, frms_frame_fmt, frms_tpal_setup_frame
Token Ring	tk_llc_fr, tk_mac_fr, tk_mac_tk
Wireless LAN	wlan_control, wlan_mac

APPENDIX F: INTERFACE CONTROL INFORMATION (ICI) FORMATS

The ICI Format Files work in a similar fashion to the Packet Format Files. In order to examine the contents of the ICI format you will need to open the *.ic.m files using OPNET Modeler. It is easy to perform a search on the directory structure for NETWARS to locate the ICI Format files. However, if you open these files up using a text editor like Wordpad or Notepad, you will quickly discover that they contain binary information that will make it difficult to read.

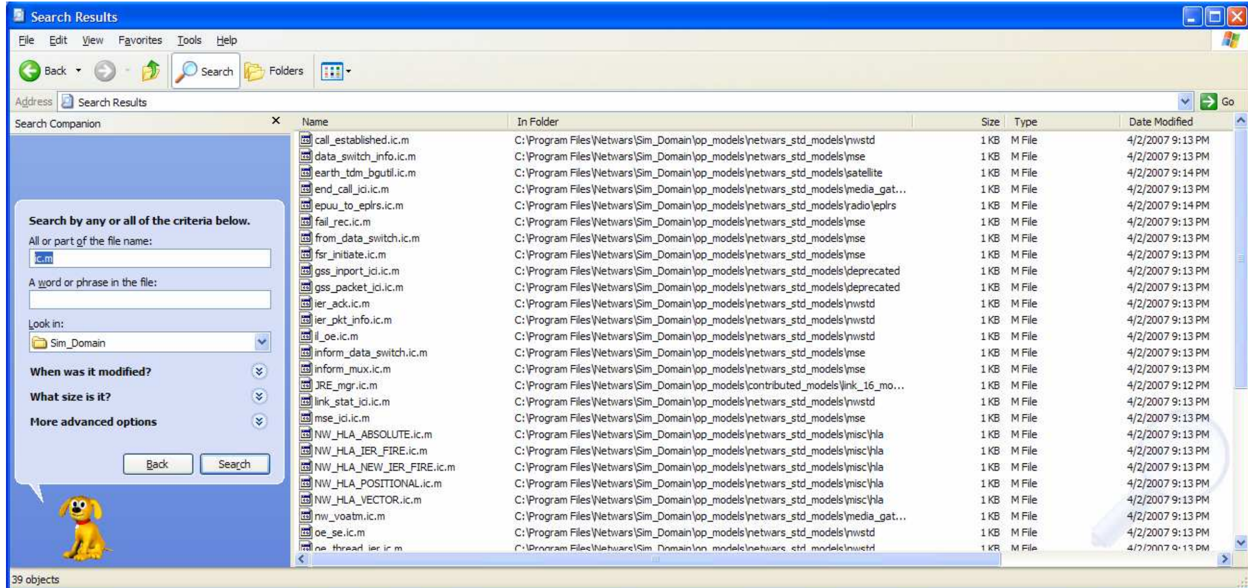


Figure F-1: ICI Format Files

A better way to look at these files is through OPNET Modeler. Select File and then Open to get to the Open Dialog box. Set the "Files of type:" field to "ICI Format Files (*.ic.m)." The example below shows the Open Dialog box for the NETWARS folder of "nwstd."

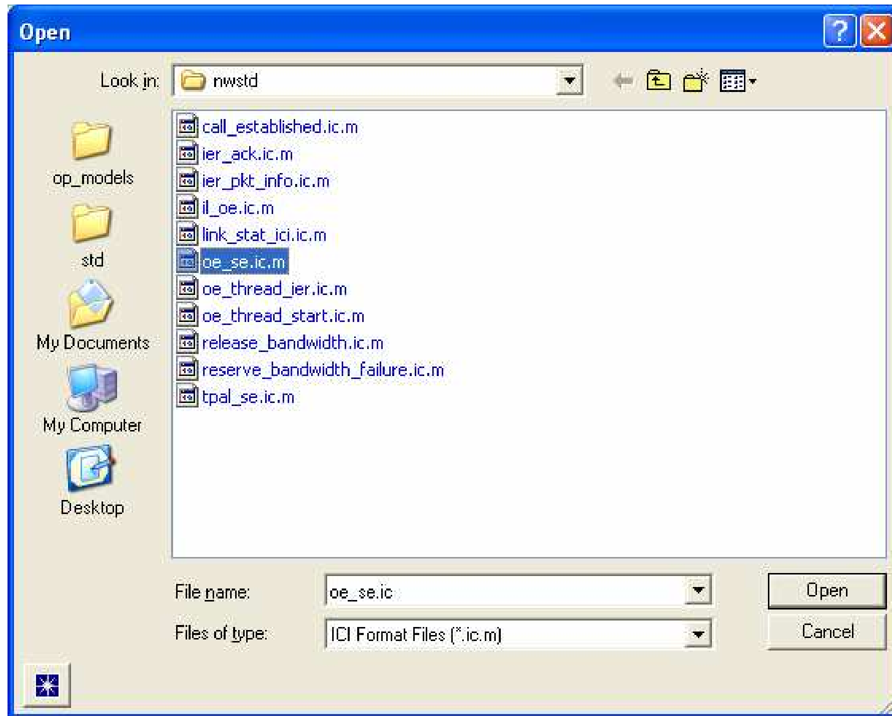


Figure F-2: Open ICI Format

Select "oe_se.ic.m" file, to display a dialog box with the ICI format Attribute Names, Type, Default Value, and Description (if any).

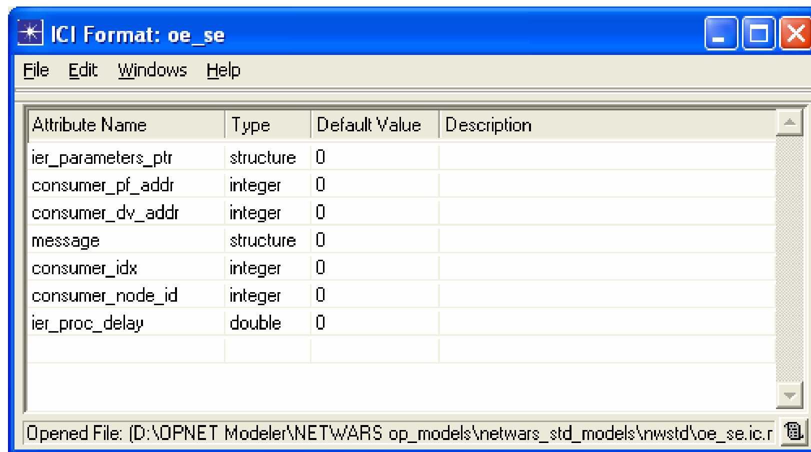


Figure F-3: ICI Format Attributes

Table F-1: Interfaces and Packet Formats

ICI Format	Location
call_established	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
data_switch_info	Netwars\Sim_Domain\op_models\netwars_std_models\mse
earth_tdm_bgutil	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
end_call_ici	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway
epuu_to_eplrs	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
fail_rec	Netwars\Sim_Domain\op_models\netwars_std_models\mse
from_data_switch	Netwars\Sim_Domain\op_models\netwars_std_models\mse
fsr_initiate	Netwars\Sim_Domain\op_models\netwars_std_models\mse
gss_inport_ici	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
gss_packet_ici	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
ier_ack	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
ier_pkt_info	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
il_oe	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
inform_data_switch	Netwars\Sim_Domain\op_models\netwars_std_models\mse
inform_mux	Netwars\Sim_Domain\op_models\netwars_std_models\mse
JRE_mgr	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
link_stat_ici	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
mse_ici	Netwars\Sim_Domain\op_models\netwars_std_models\mse
NW_HLA_ABSOLUTE	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_IER_FIRE	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_NEW_IER_FIRE	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_POSITIONAL	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_VECTOR	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
nw_voatm	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway
oe_se	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
oe_thread_ier	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
oe_thread_start	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
pro_perm_bgutil	Netwars\Sim_Domain\op_models\netwars_std_models\promina
release_bandwidth	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
reserve_bandwidth_failure	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
tpal_req	Netwars\Sim_Domain\op_models\modified_opnet_std_models\tpal
tpal_se	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
UHF_SATCOM_DAMA_Info	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Entity_Config	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Entity_Registration	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Hello	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Terminal_Rev_Info	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Token_Passing	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
voice_pkt	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars

APPENDIX G: CONSTANTS

Table G-1 outlines constants, values, and accompanying remarks.

In cases where a constant is available, great care should be used to make sure the Constant Name is used and not the Constant Value. While either would work, using the Constant Name provides for cross referencing work and simplifying changes.

Table G-1: Constants

Constant Name	Constant Value	Remarks
File Name: oe_se.h		
OE_SE_IER_SEND	0x7FFFFFE0	This is an interrupt code used by the OE to inform the SE to fire an IER.
SE_OE_IER_RECEIVED	0x7FFFFFD0	No longer used in NETWARS models. Model developer is strongly recommended to not use this any more because it will not be compatible with the latest OE models.
OE_SE_MOVED	0x7FFFFFE1	This is an interrupt code used by the OE to inform an OPFAC SE(s) about a movement.
INTER_PLATFORM_WIRE	0x7F7F7F7F	Obsolete. No longer used in NETWARS models.
File Name: oe_threads		
NWC_FIRE_IER	-444	The OE uses this interrupt code for communicating between “oe_threads” and *“oe_iers”* process models when an IER is to be fired.
NWC_OE_SE	-555	The OE uses this interrupt code for communicating between “oe_threads” and *“oe_iers”* process models when an IER is received.
NWC_INFORM_DEST_OE_FAIL	-112	This is an interrupt code used by the “se” of an IER destination end device to inform its OE of an IER failure.
NWC_INFORM_DEST_OE_RCVD	-111	This is an interrupt code used by the “se” of an IER destination end device to inform its OE of an IER reception.
NWC_INFORM_SRC_OE_RCVD	-113	This is an interrupt code used by the “se” of an IER source end device to inform its OE of an IER reception.
NWC_INFORM_SRC_OE_FAIL	-114	This is an interrupt code used by the “se” of an IER source end device to inform its OE of an IER failure.
NWC_TIME_TO_FIRE_RXN	-115	This is an interrupt code used by the OE to inform itself that it is time to fire a reaction IER for a particular thread segment.
NWC_THID_LENGTH	13	This defines the size of the thread ID of an IER, which belongs to a thread as opposed to normal traffic.

Constant Name	Constant Value	Remarks
NWC_THID_CONDITION_START	"Start"	This is a string value used by the OE to distinguish the condition of a thread segment from the segments that have conditions as other segments.
File name: netwars_support.hpp		
TPAL_SE_APP_SEND	1111	This is an interrupt code used by TPAL to inform the SE to fire an application call.

Typed Files

The *typed file* attribute is used to specify file names for intrinsic file types recognized inside the OPNET environment. Table G-2 lists typed file functions.

Table G-2: Typed File Attribute

File Suffix	File Function
.trj	trajectory for a mobile node or subnet
.orb	orbit for a satellite node
.pr.m	process model for a module
.nd.m	node model
.nd.d	derived node model
.pk.m	packet format
.ic.m	ICI format
.lk.m	link model
.lk.d	derived link model
.nt.m	Network model file. The scenarios, organizations, and OPFACs created by the study analyst using the Scenario Builder GUI are stored on disk as .nt.m files.
.gdf	Generic data file
.ex.o	External object file (created from .ex.c or .ex.cpp)

APPENDIX H: OTHER FILE FORMATS

Table H-1 lists other file formats and functions.

Table H-1: Other File Formats

File Suffix	File Function
.pdef	Platform definition file. Provides the device specification for an OPFAC. This is a text file representation of an OPFAC, which is used by the NETWARS Scenario Builder.
.sdf	Simulation description file
.ex.c	ANSI C external code file
.ex.cpp	ANSI CPP external code file
.h	C/CPP header file
.xsd	XML Schema Document
.xml	XML data file

APPENDIX I: MEASURES OF PERFORMANCE IN NETWARS

Table I-1 lists the measures of performance reported by OE in a NETWARS scenario. All MOPs are reported for the source OE. These statistics are reported in the OV format and can be collected locally, globally, or both.

Table I-1: MOPs Reported by OE

No.	Statistics Name	Method of Calculation	Scope
1	Call Completion Rate	Percentage of voice IERs sent that were successfully completed	Local and Global
2	Grade of Service	Percentage of IERs received at the destination within the perishability duration	Local and Global
3	IERs Sent Count	Number of IERs sent	Local and Global
4	IERs Received Count	Number of IERs received	Local and Global
5	Message Completion Rate	Percentage of data IERs sent that were successfully received	Local and Global
6	Message Error Rate	Percentage of the data IERs sent that failed	Local and Global
7	Perishability for the rcvd IERs	A cumulative count of IERs that are received for which the delay (IER received time—IER start time) is greater than the IER perishability duration	Local and Global
8	Speed of Service (in sec) for the rcvd IERs	The delay (IER received time—IER start time) value collected for each IER	Local and Global
9	End-to-End Delay	The latency (IER received time—IER sent time) value for each IER	Local and Global
10	Connection Latency	The time difference between the IER sent time and IER start time (or Speed of Service—End-to-End Delay)	Local and Global
11	Blocking Probability	The ratio of the IERs that were blocked at least once to the number that were sent	Local and Global
12	Number of Blocks for each IER Sent	Number of times an IER is blocked	Local and Global

Table I-2 details the various statistics groups collected for the statistics listed above. The Local scope means that this statistic is relevant only to the particular OE, whereas the Global scope means that the simulation (DES) writes this statistic for the complete network as opposed to an individual network entity.

Table I-2: Statistics Groups

No.	Statistics Name	Statistics Groups	Scope
1	IER Sent	IER Statistics	Local
		Total Traffic Sent	Global
		Total Routine Traffic	Global
		Total Priority Traffic	Global
		Total Immediate Traffic	Global
		Total Flash Traffic	Global
		Total Flash Override	Global
		Total Data Traffic	Global
		Total Voice Traffic	Global

Note: These statistics are also written per IER basis, giving a complete analysis for individual IERs in addition to the groups/categories discussed above.

Device-Level MOPs

The ability to collect device-level MOPs in NETWARS allows the model developer and user to collect any OPNET node-level statistic in a NETWARS simulation. Any statistic promoted to the node level will appear when the user chooses statistics on an OPFAC in the Scenario Builder editor. These statistics are written out to an OV file by OPNET simulation kernel, which are then converted to the corresponding VEC files during simulation post-processing. The results can subsequently be viewed using the Results Analyzer in NETWARS.

All the networking and end devices support device-level MOPs.

Device-level MOPs include protocol (ATM, IP, Ethernet, TCP, OSPF, IGMP, EIGRP, BGP)-specific statistics such as IP.Traffic Sent (packets/sec), IP.Traffic Received (packets/sec), TCP.Active Connection Count, and OSPF.Traffic Sent (packets/sec) and low-level statistics such as transmitter throughput and queuing delay. For models that support standard voice and video applications over circuit switch, the device-level MOPs should include “Application Calls Generated” and “Application Calls Succeeded” statistics. There are a large number of other statistics, including the custom statistics that can also be collected.

MOPs for Links

The following MOPs are recorded for links in NETWARS:

- Voice throughput
- Data throughput (recorded in both forward and reverse directions separately for wireline links)
- Link utilization (recorded in both forward and reverse directions separately for wireline links)

- Channel utilization.

MOPs for Broadcast Radio Networks

The following MOP is recorded for radio broadcast network:

- Broadcast network utilization

Table I-3 lists the modules that write the OV-based statistics (please refer to the notes above on Generic Statistics for other nodes that can be set up to collect statistics).

Table I-3: Modules That Write OV Statistics

Statistic	Component	Module
Voice Throughput (in bits/sec.)	Layer 2 Networking Device connected to the link	circuit_switch
Channel Utilization (percent)	Layer 2 Networking Device connected to the link	circuit_switch
Broadcast Network Utilization	Radio Device	mac

Note that a link probe set up on the link for wireline links records the data_throughput and utilization statistics for the External link. Wireless point-to-point links must record this statistic themselves.

APPENDIX J: NODE MODEL DOCUMENTATION

A node model, such as end-system devices, networking devices, and OE and Utility Nodes, is documented by providing the following information in the Comments section of the Node Interfaces option in the Node Editor.

General Description of the Device

For an end-system device, the functions and its security classification are documented in this section.

For networking equipment, the functions of the networking equipment are documented in this section.

Notes to the Military Analyst

This section includes two- to three-sentence descriptions on the usage of the device itself. This will also include any special behavior or exceptions that this device model may have.

Notes to the Model Developer

This section documents the technical details that may be of interest to a model developer. Technical details to be covered in the specific sections below should not be reiterated here.

Last Edit

Version Number, Date, Author

Supported Traffic Types

Specifies the types of traffic the networking equipment handles. The traffic type can be voice, data, or both.

Supported Protocols

The list of protocols supported by this networking device.

Interface Specification

Table J-1 contains sample data. The Interface # column specifies the numeric index of the interface. The Interface Type column specifies the type of interface, such as Ethernet, ATM, or FR. The Number of Channels column specifies the number of channels supported by this interface. The Data Rate and Packet Formats columns list the data rate and packet formats supported by the individual channels on this interface.

For every interface, there are as many rows under the Data Rate and Packet Formats columns as there are number of channels in that interface.

Table J-1: Wired Interface Specifications

Interface #	Interface Type	Number of Channels	Data Rate (bps)	Packet Formats
0	ATM	1	155,520,000	ams_atm_cell
1	ATM	1	155,520,000	ams_atm_cell

The example networking equipment in Table J-1 has two ATM interfaces, each with one channel and operating at a data rate of 155.52 Mbps. The interface supports packets of type ams_atm_cell.

For radio devices the interfaces are documented differently, as shown in Table J-2.

Table J-2: Radio Device Interface Specifications

Intf #	Modulation	Number of Channels	Data Rate	Packet Formats	Minimum Frequency	Bandwidth	Spreading Code	Power
0	Bpsk	2	1,024	wlan_mac, wlan_control	30 MHz	10 KHz	disabled	100W
			2,048	wlan_mac, wlan_control	30 MHz	10 KHz	disabled	100W

The table specifies sample data for a transmitter. This transmitter has two channels, one with a data rate of 1Mbps and the other with a data rate of 2 Mbps. Both channels support packet formats of type wlan_mac and wlan_control. The channels have a minimum frequency of 30 MHz with a bandwidth of 10 KHz and transmitting power of 100 W.

Process Models

All the process models that are invoked within the context of this node are documented in the following format. Table J-3 contains sample data.

Table J-3: Process Models

Name	Location	Description
dnvt_se	dnvt	Generates the various circuit-switch signaling packets in response to VOICE IERs.

The Name column refers to the name of the process model; the Location column to the node model within which the process model resides or is invoked. A brief description of what this process model does is provided in the Description column.

External Files Needed

All external files (header files, C files) needed by the process models in this node are documented in this section. Table J-4 contains sample data.

Table J-4: External Files Needed

Name of Process Model	List of Files Used
se_computer	netwars_support.ex.c, netwars_nato.ex.c
Ip	opnet.h, ip_addr_v4.h, ip_auto_address.ex.c

Handling Failure/Recovery

This section documents which modules in this node handle failure/recovery interrupts explicitly and how the interrupts are handled.

Pipeline Stages Used (Radio/Satellite Only)

This section documents the transceiver pipeline stages for radio/satellite devices. This section is not required for wired devices.

Orbit Specification (Satellite Only)

This section documents the orbit file used by the satellite device.

Comments

This section must be used to document any additional requirements or restrictions in using this device.

Full Edit History

Version Number, Date, Author

External Documentation

Author, Date, Title, Optional Comments

APPENDIX K: MODEL NAMING CONVENTIONS

The following is a proposed naming convention to promote clarity and reduce the chances of naming conflicts. The naming convention for NETWARS uses the communications system name as the base prefix. For example, all MSE models and related files should begin their names *mse_*. This name should be unique and distinct from existing NETWARS Standard models:

- **Node Models:** Node models should use a two-part name consisting of the communications prefix and a device type separated by underscores. If the same base model will be used for multiple derived device models, a generic function type should replace the device type.
- **Derived Node Models:** If a generic base model was developed to allow multiple specific devices to be modeled with the same model, it should be named using the above standard, that is, prefix followed by device type (replacing the generic function).
- **Process Models:** The process model should be named using the convention of the prefix of device name or device classification followed by the process function, all separated by underscores. Some of the process models perform a generic function that is common to more than one device. These process models can be named starting with a prefix signifying their technology, followed again by their function. Some of the typical examples of process model naming are discussed below:
  - pro_portmap_utility: Here the *pro* part signifies the category of the device (Promina) and *portmap_utility* signifies its function of handling port map configurations.
  - ams_atm_call_control: Here the *ams_atm* signifies the ATM technology, whereas the *call_control* signifies the ATM call control functions performed by the process model.
- **External Files:** External files are named with the prefix followed by the device (or function), if applicable, followed by descriptive name, terminated with the extension *.c* or *.cpp*. For example: netwars_satellite_support.ex.c
- **Header Files:** Header files are used to declare externally callable functions, shared type definitions, defines, and simulation-wide global variables. Those header files declaring functions should use the same file name as the external (C/C++) file but with extension *.h*. If the header file does not contain declarations of externally callable functions, it should be given a name descriptive of the communications system in which it is used, optionally a function of that communications system and the extension *.h*. For example: netwars_stat_support.h
- **Link Models:** Link models should be named using the protocol and, optionally, the link speed. For example: wire_ptp
- **Derived Link Models:** Derived link models should be named using the same convention as link models.
- **Transmitters and Receivers:** The transmitter and the receiver should be named with a substring "tx_*index*" and "rx_*index*" included in the name. The *index* should start from an integer value of 0 and be numbered sequentially. Examples of transmitter names

include tx_1, inc_tx_2, and atm_tx_3_0. Receivers could be named as rx_1, inc_rx_2, atm_tx_3_0, and so on. Note that the name should not include any more tx or rx substrings. Names that are not acceptable, for example, are mtx_tx_0 and rtx_tx_5.

- **Externally Callable Functions:** Externally callable functions should be named using an abbreviation for the external file in which it resides followed by a short phrase describing the function.

For example: nw_sdf_sup_init() function in netwars_sdf_support.ex.c file

APPENDIX L: NETWARS SIMULATION API AND HELPER FUNCTIONS

Table L-1 lists an example of an API and some of the functions with input parameters that are available in NETWARS. The following discussion will demonstrate how to locate the APIs and determine what functions are available to be called within that particular API, as well as provide an example of how the Table L-1 was started. Finally, Table L-2 will provide a list of all the APIs available and their location within the directory structure.

The APIs are the External Files, which end in either ".ex.c" for C source code files or ".ex.cpp" for C++ source code files work in a similar fashion to the Packet Format Files. Unlike the files ending in ".m" (e.g., Packet Formats and ICI Formats), these are text files and can be viewed using a text editor such as Notepad or Wordpad. The APIs can easily be found by searching the NETWARS model files looking for those files that end in ".ex.c*", which would include both C and C++ APIs.

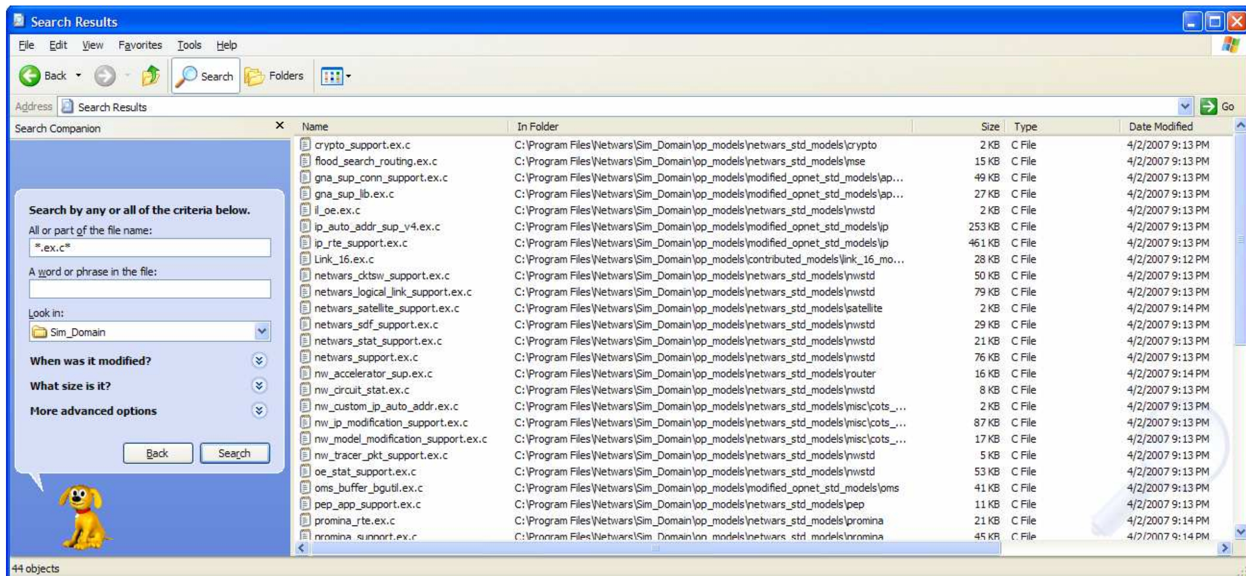


Figure L-1: API Files

A preferred method of looking at these files is through OPNET Modeler, since its editor will color-code portions of code and make certain things stand out, in particular, the API function names will be of interest. Select File and then Open to get the Open Dialog box. Set the "Files of type:" field to either "External Source (C code) Files (*.ex.c)" or "External Source (C++ code) Files (*.ex.cpp)." Currently, none of the NETWARS APIs are coded in C++, but there are OPNET APIs that are coded in C++. Specifically with version 12.0.A, 62 C++ files were located out of 414 total External Source Files. The example below shows the Open Dialog box for the NETWARS folder of "nwstd."

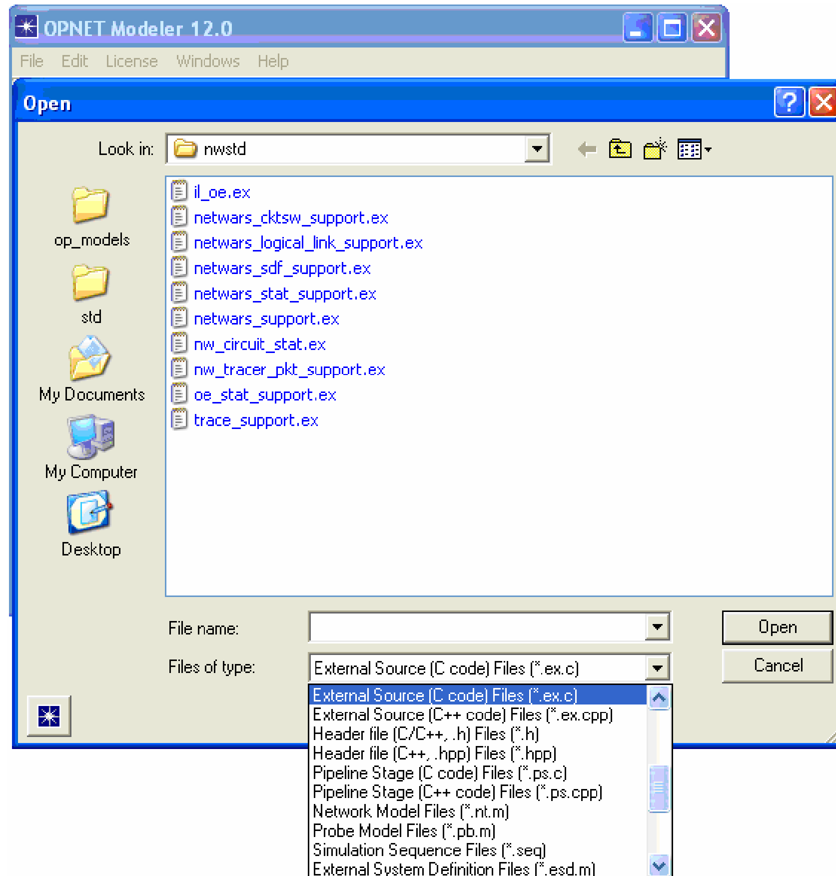


Figure L-2: Open API File

Select "oe_stat_support.ex.c" file to have the OPNET Modeler Editor display the file, as seen below.

External Source (C code): oe_stat_support

File Edit Options Compile Windows Help

```
/*******************************************/
/*      Copyright (c) 1987 - 2002          */
/*       by OPNET Technologies, Inc.       */
/*        (A Delaware Corporation)         */
/*     7255 Woodmont Av., Suite 250        */
/*      Bethesda, MD 20814, U.S.A.         */
/*        All Rights Reserved.             */
/*******************************************/

/* oe_stat_support.ex.c - Support functions for OE (IER and Thread) statistics reporti
#include "oe_env.h"
#include "oe_se.h"
#include "oe_stat_support.h"

void
nw_oe_sent_stat_write (const IER_Parameters *ierparms,
                            Stathandle* opfac_ier_stathandle,
                            double* opfac_ier_statistics_values,
                            PrgT_String_Hash_Table* ind_ier_stat_hash_table_ptr,
                            PrgT_String_Hash_Table* ind_ier_stat_value_hash_table_ptr,
                            const char* action_str
                            )
    {
    /* Purpose: Function meant to update the sent statistics for IER */
    FIN (nw_oe_sent_stat_write (ierparms));

    /* First do OPFAC level IER stat update */
    /* and alongside do the indv. IER stat  */
    /* update as well.                      */
    nw_oe_opfac_and_indv_ier_sent_stat_write (ierparms, opfac_ier_stathandle,
                                        opfac_ier_statistics_values,
                                        ind_ier_stat_hash_table_ptr,
                                        ind_ier_stat_value_hash_table_ptr,
                                        action_str);

    FOUT;
    }

void
nw_oe_opfac_and_indv_ier_sent_stat_write (const IER_Parameters* ierparms,
                                Stathandle* opfac_ier_stathandle,
                                double* opfac_ier_statistics_values,
                                PrgT_String_Hash_Table* ind_ier_stat_hash_table_ptr,
                                PrgT_String_Hash_Table* ind_ier_stat_value_hash_table_
                                const char* action_str)
    {
    /* Purpose: Function meant to update the global, opfac and individual IER stats */
    Stathandle*     indv_ier_stat_handle_ptr = OPC_NIL;
    double*         ier_stat_values_ptr = OPC_NIL;

    IerT_Global_Stats*  ier_traffic_type;
    IerT_Global_Stats*  ier_priority;

    double*             global_traffic_type_stat_values;
    double*             global_traffic_prio_stat_values;

    Boolean             ierid_found  = OPC_TRUE;

    FIN (nw_oe_opfac_and_indv_ier_sent_stat_write (...))

    /* Get the entries in the hash table */
    indv_ier_stat_handle_ptr = (Stathandle*) prg_string_hash_table_item_get (ind_ier_s
                                                                    ierpar
    ier_stat_values_ptr = (double*) prg_string_hash_table_item_get (ind_ier_stat_value
                                                                ierparms->
```

Opened File: (D:\OPNET Modeler\NETWARS op_models\netwars_std_models\nwstd\oe_stat_support.ex. Line: 1

Figure L-3: oe_stat_support API

The functions in the oe_stat_support API that can be called are in the source code. The first is "nw_oe_sent_stat_write." While there may not be a comment to describe all the functions

within an API, this particular one has a purpose that is straight to the point; "Function meant to update the sent statistics for IER". If you need to update the sent statistics for an IER, this would be the function to call and the API to include. The following table is an example of how a list of API Functions can be developed using this information.

Table L-1: Example of API Function Table

API	API Functions	Purpose	Parameters
oe_stat_support	nw_oe_sent_stat_write	Update the sent statistics for IER	ierparms opfac_ier_stathandle opfac_ier_statistics_values ind_ier_stat_hash_table_ptr ind_ier_stat_value_hash_table_ptr action_str
	nw_oe_opfac_and_indv_ier_sent_stat_write	Update the Global, OPFAC and Individual IER stats	ierparms opfac_ier_stathandle opfac_ier_statistics_values ind_ier_stat_hash_table_ptr ind_ier_stat_value_hash_table_ptr action_str
	(continued)	(continued)	(continued)

There are currently 44 NETWARS APIs, and that list is expected to grow. It would be difficult at best to develop Table L-1 for all of the functions within these APIs, but included below in Table L-2 is a list of the APIs and their locations.

Table L-2: NETWARS APIs and Locations

API List	Location
crypto_support	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
flood_search_routing	Netwars\Sim_Domain\op_models\netwars_std_models\mse
gna_sup_conn_support	Netwars\Sim_Domain\op_models\modified_opnet_std_models\applications
gna_sup_lib	Netwars\Sim_Domain\op_models\modified_opnet_std_models\applications
il_oe	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
ip_auto_addr_sup_v4	Netwars\Sim_Domain\op_models\modified_opnet_std_models\ip
ip_rte_support	Netwars\Sim_Domain\op_models\modified_opnet_std_models\ip
Link_16	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
netwars_cktsw_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
netwars_logical_link_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
netwars_satellite_support	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
netwars_sdf_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
netwars_stat_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
netwars_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
nw_accelerator_sup	Netwars\Sim_Domain\op_models\netwars_std_models\router
nw_circuit_stat	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
nw_custom_ip_auto_addr	Netwars\Sim_Domain\op_models\netwars_std_models\misc\cots_support
nw_ip_modification_support	Netwars\Sim_Domain\op_models\netwars_std_models\misc\cots_support
nw_model_modification_support	Netwars\Sim_Domain\op_models\netwars_std_models\misc\cots_support
nw_tracer_pkt_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
oe_stat_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
oms_buffer_bgutil	Netwars\Sim_Domain\op_models\modified_opnet_std_models\oms
pep_app_support	Netwars\Sim_Domain\op_models\netwars_std_models\pep
promina_rte	Netwars\Sim_Domain\op_models\netwars_std_models\promina

promina_support	Netwars\Sim_Domain\op_models\netwars_std_models\promina
promina_support_alt	Netwars\Sim_Domain\op_models\netwars_std_models\promina
promina_topo	Netwars\Sim_Domain\op_models\netwars_std_models\promina
promina_voice_support	Netwars\Sim_Domain\op_models\netwars_std_models\promina
rtp_support	Netwars\Sim_Domain\op_models\modified_opnet_std_models\rtp
sincgars\radio_support	Netwars\Sim_Domain\op_models\netwars_std_models\radio
tcp_api	Netwars\Sim_Domain\op_models\modified_opnet_std_models\tcp
tirem_support	Netwars\Sim_Domain\op_models\netwars_std_models\misc\tirem
tpal_api	Netwars\Sim_Domain\op_models\modified_opnet_std_models\tpal
tpal_app_support	Netwars\Sim_Domain\op_models\modified_opnet_std_models\applications
trace_support	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
UHF_SATCOM_CPS_Entity	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_CPS_ServicePlan_Parser	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_CPS_TextManipulation	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Noise_Area	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Orderwires	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Platform	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Platform_Utilization	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Port_Map	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USN_ckt_supp	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models

Note: The NETWARS XML Schema defines the official input and output file for NETWARS Scenario Builder. The SDF file format is now obsolete. A traffic XML file has replaced it, which contains the IER and threaded IER information for the scenario.

APPENDIX M: ATTRIBUTE TYPE DEFINITIONS

This appendix describes the various attribute types used in NETWARS. For more information, refer to OPNET Modeler Online documentation, Modeling Concepts Manual, "Modeling Framework" chapter, "Fram.3.3, Attributes" section.

Toggle

When a variable takes Boolean values such as On/Off or Included/Not Included, it is defined as a Toggle variable. An example of a Toggle variable is *availability_status* of a node, which is "1" to indicate that it is available for communication or "0" to indicate that it is not available.

Integer

When a variable takes whole-number values, it is defined as an integer variable. An example of an integer variable is *max_active_calls* for a phone, which cannot be defined in fractions of the number of supported calls.

Double

When a variable needs to represent a precise numerical quantity, it is defined as a double variable. An example of a double variable is *x position* of a node, which could take a value such as "38.324."

String

When a variable is used to hold a set of characters, it is defined as a string variable. An example of a string variable is the *name* attribute of a node.

Enumerated

When a variable takes only a set of pre-defined values, it is represented as an enumerated variable. The value for an enumerated variable is represented as a string during specification and as an integer during simulation.

An example of an enumerated value is the *classification* attribute of a node. This variable takes a certain number of pre-defined values such as "classified," "unclassified," or "secret." These are typically loaded as public attribute definition files and can be shared across models, for example, the classification.ad.m file.

The NETWARS Standard enumerated types have been defined as public attributes, and these are defined in Appendix C.

Compound

When a variable cannot be represented by one of the simple data types described above, it is represented by a compound data type. A compound data type is a collection of simple data types and other complex data types. A compound data type can have arbitrary levels of nesting.

An example of a compound variable is the *channel* attribute of a transmitter module. The *channel* attribute is a combination of two simple data types—an integer called *data rate* and an enumerated field called *packet format*.

Typed file

This is a character string that represents the name of a file. A typed file could be a trajectory file for a mobile node, an orbit file for a satellite node, or any of the other supported file formats. For a full listing of the supported typed files, refer to Appendix G.

Structure

This is similar to the *compound* attribute type. While the *compound* attribute type is used in the node model attributes, the *structure* attribute is used in packet format attributes.

Information

The *information* attribute type is used in packet fields. These fields cannot contain any actual value, and they are used only as padding for the packets, so that the packet can have a certain number of bits.

Objid

An object ID is used to uniquely identify a simulation object. Nodes, modules inside of nodes, and compound attributes are all examples of simulation objects. The data type *Objid* is used to declare these identifiers. This value is not modifiable.

APPENDIX N: EXAMPLES OF NETWARS MODELS

This appendix will go through an example of locating a NETWARS Model and the information relevant to the model. Table N-1 provides a list of all the NETWARS Models in alphabetic order to use as a reference in locating a specific model.

Node Models end in ".nd.m." While it is easy to get a list of the files in Microsoft Windows Explorer, the files contain binary data and will not be able to be opened and easily read with a text editor, e.g., Notepad or Wordpad. The Node Models can easily be found by searching the NETWARS model files looking for those files that end in ".nd.m," as shown in Figure N-1

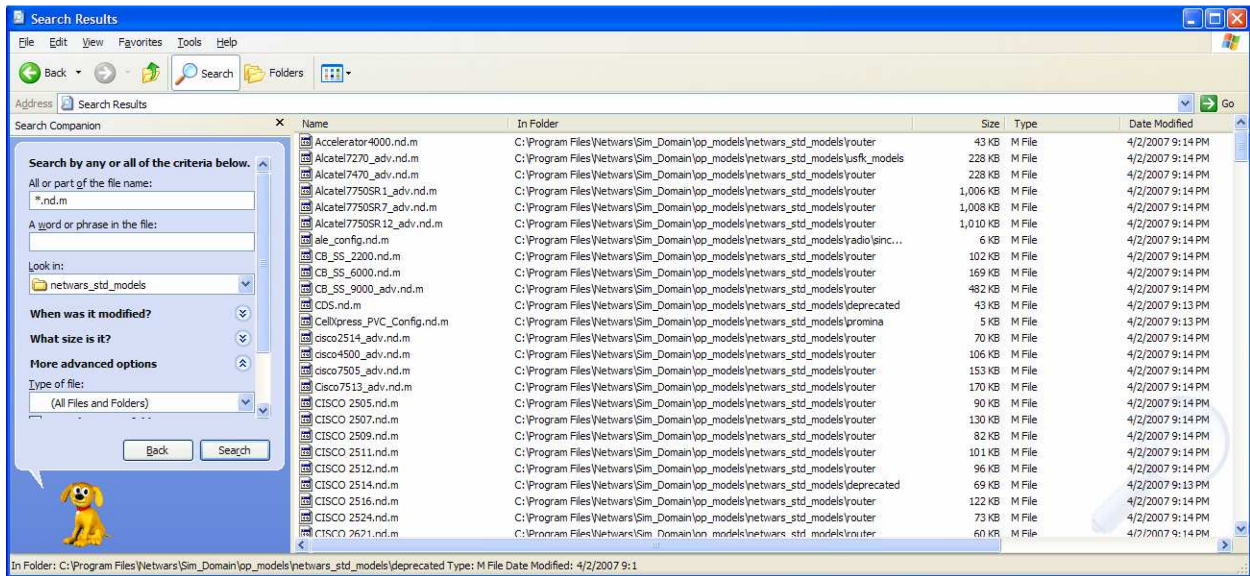


Figure N-1: List of Node Models

These files can be examined through OPNET Modeler by opening them as shown in Figure N-2. Once the Node Model is displayed, then you can examine items such as Model Attributes, Node Interfaces, Node Statistics, Self Description and all associated Process Models that pertain to all the modules within the NETWARS Models.

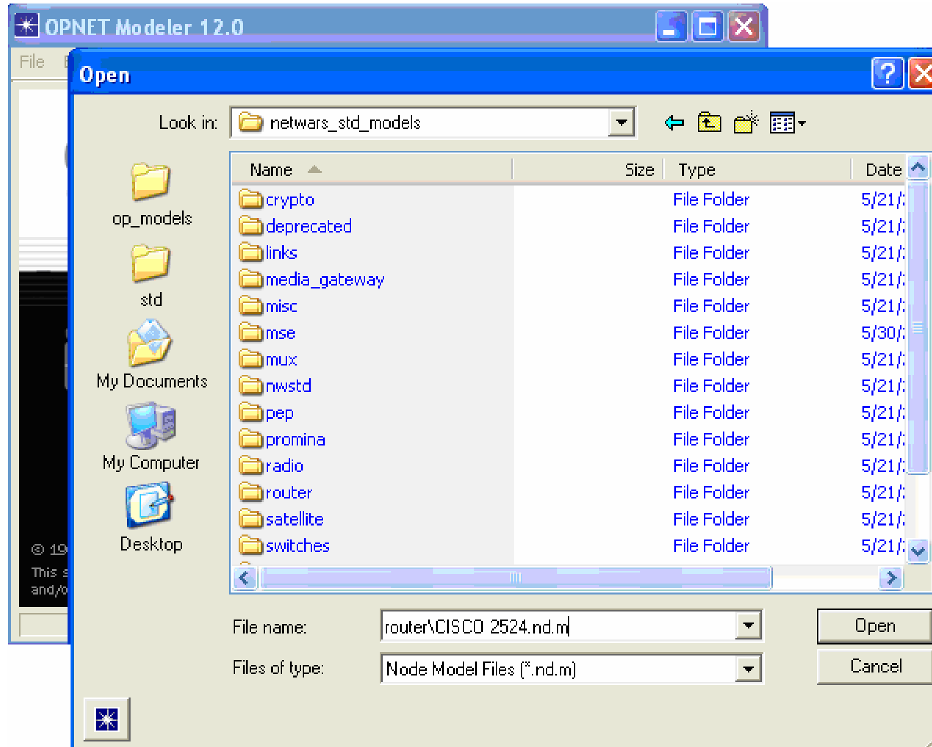


Figure N-2: Open NETWARS Model

A valuable section within the Node Interfaces is the Comments. The comments may follow a template that contains information such as;

1. Section One – General Information
   a. Model Name
   b. Communications Device Model Description
   c. Interface List
   d. Routing and Transport
   e. Supported Multi-access schemes
   f. Supported multiplexing schemes
   g. Configurable attributes
   h. Supported traffic
2. Section Two - Failure recovery support
3. Section Three - Developer notes.
   a. ICI Formats
   b. External Files
   c. Header Files
   d. Process Models
   e. Pipeline Stages

4. Section Four - Model Fidelity
5. Section Five – Military Analyst Nodes
   a. Model Usage
   b. Exceptions and Elaborations
   c. Military Analyst Comments
6. Section Six - Comments
   a. Full Edit History
   b. External Documentation
   c. References and Specifications Used

The following table is a complete list of the NETWARS Models in alphabetical order.

Table N-1: List of NETWARS Models (Alphabetic)

NETWARS Models	Location
Accelerator4000	Netwars\Sim_Domain\op_models\netwars_std_models\router
Alcatel7270_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
Alcatel7470_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Alcatel7750SR1_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Alcatel7750SR12_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Alcatel7750SR7_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
ale_config	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
AN_FCC_100_V	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_131_V_BB_Transmitter	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_131_V_NB_Transmitter	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_131_V_Receiver	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_139_V	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_USC_38_MDR	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V11	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V14	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V15	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V17	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V18	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V2	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V3	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V6	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V7	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V9	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_5_V	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V2	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V4	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V5	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V7	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V9_C	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V9_X	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_8_V1	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_8_V2	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
CA_Satellite	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
CB_SS_2200	Netwars\Sim_Domain\op_models\netwars_std_models\router
CB_SS_6000	Netwars\Sim_Domain\op_models\netwars_std_models\router
CB_SS_9000_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
CDS	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
CellXpress_PVC_Config	Netwars\Sim_Domain\op_models\netwars_std_models\promina
CISCO 2505	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2507	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2509	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2511	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2512	Netwars\Sim_Domain\op_models\netwars_std_models\router

CISCO 2514	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
CISCO 2516	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2524	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2621	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2916	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2924	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2950G 24 EI	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 2950G 24 EI_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3000	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3620	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3640	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3660	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3725_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3745	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 3745_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Cisco 3750_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 4006	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 4500-M	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 4700-M	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 7010	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 7206	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 7505	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
CISCO 7507	Netwars\Sim_Domain\op_models\netwars_std_models\router
CISCO 7513	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Cisco_LS_1010	Netwars\Sim_Domain\op_models\netwars_std_models\switches
cisco2514_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
cisco4500_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
cisco7505_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Cisco7513_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
computer_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
computer_ethernet_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
computer_TCP_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
computer_TCP_ethernet_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Definity Prologic	Netwars\Sim_Domain\op_models\netwars_std_models\mse
dnvt	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DPA	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DPM	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DSCS SLEP	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
DSCS_III_Satellite	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
DSS-1	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DSS-2	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DSS-3	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DTA	Netwars\Sim_Domain\op_models\netwars_std_models\mse
DVS-G Bridge	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
Testing_Module	Netwars\Sim_Domain\op_models\netwars_std_models\misc\dynamic_testing_softw are
EPLRS	Netwars\Sim_Domain\op_models\netwars_std_models\radio\eplrs
Falcon_II	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
FCC-100 V7	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
FCC-100 V9	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
FCC-100V9	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Firewall_2NIC	Netwars\Sim_Domain\op_models\netwars_std_models\router
Firewall_3NIC	Netwars\Sim_Domain\op_models\netwars_std_models\router
Firewall_4Slot	Netwars\Sim_Domain\op_models\netwars_std_models\router
FLBCST	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
FoundryFastIron1500Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron2402Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron400Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron4802Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron800Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron9604Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryNetIron1500Router_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models

FoundryNetIron400Router_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryNetIron800Router_adv	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
FSC-78	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
gen_sat_earth_term	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Generic ATM Switch	Netwars\Sim_Domain\op_models\netwars_std_models\router
Generic C Band Trml	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Generic Ckt Switch	Netwars\Sim_Domain\op_models\netwars_std_models\mse
Generic H 320 Bridge	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
Generic Hub	Netwars\Sim_Domain\op_models\netwars_std_models\switches
Generic IDS	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
Generic IP Data Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Generic Ku Band Trml	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Generic Layer 2 Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\switches
Generic Layer 3 Switch_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Generic MW LOS	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
Generic Router	Netwars\Sim_Domain\op_models\netwars_std_models\router
Generic Server	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
Generic Smart Mux	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Generic TDM Mux	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Generic Telephone	Netwars\Sim_Domain\op_models\netwars_std_models\mse
Generic UFO	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
Generic VTC Trml	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
generic_broadcast_satellite	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
generic_space_segment	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
GSC-39	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
GSC-52	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Harris_6010_adv	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
Harris_Megastar_155	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
havequick_rt	Netwars\Sim_Domain\op_models\netwars_std_models\radio\havequick
hf_rt	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
IDNX-20	Netwars\Sim_Domain\op_models\netwars_std_models\promina
IDNX-90	Netwars\Sim_Domain\op_models\netwars_std_models\promina
ier_loader	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
INMARSAT_B_HSD	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
INMARSAT_B_Satellite	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
Intelsat	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
IP_ATM_TACLANE	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
ISDN_MCU	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
ISDN_VTC_Trml	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
JRE Gateway	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
JRE Gateway_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
JRE_Gateway	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
jtids	Netwars\Sim_Domain\op_models\netwars_std_models\radio\jtids
JTIDS_Terminal	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
KG_194	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
KG_84	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
KG-175 ATM	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG-175 E-100	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
KG-175 IP	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG175-E_10	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG175-E10	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
KG175-E100	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG194_crypto_base	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG-235	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG-250	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KG84_crypto_base	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KIV7_crypto_base	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
KY68	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
LAN WAN IP network_adv	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
layer_1_crypto_base	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
len	Netwars\Sim_Domain\op_models\netwars_std_models\mse
Link_11	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models

Link_16_Config	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
Link_16_Host_Processor	Netwars\Sim_Domain\op_models\contributed_models\link_16_models
Live PCS-100	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
Marconi_ASX1000	Netwars\Sim_Domain\op_models\netwars_std_models\router
Marconi_ASX1200	Netwars\Sim_Domain\op_models\netwars_std_models\router
Marconi_ASX200BX	Netwars\Sim_Domain\op_models\netwars_std_models\router
Marconi_PH6000_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Marconi_PH7000_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Marconi_PH8000_adv	Netwars\Sim_Domain\op_models\netwars_std_models\router
Marconi_TNX1100	Netwars\Sim_Domain\op_models\netwars_std_models\router
MC-6000	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
media_gateway	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway
MilStar_2_Satellite	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
MMT	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
Motorola NES	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
MRC-142	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
MSE_Switch_a6_e25_sl38	Netwars\Sim_Domain\op_models\netwars_std_models\mse
multiplexer_utility	Netwars\Sim_Domain\op_models\netwars_std_models\misc\utility
mux_12inputs	Netwars\Sim_Domain\op_models\netwars_std_models\mux
mux_16inputs	Netwars\Sim_Domain\op_models\netwars_std_models\mux
mux_2inputs	Netwars\Sim_Domain\op_models\netwars_std_models\mux
mux_4inputs	Netwars\Sim_Domain\op_models\netwars_std_models\mux
mux_8inputs	Netwars\Sim_Domain\op_models\netwars_std_models\mux
mux_etssp_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
NAVMACS	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
ncs	Netwars\Sim_Domain\op_models\netwars_std_models\mse
NES	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
NIMA	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
nw_eth_switched_lan_adv	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
nw_ethernet_server	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
nw_ethernet_wkstn	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
nw_generic_device	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
nw_hla_interaction	Netwars\Sim_Domain\op_models\netwars_std_models\misc\hla
nw_jam_pulsed	Netwars\Sim_Domain\op_models\netwars_std_models\radio\jammers
nw_jam_sb	Netwars\Sim_Domain\op_models\netwars_std_models\radio\jammers
nw_jam_swept	Netwars\Sim_Domain\op_models\netwars_std_models\radio\jammers
NW_KG-194	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
NW_KG-84	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
NW_KIV-7	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
nw_multihommed_server	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
nw_multihommed_wkstn	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
nw_ppp_server	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
nw_ppp_wkstn	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
Nw_QoS_Attribute_Config	Netwars\Sim_Domain\op_models\netwars_std_models\misc\utility
Nw_Sink	Netwars\Sim_Domain\op_models\netwars_std_models\misc\utility
oe	Netwars\Sim_Domain\op_models\netwars_std_models\nwstd
Omni Switch 3WX	Netwars\Sim_Domain\op_models\netwars_std_models\switches
Omni Switch 5WX	Netwars\Sim_Domain\op_models\netwars_std_models\switches
Omni Switch 9WX	Netwars\Sim_Domain\op_models\netwars_std_models\switches
Patch_Panel_48	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
Patch_Panel_96	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
pro_cell_express_adv	Netwars\Sim_Domain\op_models\netwars_std_models\promina
pro_portmap_utility	Netwars\Sim_Domain\op_models\netwars_std_models\promina
Promina	Netwars\Sim_Domain\op_models\netwars_std_models\promina
PROMINA 200	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
PROMINA 400	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
PROMINA 800	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Promina_10l_5w_2eth_2sclx_2cx_adv	Netwars\Sim_Domain\op_models\netwars_std_models\promina
Promina-100	Netwars\Sim_Domain\op_models\netwars_std_models\promina
Promina-200	Netwars\Sim_Domain\op_models\netwars_std_models\promina
Promina-400	Netwars\Sim_Domain\op_models\netwars_std_models\promina
Promina400_e180_sl180	Netwars\Sim_Domain\op_models\netwars_std_models\promina

Promina-800	Netwars\Sim_Domain\op_models\netwars_std_models\promina
Proteon CNX 500	Netwars\Sim_Domain\op_models\netwars_std_models\router
Proteon CNX 600	Netwars\Sim_Domain\op_models\netwars_std_models\router
RadVision Bridge	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
RadVision VTC Suite	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
REDCOM HDX	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 1 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 10 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 16 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 2 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 3 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 4 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 5 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 6 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 7 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
REDCOM IGX 8 Shelf	Netwars\Sim_Domain\op_models\netwars_std_models\mse
RedEagle_INE-100	Netwars\Sim_Domain\op_models\netwars_std_models\crypto
sat_term_basic	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
sat_term_etssp_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_etssp_non_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_etsspG3_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_etsspG3_non_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_generic_1port	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_generic_8port	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_tssp_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
sat_term_tssp_non_nodal	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
satellite_generic	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
SB-3865 1 Stack	Netwars\Sim_Domain\op_models\netwars_std_models\mse
SB-3865 2 Stack	Netwars\Sim_Domain\op_models\netwars_std_models\mse
SB-3865 3 Stack	Netwars\Sim_Domain\op_models\netwars_std_models\mse
sen	Netwars\Sim_Domain\op_models\netwars_std_models\mse
Server_4Slot	Netwars\Sim_Domain\op_models\netwars_std_models\trafgen
SHM-1337	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
sincgars_inc_adv	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
sincgars_rt	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
SMU	Netwars\Sim_Domain\op_models\netwars_std_models\mse
SRAP_application_v2	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
SRC-57	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
STU-III	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
TACINTEL	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
TCDL_Radio	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
tcp_pep_adv	Netwars\Sim_Domain\op_models\netwars_std_models\pep
TD1271	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
Thales_SONET_Datacryptor	Netwars\Sim_Domain\op_models\netwars_std_models\usfk_models
Timeplex_CX-1500	Netwars\Sim_Domain\op_models\netwars_std_models\switches
Timeplex_Link_100	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
Timeplex_Link_2	Netwars\Sim_Domain\op_models\contributed_models\navy_spawar_models
trc-170	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-170 V2	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-170 V3	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-170 V5	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-173B	Netwars\Sim_Domain\op_models\netwars_std_models\radio\trc170
trpak_gen	Netwars\Sim_Domain\op_models\netwars_std_models\misc\utility
TSC-100A	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-152 w ETSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-152 w TSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-152 wo TSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-152_C_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-152_Ku_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-152_X_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-154	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-161_C_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated

TSC-161_Ka_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-161_Ku_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-161_X_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-85B	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-85C	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-85C w ETSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-93B	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-93C	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSC-93C w ETSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
tsc-94	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
TSC-94A	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
TSQ-190	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
ttc-39	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-39A V3	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-39A V4	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-39D	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-39E	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-42	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-46 LEN	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-48 SEN	Netwars\Sim_Domain\op_models\netwars_std_models\mse
TTC-56	Netwars\Sim_Domain\op_models\netwars_std_models\mse
udp_comp_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
udp_comp_ethernet_adv	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
uhf_rt	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
UHF_SATCOM_CPS	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_NCS_Platform	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Satellite_FLTSATCOM	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Satellite_UFO	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
UHF_SATCOM_Terminal_Platform	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-59 w ETSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-59 w TSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-59 wo TSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-60A w ETSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-60A w TSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-60A wo TSSP	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
USC-60A_C_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-60A_Ku_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-60A_X_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-65_V1_C_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-65_V1_Ku_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-65_V1_X_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-65_V2_C_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-65_V2_Ku_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
USC-65_V2_X_Band	Netwars\Sim_Domain\op_models\netwars_std_models\deprecated
Venue 2000	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
vhf_rt	Netwars\Sim_Domain\op_models\netwars_std_models\radio\sincgars
VIXS Bridge	Netwars\Sim_Domain\op_models\netwars_std_models\vtc
voice_config	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway
voip_phone	Netwars\Sim_Domain\op_models\netwars_std_models\media_gateway
Wireless_Configuration	Netwars\Sim_Domain\op_models\netwars_std_models\misc\utility
WSC-6 V5	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
WSC-6 V6	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
WSC-6 V7	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
WSC-6 V9 C Band	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
WSC-6 V9 X Band	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
WSC-8	Netwars\Sim_Domain\op_models\netwars_std_models\satellite
Zydacron	Netwars\Sim_Domain\op_models\netwars_std_models\vtc

APPENDIX O: NETWARS DOCUMENTATION SET

Figure O-1 illustrates a NETWARS documentation set.

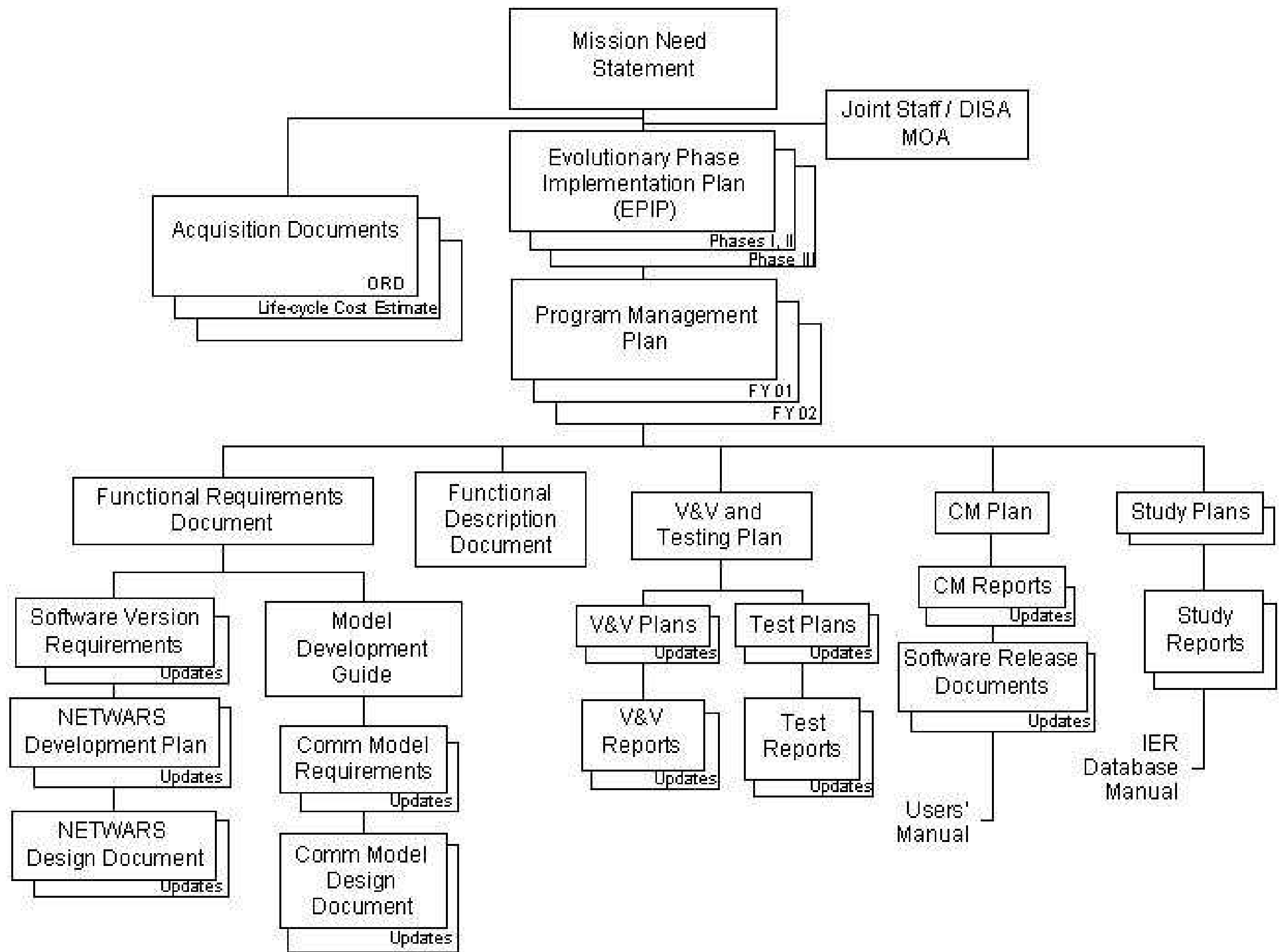


Figure O-1: NETWARS Documentation Set

APPENDIX P: CREATING MODEL REPOSITORIES IN NETWARS

The repositories are the shared object files that represent a set of models (model library). Using the repositories precludes the necessity for dynamic binding of simulation. DES in NETWARS supports dynamic binding of simulations implicitly in the sense that execution of a simulation can automatically trigger the binding process. The underlying utility that automates this process is called *op_runsim*. This utility can be used to execute simulations from a NETWARS console on the host computer just as Scenario Builder launches it from the *Configure/Run Discrete Event Simulation* dialog. *op_runsim* is essentially the starting point for all dynamically bound simulation programs. It determines which component files a simulation needs; it then uses the host computer's linker to load all the components and bind them together. Finally, it begins executing the simulation.

To avoid the dynamic binding process of user-defined components during simulation runtime, use the *op_mkso* utility to bind the user-defined components (such as process models, pipeline stages, and external files) into a single larger object called a *repositor*y. Then use this repository during simulation startup. From the linker's point of view, a repository exists as a shared object file.

Building a Repository

On the OPNET Console (Start/Program/OPNET Modeler 8.1/OPNET Console), type the following command:

For building a development repository:

```
op_mkso -env_db
“<drive_letter>\Netwars..\Sim_Domain\op_admin\env_db8.1″ -type repos -
m NAME_OF_REPOSITORY -pr_files ALL -ps_files ALL -ex_files ALL -
comp_trace_info TRUE -kernel_type development -c
```

For building a optimized repository:

```
op_mkso -env_db
“<drive_letter>\Netwars..\Sim_Domain\op_admin\env_db8.1″ -type repos -
m NAME_OF_REPOSITORY -pr_files ALL -ps_files ALL -ex_files ALL -
comp_trace_info TRUE -kernel_type optimized –c
```

Using a Repository

- Make sure that repository file *NAME_OF_REPOSITORY.i0.sid.so* (development) or *NAME_OF_REPOSITORY.i0.sio.so* (optimized) is in one of the directories listed in the `mod_dirs` preference of your env_db file (located in “<drive_letter>\Netwars..\Sim_Domain\op_admin” folder)
- Put this environment variable in the env_db file (located in “<drive_letter>\Netwars..\Sim_Domain\op_admin” folder)
  `repositories    : NAME_OF_REPOSITORY`

APPENDIX Q: TROUBLESHOOTING NETWARS SIMULATION

Refer to OPNET COTS documentation for troubleshooting a DES in the "General Tutorials | Troubleshooting Modeler Tutorials" section.

APPENDIX R: FREQUENTLY ASKED QUESTIONS

Table R-1 outlines the solution to several frequently asked questions (FAQ).

Table R-1: FAQs

Question/Problem	Solution
How should I set the mod_dirs environment variable for the various env_db files for custom model development?	There are two environment database files that the developer needs to be aware of when doing any development. One env_db file, which is used by NETWARS, is located under the Scenario_Builder\op_admin folder of the NETWARS installation. The other env_db file is the OPNET Modeler env_db file. This file is located in the op_admin folder of the opnet_user_home, and these environment settings are used when the OPNET Modeler software is used. All the new models developed are saved in the primary mod_dirs (first entry of the mod_dirs environment variable). To use the custom models in the NETWARS environment the user needs to include this mod_dirs entry in the NETWARS env_db files. Please note that if the custom models are modified NETWARS or OPNET Standard models, they must be placed before the NETWARS and OPNET Standard model directories in both the env_db files.
How do I enable the debug mode in my simulation? How can I enable OPNET debugger in my NETWARS simulation?	To enable the OPNET debugger (odb) for NETWARS simulation, check the “Use OPNET Simulation Debugger” checkbox under Execution | OPNET Debugger in the Configure/Run dialog box before running the simulation.
I want to specify simulation attributes. Where can I do that?	This can be done in the Configure/Run dialog box before running the simulation. The simulation attribute can be defined under Inputs| Global Attributes.
I want to see the routing tables generated during the simulation. How can I do that?	Simulation attribute “IP Routing Table Export/Import” under Inputs | Global Attributes needs to be set in the Configure/Run dialog box. The integer value 1 is used for this attribute to export the routes; 2 (import) and 0 are not to be used.
I have the TIREM data files on my system but still the TIREM is not enabled. Why?	To enable TIREM in the simulation, please make sure that: • The files are WOTL format data files. • These files are located in the primary mod_dirs. • The “TIREM” checkbox is turned on. This checkbox is available in the “Advanced Simulation Configuration” dialog box.
No traffic flows through the network even though there are IERs specified in the text files?	There are two things that need to be checked: • Make sure that the “Import IERS from Text Files” option is checked in the General Description block. • If the IER text files are changed with the scenario open in the editor, make sure that the IER text files are refreshed from the File menu.
All my data IERs are failed or reported miscellaneous. What could be the reason?	There could be many reasons why the data IERs may not go through the network, including the following: • Routes were not determined: For some reason if the routes were not determined by the routing protocol either because of configuration issues or convergence problems the packets get dropped and hence the IER is not received at the destination. • Circuits were not set up: For Promina or Multiplexers if the circuits are not set up correctly the traffic (IER) cannot flow. • Large IERs: IERs of very large size can be dropped because of several reasons (refer to the following question for details). • Other protocol issues: These issues are logged in a simulation log file per scenario. This file can be accessed via Results-> Simulation Log.

Question/Problem	Solution
I have large data IERs (greater than 20 Mb) and none of the IERs go through? Why?	IERs of very large sizes can be dropped because of various reasons including the following: • Reassembly timeouts: If the time taken by the complete IER to reach the destination is more than the reassembly time, the queue is flushed. • Buffer overflows: For IERs of such large sizes the queues at various interfaces may overflow causing packet drops. • Low processing speeds: If the IP processing speeds are low, the servicing of the IP datagrams may be slower, causing reassembly timeouts on the destination host. • Transport layer: If the transport layer protocol is not reliable (e.g., UDP) or has a limit on the size of the application layer packets it can handle, this may also be responsible for the drops at the transport layer. In the case of the application layer in NETWARS, the size limitation of the transport layer is handled by segmenting the application layer packets.
What are Miscellaneous IERs? How are they calculated?	The IERs are reported as Miscellaneous when they do not make it to the destination before the simulation completes. $\text{Miscellaneous IERs} = \text{IERs Sent} - (\text{IERs Received} + \text{IERs failed})$
I see some of the IERs reported as Miscellaneous. Where did they go?	The IERs are reported as Miscellaneous when they do not make it to the destination before the simulation completes. There can variety of reasons for this including the following: • Lossy networks: The packets are dropped in the network without intimation to the host. • Transport layer: If the transport layer is not reliable, the packets dropped in the network are never retransmitted, and the IERs are counted as miscellaneous. • Delays: Higher delays in the network may cause the simulation to be completed before some of the IERs can reach the destination. The IER stop times can be changed so that the IER has ample time to reach the destination before the simulation ends.
What are the Perished IERs?	These are the IERs that reached the destination after the perishability time defined in the IER/Demand definition.
I see the IERs to be received at the destination, but when I look at the grade of service statistics I see that it reports a lesser percentage of IERs received. Why?	The grade of service is the percentage of IERs sent that were received within the perishability time limit. If an IER received takes more time than the perishability value specified for it, it will not be counted for the grade of service calculation.

APPENDIX S: MIGRATION FROM EARLIER OPNET VERSIONS

This Appendix will discuss the most important issues regarding enhancements to the OPNET product as they pertain to NETWARS Model Development. For a comprehensive list of enhancements, consult the Release Notes for the Product Release of interest. The following need to be noted when upgrading models that were built using earlier versions of OPNET:

1) Modeler Release 12.1[41]

   a) Functional Enhancements

      i) Model Comparison

         Model Comparison allows comparison of different versions of models to see what differences exist between versions. For example, a comparison between a new version of a standard model with a customized copy of the previous version to estimate the work required to carry the customization forward in the new version.

         Compare the following types of Models:

         (1) Protocol Models (standard, specialized, and custom)
         (2) Process Models
         (3) Node, link, path, demand, and wireless domain models (base and derived)
         (4) Packet formats
         (5) ICIs
         (6) Text files (e.g., .ef, .ets, .ex.c, and .gdf)

   b) For additional enhancements, please see release notes

2) Modeler Release 12.0.PL5[42] and Modeler Release 12.0 PL3[42]

   a) For enhancements, please see release notes

3) Modeler Release 12.0.PL0 and PL1[42]

   a) Application Model Enhancements

      i) Application Delay Tracking

         This feature will identify the largest sources of application delay during a discrete event simulation.

         The Standard Models' GNA application models have been updated to support this feature. To use this feature with custom-developed application models, you will need to modify your process models.

[41] For additional information see the OPNET Modeler Suite Release Notes for Product Release 12.1
[42] For additional information see the OPNET Modeler Suite Release Notes for Product Release 12.0

Application Delay Tracking can answer the questions:

(1) What are my slowest applications?
(2) Did my application spend more time processing at the application layer or in the network?
(3) What intermediate nodes or links were the largest bottlenecks
(4) Where were application packets dropped?

b) BGP Model Enhancements

i) Attribute Organization

The BGP Attribute structure has been significantly reorganized. Support was added for neighbor groups, address family groups, and session groups. For more information see the BGP Model User Guide.

ii) IPv6 Support

(1) The BGP model now supports advertising of IPv6 routing information in discrete event simulations. All BGP features that are supported for IPv4 are now also supported for IPv6.

These features include:

(a) BGP route selection process
(b) Communities
(c) Policies
(d) Route reflectors
(e) Confederations

(2) Note: MPLS-BGP VPNs are NOT supported for IPv6.

c) IP Model Enhancements

i) Implementation Change for IP Fragmentation

The IP model suite now uses the Segmentation and Reassembly (SAR) package (one of the DES kernel APIs) in its implementation of IP fragmentation. Fragmentation was already modeled in the previous release and if you are using the standard IP model without modifications, this enhancement will not affect the way you configure the model or the simulation results you get in this release. If you have added custom code to the IP model, you might need to modify your models to account for the new fragmentation architecture.

d) For additional enhancements, please see OPNET Modeler 12.0 release notes

APPENDIX T: SUPPORTED CLASSIFICATION VALUES

Current public attribute definition of the classification attribute (*string*) has the following values:

- Unclassified
- Classified
- Confidential
- Secret
- Top Secret.

Models can support additional custom, user-defined classifications.

APPENDIX U: SELF-DESCRIPTION GUIDELINES

The self-description information for each model varies depending on factors such as the category to which the model belongs (e.g., a network layer device versus a datalink layer device) and the technologies it can support. Among the most common information that is looked for is the information on the ports. The following discussion points out how this information is specified for the NETWARS models. If the custom models do not support the same packet format information as the NETWARS models, the developer will have to develop self-description information based on the models developed.

The capacity planning feature uses the self-description information. Refer to the Section 3, "Compliance for Non-Discrete Event Simulation (Capacity Planning)" subsection for details.

Port and Port Groups

For all the NETWARS models, each port category must have a self-description port object. For example, MRC-142 (NETWARS Standard device model) has the following ports:

- Point-to-point ports: ptp_pt_0, ptp_pt_1
- Radio ports: radio_tx_0, radio_tx_1

Two port objects will be created, ptp_pt_<n> and radio_tx_<n>, with a range from 0 to 1 (see Figure U-1).

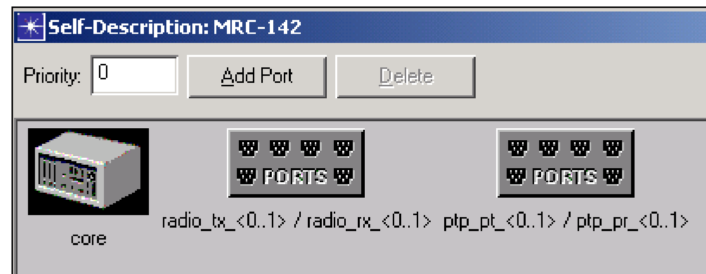


Figure U-1: Self-Description Port Objects

Each port category needs an "*interface type*" characteristic defined for it. This interface type defines the technologies that the set of ports support. In Table U-1 technologies are defined based on the packet formats for each port category. Then, based on this definition, in Table U-2 interface types are defined for each port category depending on what packet formats they support (see for details).

Table U-1: Packet Formats to Interface Types

Packet Formats	Suggested Technologies (Interface Type)
ams_atm_cell	atm:OC1,atm:OC3,atm:OC12,atm:OC24,atm:OC48
Ckswpkt	Circuit_Switched:Voice WAN
ethernet_v2	ethernet:10BaseT, ethernet:100BaseT, ethernet:1000BaseX
ip_dgram_v4	serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC 3, serial:OC12,serial:OC36,serial:OC48,serial:OC192
KG194_19	Encryptor:KG194_KIV19
KG84_7	Encryptor:KG84_KIV7
mse_data_packet	Circuit_Switched:Data_WAN
phone_switch	Circuit_Switched:Voice_LAN
pro_cx_pk	Multiplexer:CellXpress
pro_hello_pk, pro_wan_pk	Multiplexer:WAN
satellite_pk	Radio:Satellite
havequick_packet	Radio_Wired:Sincgars INC Interface
sincgars_inc_packet	Radio_Wired:EPLRS INC Interface
eplrs_inc_packet	Radio_RF:Sincgars
radio_packet	Radio_RF:EPLRS Routing
eplrs_packet_0,eplrs_packet_1, eplrs_packet_2, eplrs_packet_3, eplrs_packet_4, eplrs_packet_5, eplrs_packet_6, eplrs_packet_7,eplrs_eot_packet, eplrs_routing_packet	Radio_RF:EPLRS Broadcast1
ale_word_data, ale_word_lqa, ale_word_std	Radio_RF:EPLRS Broadcast2
Packet Formats	Radio_RF:EPLRS Broadcast3
ams_atm_cell	Radio_RF:EPLRS Broadcast4
Ckswpkt	Radio_RF:EPLRS Broadcast5
ethernet_v2	Radio_RF:EPLRS Broadcast6
ip_dgram_v4	Radio_RF:EPLRS Broadcast7
KG194_19	Radio_RF:EPLRS Broadcast8

Table U-2: Supporting Technologies per Port Category

Port Type	Supporting Packet Formats	Supported Technologies
dtg	ip_dgram_v4, KG194_19, KG_84_7,mse_data_packet,ckswpkt	serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC3,serial:OC12,serial:OC36,serial:OC48,serial:OC192,encryptor:KG194_KIV19,encryptor:KG84_KIV7,circuit_switched:Data_WAN,circuit_switched:Voice_WAN
ckt	phone_switch	circuit_switched:Voice_LAN
dat	ip_dgram_v4, KG194_19, KG_84_7	serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC3,serial:OC12,serial:OC36,serial:OC48,serial:OC192,encryptor:KG194_KIV19,encryptor:KG84_KIV7
lan	ip_dgram_v4, KG194_19, KG_84_7, ckswpkt	serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC3,serial:OC12,serial:OC36,serial:OC48,serial:OC192,encryptor:KG194_KIV19,encryptor:KG84_KIV7, circuit_switched:Voice_WAN
wan	pro_cx_pk, pro_hello_pk, pro_wan_pk	multiplexer:CellXpress,multiplexer:WAN
intf (KG 194)	ams_atm_cell, ckswpkt, ethernet_v2, ip_dgram_v4, KG_194_19, kg84_7, mse_data_packet, mse_hello_packet, phone_switch, pro_hello_pk, pro_wan_pk	atm:OC1,atm:OC3,atm:OC12,atm:OC24,atm:OC48,circuit_switched:Voice_WAN,ethernet:10BaseT, ethernet:100BaseT,ethernet:1000BaseX,serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC3,serial:OC12,serial:OC36,serial:OC48,serial:OC192,encryptor:KG194_K

enc (KG 194)	KG194_19, phone_switch, ckswpkt	encryptor:KG194_KIV19,circuit_switched:Voice_LAN,circuit_switched:Voice_WAN
intf (KG 84)	ams_atm_cell, ckswpkt, ethernet_v2, ip_dgram_v4, KG_194_19, kg84_7, mse_data_packet, mse_hello_packet, phone_switch, pro_hello_pk, pro_wan_pk	atm:OC1,atm:OC3,atm:OC12,atm:OC24,atm:OC48,circuit_switched:Voice_WAN,ethernet:10BaseT,ethernet:100BaseT,ethernet:1000BaseX,serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC3,serial:OC12,serial:OC36,serial:OC48,serial:OC192, encryptor:KG194_K
enc (KG 84)	KG87_7, phone_switch, ckswpkt	encryptor:KG84_KIV7,circuit_switched:Voice_LAN, circuit_switched:Voice_WAN
satellite terminal	ckswpkt, ip_dgram_v4, KG_194_19, KG84_7, mse_data_packet, pro_hello_pk, pro_wan_pk	circuit_switched:Voice_WAN,serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3,serial:OC3,serial:OC12,serial:OC36,serial:OC48, serial:OC192,encryptor:KG194_KIV19, encryptor:KG84_KIV7,circuit_switched:Data_WAN, multiplexer:WAN
satellites	satellite_pk	radio:Satellite
radio sincgars inc interface	sincgars_inc_packet	radio_wired:Sincgars_INC_Interface
radio eplrs inc interface	eplrs_inc_packet	radio_wired:EPLRS_INC_Interface
radio sincgars RF	radio_packet	radio_rf:Sincgars
radio EPLRS ptp RF	ip_dgram_v4	serial:DS0,serial:DS1,serial:DS3,serial:T1,serial:T3, serial:OC3,serial:OC12,serial:OC36,serial:OC48, serial:OC192
radio EPLRS bn RF	eplrs_packet_0,eplrs_packet_1, eplrs_packet_2, eplrs_packet_3, eplrs_packet_4, eplrs_packet_5, eplrs_packet_6, eplrs_packet_7,eplrs_eot_packet, eplrs_routing_packet	radio_rf:EPLRS_Broadcast

APPENDIX V: IP AUTO ADDRESSING IN CUSTOM MODELS

Custom Node with Port Mappings Attribute

Overview

For nodes that have a direct mapping from one port to another, such as a multiplexer device, custom model developers can define a particular compound attribute to facilitate IP auto addressing in DES and in Scenario Builder.

Details

Define a compound model attribute at the node attribute level called *Custom Port Mappings*. This attribute must have the following sub-attributes:

- **Local Port (string).** This serves as a string representation of the port on the local device that can connect to an IP device.
- **Remote Ports (compound):**
  - **Remote Device (string).** This serves as a string representation of the remote device's hierarchical name, that is, the device to which the local device will connect either directly or indirectly.
  - **Remote Port (string).** This serves as a string representation of the port on the remote device that can connect to an IP device.

It must also have a row for each port on the local device that can connect to an IP device, so each port name (based on the transmitter module's name) should appear exactly once in the "Custom Port Mappings" table under the "Local Port" column.

The file IP auto-addressing functionality of the NETWARS Standard model library has code that will make use of these attributes on any device on which it finds them.

Example

In this example a model developer has built a custom device that has two LAN-side ports, two WAN-side ports, and one radio port. The device can connect its LAN-side ports to IP devices, so IP auto addressing needs a way to topowalk from the LAN-side port on one device to the associated LAN-side ports on other devices. The *Custom Port Mappings* attributes across all the instances of the custom device map those LAN-side ports to each other (see Figure V-1 and Figure V-2).

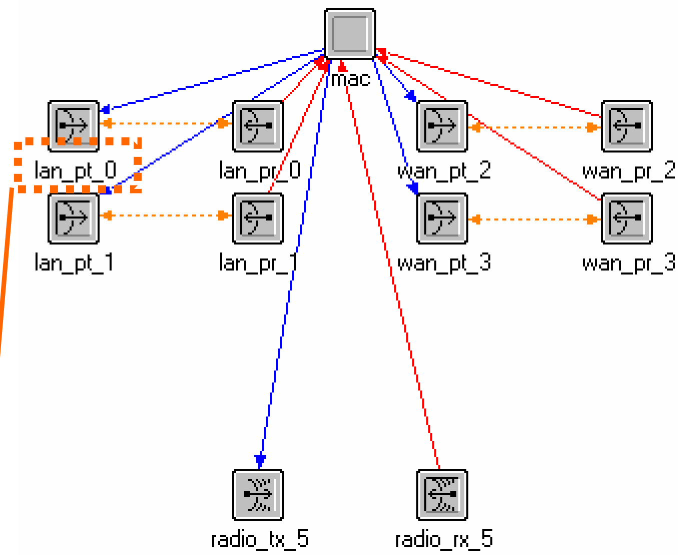


Figure V-1: Node Model Contents

(Custom Port Mappings) Table

Local Port	Remote Ports
lan_pt_0	(...)
lan_pt_1	(...)

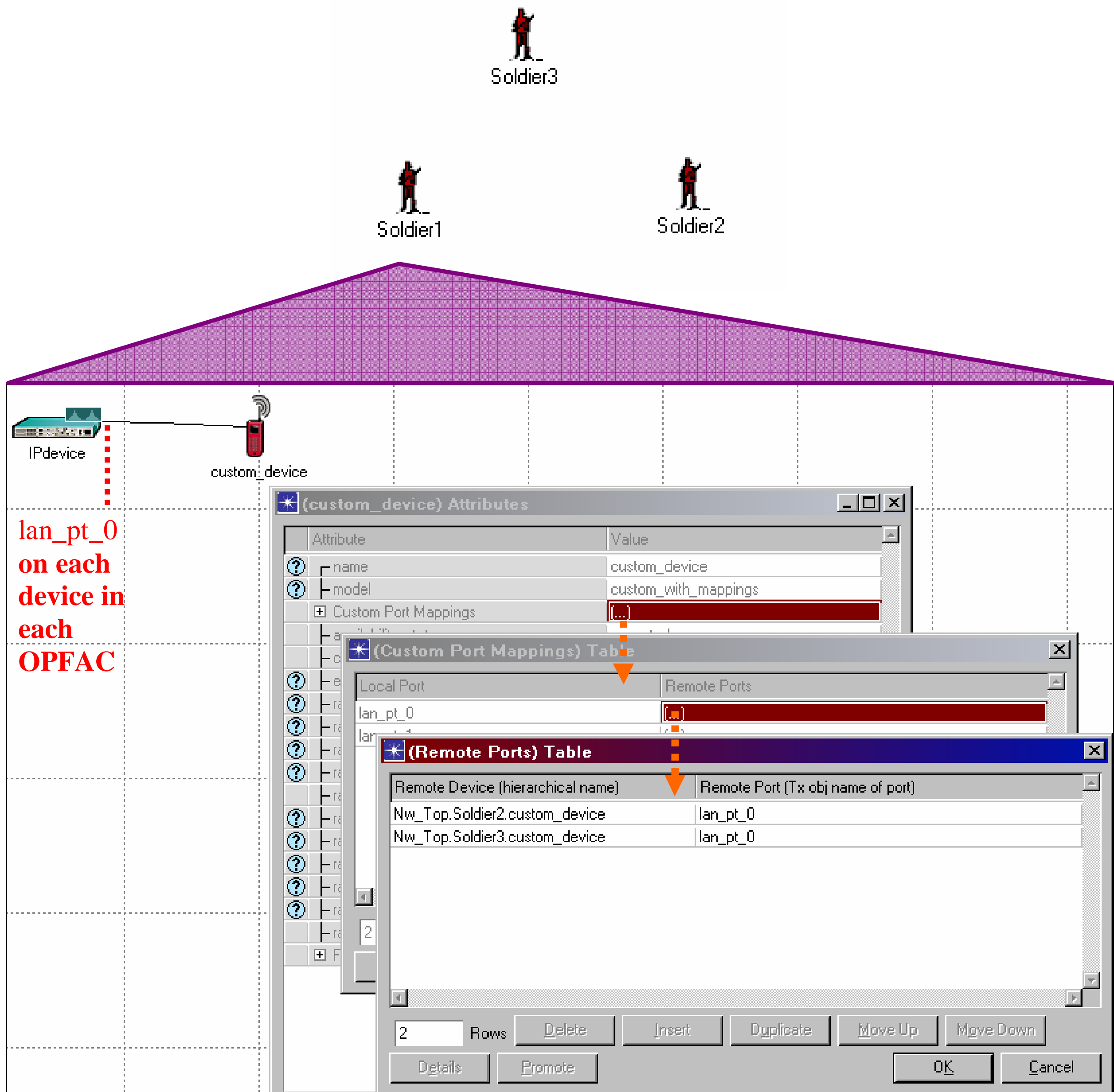
2 Rows | Delete | Insert | Duplicate | Move Up | Move Down | Details | Promote | OK | Cancel

(Remote Ports) Table

Remote Device (hierarchical name)	Remote Port (Tx obj name of port)
Nw_Top.Soldier2.custom_device	lan_pt_0
Nw_Top.Soldier3.custom_device	lan_pt_0

2 Rows | Delete | Insert | Duplicate | Move Up | Move Down | Details | Promote | OK | Cancel

Figure V-2: Custom Device Attribute Values in OPFAC Soldier 1


Soldier3
Soldier1
Soldier2
IPdevice
custom_device
lan_pt_0 on each device in each OPFAC
(custom_device) Attributes
Attribute
Value
name
custom_device
model
custom_with_mappings
Custom Port Mappings
(Custom Port Mappings) Table
Local Port
Remote Ports
lan_pt_0
(Remote Ports) Table
Remote Device (hierarchical name)
Remote Port (Tx obj name of port)
Nw_Top.Soldier2.custom_device
lan_pt_0
Nw_Top.Soldier3.custom_device
lan_pt_0
2
Rows
Delete
Insert
Duplicate
Move Up
Move Down
Details
Promote
OK
Cancel

Custom IP Auto Addressing Implementation

Overview

For some cases, the makeup of a node model may not permit the use of either framework described above. In those cases, model developers will need to write custom code in a file reserved only for custom models.

Details

Add a node model attribute *Custom IP Auto Address ID (integer)* to the node model for which custom IP auto addressing implementation is wanted. The device type must have a unique attribute with respect to all other implementations that already exist. Examine the file *nw_custom_ip_auto_addr.h* for a list of `const int` declarations that define a unique ID for device types that already exist. Add a new declaration for the new type to this file and assign that value to the *Custom IP Auto Address ID* attribute defined for the custom device.

Next, add the code needed to support the custom device model. Modify the file *nw_custom_ip_auto_addr.ex.c*. That file contains a function `void nw_custom_ip_traverse ()` that primarily serves to call the correct routine that implements custom IP auto addressing. It takes these parameters:

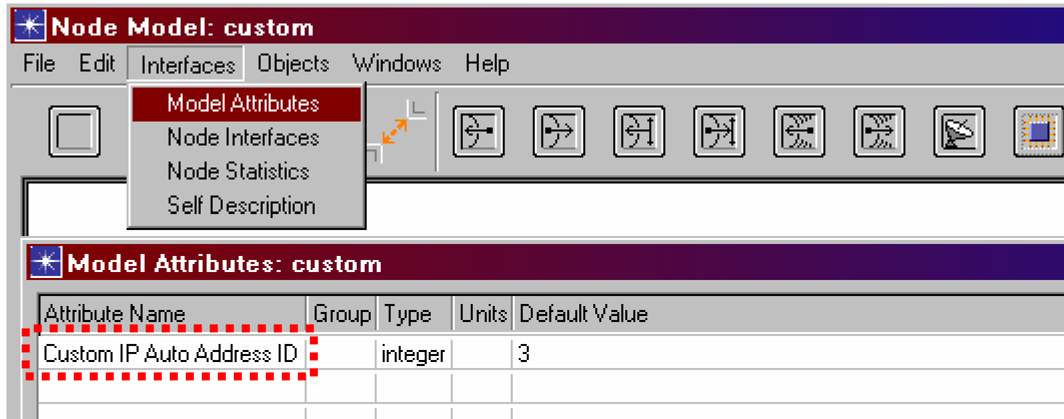
- **ipaa_id.** The IP auto addressing ID assigned to the node attribute; it will use this to determine which routine to call to perform the topology walk over custom devices.
- **local_node_objid.** Objid of the node of the current iteration of the IP graphwalk.
- **local_link_objid.** Objid of the link of the current iteration of the IP graphwalk.
- **neighbor_node_link_objids_lptr.** List of ports (identified by node/link Objid pairs) that have an IP graph connection to the passed port (identified by the passed local node and link objids). This function must add entries to this list prior to returning.

An entry needs to be added to the `switch` statement of `nw_custom_ip_traverse ()` to call the custom routine, and of course it will need to be added to the custom routine. This can be considered an entry point of program flow into the custom code.

Example

In this example, a model developer has added custom IP auto addressing code to support a custom device model called “Custom_Device_C” where custom IP auto addressing code already exists to support “Custom_Device_A” and “Custom_Device_B”.

Step 1: Add a *Custom IP Auto Address ID* attribute to the node model.



Step 2: Add a custom IP auto address ID constant and a function prototype for the custom IP auto addressing function to the *nw_custom_ip_auto_addr.h* header file.

```
#ifndef NW_CUSTOM_IP_AUTO_ADDR_H
#define NW_CUSTOM_IP_AUTO_ADDR_H

typedef enum IpT_Custom_Device_Model
    {
    IpC_Custom_Device_Model_A = 1,
    IpC_Custom_Device_Model_B = 2,
    IpC_Custom_Device_Model_C = 3
    };

void ip_traverse_custom_node           (int, Objid, Objid, Nw_neighbor_node_link_struct*);
void ip_traverse_custom_device_type_a   (Objid, Objid, Nw_neighbor_node_link_struct*);
void ip_traverse_custom_device_type_b   (Objid, Objid, Nw_neighbor_node_link_struct*);
void ip_traverse_custom_device_type_c   (Objid, Objid, Nw_neighbor_node_link_struct*);
```

Step 3: Add an entry to the `switch` statement of `nw_custom_ip_traverse ()` and add a custom function to the file *nw_custom_ip_auto_addr.ex.c*.

```
void
nw_custom_ip_traverse (
        int    ipaa_id,
        Objid local_node_objid,
        Objid local_link_objid,
        List* neighbor_node_link_objids_lptr)
    {
    // PURPOSE : Call the appropriate custom model IP auto addressing function.
    //
    // REQUIRES: 'ipaa_id' - custom IP auto addressing ID
    //           'local_node_objid' - Objid of the node of an iteration of the IP graphwalk
    //           'local_link_objid' - Objid of the link of an iteration of the IP graphwalk
    //           'neighbor_node_link_objids' - list of ports (identified by node/link objid pairs)
    //              that have an IP graph connection to the passed port (identified by the
    //              passed local node and link objids), this function must add entries to this
    //              list prior to returning
    //
    // EFFECTS : returns void, populates the 'neighbor_node_link_objids' list with
    //           node/link pairs that have IP associations with the port of the passed
    //           local node and link objects.
    FIN (nw_custom_ip_traverse (ipaa_id, local_node_objid, local_link_objid, nbr_node_link_objids));

    switch (ipaa_id)
        {
        case IpC_Custom_Device_Model_A:

            nw_custom_ip_traverse_device_model_a (local_node_objid, local_link_objid,
                neighbor_node_link_objids_lptr);

            break;

        case IpC_Custom_Device_Model_B:

            nw_custom_ip_traverse_device_model_b (local_node_objid, local_link_objid,
                neighbor_node_link_objids_lptr);

            break;

        case IpC_Custom_Device_Model_B:

            nw_custom_ip_traverse_device_model_b (local_node_objid, local_link_objid,
                neighbor_node_link_objids_lptr);

            break;
        }

    FOUT;
    }
void
nw_custom_ip_traverse_device_model_c (
        Objid local_node_objid,
        Objid local_link_objid,
        List* neighbor_node_link_objids_lptr)
    {
    Nw_neighbor_node_link_struct* neighbor_node_link_objids_ptr;

    // PURPOSE : ...
    // REQUIRES: ...
    // EFFECTS : ...
    FIN (nw_custom_ip_traverse_device_model_c (local_node_objid, local_link_objid, nbr_node_link_objids));

    neighbor_node_link_objids_ptr = prg_mem_alloc (sizeof (Nw_neighbor_node_link_struct));

    ...

    prg_list_insert (neighbor_node_link_objids_lptr, neighbor_node_link_objids_ptr, PRGC_LISTPOS_TAIL);

    FOUT;
    }
```

APPENDIX W: REFERENCES

- OPNET Modeler Online Documentation
- NETWARS Verification and Validation Testing Plan
- Introduction to the ACE Editors
- NETWARS Interface Control Document
- NETWARS User Manual.
- ACE Whiteboard Tutorial
- DoDD 5000.59 – DoD Modeling and Simulation (M&S) Management **http://www.dtic.mil/whs/directives/corres/html/500059.htm**
- DoDI 5000.61 – DoD Modeling and Simulation (M&S) Verification, Validation, and Accreditation (VV&A) **http://www.dtic.mil/whs/directives/corres/html/500061.htm**
- VV&A Recommended Practices Guide – Build 3.0 / September 2006 **http://vva.dmso.mil/**
- NETWARS Communications Model Verification and Validation Plan
- DoD VV&A Documentation Tool
- NETWARS 2006-2 Communications Device Model Validation and Verification Plan
- NETWARS Equipment Strings Version 1.1, June 2006
- NETWARS 2006-2 Equipment Strings Final Test Plan, OPNET 3.4.4, Delivered August 25, 2006
- DoD Standard Practice: Documentation of Verification, Validation and Accreditation (VV&A) for Models and Simulations. (MIL-STD-XXX002, Draft of 5 December 2006)

APPENDIX X: NETWARS MODEL DEVELOPMENT GUIDE CHECKLIST

The purpose of the checklist in Table X-1 is to help the developer and program managers determine levels of effort to develop new NETWARS Standard models or integrate existing models to NETWARS.

Table X-1: NETWARS Model Development Guide Checklist

NETWARS Model Development Guide Checklist		
Model Compliance	**Yes/No**	**Comments**
Does the model contain all the NETWARS required attributes? Please refer to the *NETWARS Model Development Guide* to identify the required device attributes associated with the model.		
Does the model work in capacity planner?		
Does the model support logical view?		
Does the model work in discrete event simulation?		
Does the model support IP auto addressing?		
Does the model work in the correct OPNET version that corresponds to the most recent NETWARS version?		
Does the model work with the following traffic-generation mechanisms?		
1. Standard Application Models		
2. IER		
3. Flows		
4. ACE or ACE Whiteboard		
Was the model evaluated using the Static Testing Tools?		
1. Was anything flagged?		
2. Were all flags addressed and successfully mitigated?		
Was the model evaluated using various equipment strings?		
1. Transmission Networks (Pure Transmission Devices, Prominas, Other Multiplexors)		
2. Routers		
3. Circuit Switched Voice		
4. Layer-1 Encryptors		
5. Tactical Radios		
Was the model evaluated using Capacity Planner to obtain reasonable and expected results within specifications?		
1. Shortest-hop routing		
2. Link and circuit utilizations		
3. Bandwidth requirements		
Does the model contain the following model documentations?		
1. Embedded Documentation (BNF)		
2. User Documentation		
3. Test Plan		
4. Static Testing Results (including parameters used to get the results)		

NETWARS Model Development Guide Checklist		
Model Compliance	**Yes/No**	**Comments**
5. Node Self-Description, such as: Portgroup—Interface Type Portgroup—Max Port Data Rate (optional) Coregroup—Machine Type		
Does the model interface to appropriate devices in NETWARS Standard Pallet? What devices?		
1. End System		
2. Layer 1 device		
3. Layer 2 device		
4. Layer 3 device		
5. Circuit-switched device		
6. Wireless device		
Do the model's node modules use the correct port conventions? These include:		
1. Wired Ports Transmitter Names (end with <technology>_*pt*_<*n*>)		
2. Wired Ports Receiver Names (end with <technology>_*pr*_<*n*>)		
3. Wireless Ports Transmitter Names (end with _tx_<n>)		
4. Wireless Ports Receiver Names (end with _rx_<n>)		
Does the model include the following modules? Applies only for end-system models:		
1. IER Traffic source node contains SE module		
2. Traffic sink node contains SE module		
3. Traffic source node contains application module		
4. Traffic sink node contains application module		
Does the model promote and add the following attributes? These are only for models that support radio broadcast and point-to-point operations:		
1. Are the transmitter and receiver named in a pair?		
2. Promote rx and tx (data rate)		
3. Promote rx and tx (min frequency)		
4. Promote rx and tx (bandwidth)		
5. Promote rx and tx (spreading code)		
6. Add extended Net ID attribute to tx and rx		
7. Promote rx and tx (Net ID)		
Does the model work with custom links? If yes, please answer the following question:		
Are the custom links added to the Linktypemap.gdf file and documentation?		