

JCSS

Model Development Guide (MDG)

Version 4.0

Contract DASW01 03 D 0008



Disclaimer: As of October 2007, NETWARS was redesignated by the Program Manager Office as the Joint Communications Simulation System (JCSS). JCSS was selected as the new industry name to better reflect the inherent joint communication capabilities of the software. Users should be aware that no software updates were conducted as part of the software name change.

July, 2009

Prepared for:
Defense Contracting Command -
Washington
Washington, DC 20310-5200

Prepared by:
OPNET Technologies, Inc.
Bethesda, MD 20814-7904

OPNET[®]
Optimum Network Performance

TABLE OF CONTENTS

1 EXECUTIVE OVERVIEW 1-1

1.1 PURPOSE OF THIS DOCUMENT 1-1

1.2 BENEFITS OF MAKING A JCSS-COMPLIANT MODEL 1-1

1.2.1 Leveraging a Standard Modeling Framework 1-1

1.2.2 Use of Full JCSS Functionality..... 1-2

1.3 MODELING BASICS 1-3

1.3.1 Defining the Purpose..... 1-3

1.3.2 Determining Model Requirements..... 1-4

1.3.3 Surveying Existing Models..... 1-4

1.3.4 Developing the Model..... 1-5

1.4 HOW TO USE THIS DOCUMENT 1-6

2 TECHNICAL OVERVIEW 2-1

2.1 INTRODUCTION TO JCSS MODELS 2-2

2.1.1 Goals of Model Development..... 2-2

2.1.2 JCSS Application Architecture..... 2-3

2.1.3 JCSS/OPNET Model Hierarchy 2-9

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 JCSS MODEL DEVELOPMENT 3-1

3.1 TRAFFIC MODEL DEVELOPMENT PROCESS 3-1

3.1.1 IERs..... 3-2

3.1.2 Operational Element 3-4

3.1.3 System Element 3-5

3.1.4 OE – SE Interaction 3-5

3.1.5 DoDAF Integration 3-6

3.2 COMMUNICATIONS DEVICE AND PROCESS MODEL DEVELOPMENT PROCESS 3-7

3.2.1 Development Approaches..... 3-8

3.2.2 Modifying the Existing OPNET Model to Be JCSS Compatible..... 3-8

3.2.3 Surrogating From the Existing JCSS Model..... 3-9

3.2.4 Developing a New Model 3-9

3.3 MODEL INTEROPERABILITY ISSUES 3-10

3.3.1 Compatibility Issues..... 3-10

3.3.2 Interfacing Issues 3-12

3.3.3 Communication Aspects..... 3-14

3.3.4 Self-Description Issues 3-17

3.3.5 Versioning Issues 3-18

3.4 JCSS CAPACITY PLANNING COMPLIANCE REQUIREMENTS 3-19

3.4.1 Factors of Interest during Analytical Modeling in Capacity Planning 3-19

3.4.2 Handling CP Routing..... 3-19

3.4.3 Logical Views 3-22

3.4.4 Handling Models Modifying Message Sizes 3-23

3.4.5 Handling Specific Port Selection for Alternate Links Selection in the CP.. 3-23

3.5	METHODOLOGIES FOR CREATING DEVICES FOR JCSS WITH DEVICE CREATOR.....	3-24
3.5.1	Model Names	3-27
3.5.2	Creating a Custom Device	3-28
3.6	COMPLIANCE FOR END-SYSTEM DEVICES	3-29
3.6.1	Attributes.....	3-30
3.6.2	Self Descriptions	3-30
3.6.3	Required Modules.....	3-31
3.6.4	End-System Devices Categories	3-34
3.6.5	Interfaces and Packet Formats	3-35
3.6.6	Interfacing with Other Classes.....	3-35
3.6.7	Creating Custom Transport Protocols for End-Systems	3-37
3.6.8	Handling Failure/Recovery	3-38
3.6.9	Collecting Statistics	3-38
3.6.10	JCSS Standard SE Models	3-39
3.6.11	Example: Constructing a Computer Model	3-40
3.7	COMPLIANCE FOR LAYER 1 NETWORKING EQUIPMENT	3-41
3.7.1	Attributes.....	3-41
3.7.2	Required Modules	3-41
3.7.3	Interfacing with Devices	3-42
3.7.4	Handling Background Traffic	3-42
3.7.5	Handling Failure/Recovery	3-42
3.7.6	Collecting Statistics	3-43
3.7.7	Example: Constructing an Encryptor Model	3-43
3.8	COMPLIANCE FOR LAYER 2 NETWORKING EQUIPMENT	3-44
3.8.1	Attributes.....	3-44
3.8.2	Required Modules	3-44
3.8.3	Initialization	3-45
3.8.4	Interfacing with End-System Devices and Networking Equipment	3-46
3.8.5	Supported Protocols	3-46
3.8.6	Handling Failure/Recovery	3-46
3.8.7	Collecting Statistics	3-47
3.8.8	Example: Constructing a Multi-Service Switch	3-47
3.9	COMPLIANCE FOR LAYER 3 NETWORKING EQUIPMENT	3-48
3.9.1	Attributes.....	3-48
3.9.2	Required Modules	3-48
3.9.3	Handling Security Classification	3-50
3.9.4	Interfacing with End-System Devices and Networking Equipment	3-50
3.9.5	Supported Protocols	3-51
3.9.6	Creating Custom Routing Protocols for IP	3-51
3.9.7	Handling Failure/Recovery	3-53
3.9.8	Collecting Statistics	3-54
3.10	COMPLIANCE FOR DEVICES WITH CIRCUIT-SWITCHED TECHNOLOGY	3-55
3.10.1	Attributes.....	3-55
3.10.2	Initialization	3-55
3.10.3	Routing in Circuit-Switched Devices	3-56
3.10.4	Circuit-Switched Links	3-56

3.10.5	Interfacing with Packet-Switched Networks	3-56
3.10.6	Handling Failure/Recovery	3-57
3.10.7	Collecting Statistics	3-58
3.11	COMPLIANCE FOR WIRELESS INTERFACES	3-59
3.11.1	Attributes.....	3-59
3.11.2	Required Modules	3-61
3.11.3	Initialization	3-61
3.11.4	Interfacing with Other Classes.....	3-61
3.11.5	Interfacing with TIREM	3-62
3.11.6	Restrictions in Building Radio Devices.....	3-62
3.11.7	Handling Failure/Recovery	3-63
3.11.8	Collecting Statistics	3-63
3.11.9	Building Custom Pipeline Stages.....	3-63
3.11.10	Satellite Considerations	3-63
3.11.11	JCSS Standard Geostationary Satellite Communications System Models.....	3-64
3.11.12	Generic Satellite Device Model (for Bent Pipe Links).....	3-65
3.11.13	Generic Satellite Ground Terminal Device Model (for Bent Pipe Links).....	3-65
3.11.14	TSSP Satellite Terminal Device Model	3-65
3.11.15	Broadcast Radio Considerations	3-65
3.11.16	EPLRS Radio Considerations	3-66
3.12	COMPLIANCE FOR LINK MODELS.....	3-69
3.12.1	Attributes.....	3-69
3.12.2	Building Custom Pipeline Stages.....	3-70
3.12.3	Handling Background Routed Traffic	3-71
3.12.4	Handling Failure/Recovery	3-71
3.12.5	Building Simplex Links, Buses, and Bus Taps.....	3-71
3.12.6	Collecting Statistics	3-71
3.12.7	Documentation	3-72
3.13	COMPLIANCE FOR UTILITY NODES	3-73
3.13.1	Attributes.....	3-73
3.13.2	Self Description	3-73
3.13.3	Required Modules	3-73
3.13.4	Interfacing with Other Classes.....	3-73
3.13.5	Interfacing with the Scenario Builder GUI.....	3-74
3.14	API AND FRAMEWORK	3-75
3.14.1	Generic Circuit GUI API	3-75
3.14.2	IP Auto Addressing.....	3-78
3.14.3	Hybrid API.....	3-78
3.14.4	Link Deployment Wizard	3-79
3.14.5	Broadcast Network Framework	3-80
3.14.6	Wireless Configuration Node	3-82
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	4-14
4.3.1	Problem Statement	4-14
4.3.2	High-Level Design.....	4-14
4.3.3	Detailed Design: Event Response Table.....	4-15
4.3.4	Implementation	4-21
4.4	WIRED END DEVICE EXAMPLE 2	4-28
4.4.1	Overview.....	4-28
4.4.2	Steps.....	4-28
4.4.3	Process Model: SE.....	4-31
4.4.4	Statistics	4-32
4.5	LAYER 1 DEVICE EXAMPLE: BULK ENCRYPTOR	4-33
4.5.1	Overview.....	4-33
4.5.2	Steps.....	4-33
4.5.3	Process Model.....	4-33
4.6	LAYER 2 DEVICE EXAMPLE: MULTI-SERVICE SWITCH	4-36
4.6.1	Overview.....	4-36
4.6.2	Steps.....	4-36
4.6.3	Process Models: Voice Dispatch and Voice Over ATM	4-37
4.7	LAYER 2 DEVICE EXAMPLE: MULTIPLEXER DEVICE USING CIRCUIT API	4-39
4.7.1	Overview.....	4-39
4.7.2	Attributes and Process Model Code.....	4-40
4.8	LAYER 3 DEVICE EXAMPLE: CUSTOM ROUTER	4-43
4.8.1	Overview.....	4-43
4.8.2	Steps.....	4-43
4.8.3	Process Model: Custom Routing Protocol.....	4-46
4.9	CIRCUIT-SWITCHED DEVICE EXAMPLE: END SYSTEM.....	4-47
4.9.1	Overview.....	4-47
4.9.2	Steps.....	4-47
4.9.3	Process Model: se	4-48
4.10	WIRELESS DEVICE EXAMPLE.....	4-51
4.10.1	Overview.....	4-51
4.10.2	Steps.....	4-51
4.10.3	SE Process Model	4-53
4.11	WIRELESS DEVICE EXAMPLE 2	4-54
4.11.1	Problem Statement	4-54
4.11.2	High-Level Design.....	4-54
4.11.3	fwd module: Detailed Design	4-55
4.11.4	mac Module	4-57
4.11.5	se Module.....	4-58
4.11.6	Addressing and Other Issues.....	4-61
4.11.7	Optimization and Efficiency Considerations	4-61
4.12	SATELLITE TERMINAL GENERIC EXAMPLE	4-63
4.12.1	Node Model Contents	4-63
4.12.2	Core Self-Description Attributes	4-63
4.12.3	Additional Attributes	4-63

4.12.4	Antenna Aim Process.....	4-65
4.12.5	Description of Antenna Aim Process.....	4-65
4.13	SATELLITE TERMINAL WITH TSSP EXAMPLE	4-66
4.13.1	Overview.....	4-66
4.13.2	Node Model Contents	4-66
4.13.3	Core Self-Description Attributes	4-67
4.13.4	Additional Attributes	4-67
4.13.5	Node Model Specific Configuration.....	4-68
4.13.6	TSSP Process	4-72
4.13.7	Key Code Snippets from TSSP Process.....	4-74
4.14	SATELLITE GENERIC EXAMPLE.....	4-77
4.14.1	Overview.....	4-77
4.14.2	Node Model Contents	4-77
4.14.3	Additional Attributes	4-77
4.14.4	Satellite Switch Process	4-80
4.15	LINK MODEL EXAMPLE	4-83
4.15.1	Overview.....	4-83
4.15.2	Steps.....	4-83
4.15.3	Pipeline Stage: txdel	4-83
4.16	UTILITY NODE EXAMPLE.....	4-85
4.16.1	Overview.....	4-85
4.16.2	Details	4-85
4.16.3	Process Model.....	4-85
4.17	CONVERTING A DEVICE MODEL FROM THE OPNET STANDARD MODEL LIBRARY ..	4-87
4.17.1	Overview.....	4-87
4.17.2	Details	4-87
4.18	CP MODEL EXAMPLE.....	4-91
4.18.1	Overview.....	4-91
4.18.2	CP Implementation	4-91
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	JCSS COMPLIANCE V&V	5-4
5.2.1	JCSS Model Development Checklist.....	5-4
5.2.2	JCSS Static Testing.....	5-4
5.2.3	JCSS Equipment String.....	5-5
5.2.4	Capacity Planner	5-6
5.2.5	DoD/Joint VV&A Documentation Tool (DVDT/JVDT)	5-6
	APPENDIX A: ACRONYMS	1
	APPENDIX B: GLOSSARY	3
	APPENDIX C: ENUMERATED VALUES.....	4
	APPENDIX D: PACKET FORMATS	5

APPENDIX E: STANDARD OPNET INTERFACES AND PACKET FORMATS.....6

APPENDIX F: INTERFACE CONTROL INFORMATION (ICI) FORMATS.....9

APPENDIX G: MODELING FILE FORMATS10

APPENDIX H: OTHER FILE FORMATS.....10

APPENDIX I: MEASURES OF PERFORMANCE IN JCSS.....11

APPENDIX J: NODE MODEL DOCUMENTATION19

APPENDIX K: MODEL NAMING CONVENTIONS21

APPENDIX L: JCSS SIMULATION API AND HELPER FUNCTIONS.....23

APPENDIX M: ATTRIBUTE TYPE DEFINITIONS24

APPENDIX N: EXAMPLES OF JCSS MODELS26

APPENDIX O: JCSS DOCUMENTATION SET.....31

APPENDIX P: CREATING MODEL REPOSITORIES IN JCSS.....31

APPENDIX Q: TROUBLESHOOTING JCSS SIMULATION32

APPENDIX R: FREQUENTLY ASKED QUESTIONS.....32

APPENDIX S: FUNCTIONAL ENHANCEMENTS FROM EARLIER JCSS VERSIONS34

APPENDIX T: SELF-DESCRIPTION GUIDELINES.....37

APPENDIX U: IP AUTO ADDRESSING IN CUSTOM MODELS.....39

APPENDIX V: REFERENCES.....41

APPENDIX W: JCSS MODEL DEVELOPMENT GUIDE CHECKLIST42

LIST OF FIGURES

Figure 2-1: A Repeatable Process..... 2-2

Figure 2-2: A Model of a Repeatable Process 2-3

Figure 2-3: The JCSS Architecture..... 2-4

Figure 2-4: Sample JCSS Scenario—the Network-Level Model 2-5

Figure 2-5: Editing Device Attributes..... 2-6

Figure 2-6: An Example of the Statistics Available in DES..... 2-7

Figure 2-7: The JCSS/OPNET Model Hierarchy 2-9

Figure 2-8: Editing a JCSS Cisco 2514 Router Model 2-11

Figure 2-9: The Process Models Within the PRC Radio Model..... 2-12

Figure 2-10: The Process Model Editor..... 2-13

Figure 2-11: Editing C Code in the Process Model Editor 2-14

Figure 2-12: Receive Pipeline Stages 2-15

Figure 2-13: JCSS Model Development Life Cycle 2-16

Figure 3-1: IER Demand Model Attributes 3-3

Figure 3-2: IER Thread Information Attributes..... 3-4

Figure 3-3: Node Model with SE modules 3-5

Figure 3-4: OE-SE Interaction..... 3-6

Figure 3-5: Viewing the DoDAF Information Attribute..... 3-7

Figure 3-6: High-Level Model Development Process 3-8

Figure 3-7: Protocol Dependency (e.g., Ethernet Computer Model)..... 3-11

Figure 3-8: Module-Wide Memory (e.g., Ethernet Computer Model)..... 3-13

Figure 3-9: Declaration of Global Variable in Two Process Header Blocks..... 3-14

Figure 3-10: Default Interrupt Handling..... 3-16

Figure 3-11: Self-Description Port Objects 3-18

Figure 3-12: CP Layers..... 3-20

Figure 3-13: Example Logical View..... 3-23

Figure 3-14: Create Custom Device Dialog..... 3-27

Figure 3-15: An Ethernet End-System Device—Node Model 3-32

Figure 3-16: End-System Device with Frame Relay as the MAC Technology—Node Model
..... 3-33

Figure 3-17: A Valid End-System to End-System Connection 3-34

Figure 3-18: Example of a Circuit-Switched End-System Device—Node Model 3-35

Figure 3-19: Example of a Circuit-Switched End-System Device That Handles Voice
Applications As Well As Voice IERs..... 3-35

Figure 3-20: Remote Interrupt from the OE to the SE..... 3-36

Figure 3-21: Example of Layer 1 Networking Equipment—Node Model 3-41

Figure 3-22: Example of Layer 2 Networking Equipment—Node Model 3-45

Figure 3-23: Example of Layer 3 Networking Equipment—Node Model 3-49

Figure 3-24: Networks with Different Security Classification Levels 3-50

Figure 3-25: Circuit-Switched and Packet-Switched Network Intercommunication 3-57

Figure 3-26: Example of a Radio End-System Device—Node Model..... 3-61

Figure 3-27: An ATM Device with a Radio Interface 3-62

Figure 3-28: Internal Representation of an ATM Device and Intermediate Node 3-63

Figure 3-29: (Channel) Table..... 3-66

Figure 3-30: EPLRS ENM System Parameters Attribute.....	3-68
Figure 3-31: Path Port Associations Diagram	3-76
Figure 3-32: Circuit Interface Module Port Configuration Table.....	3-78
Figure 3-33: Explicit Traffic.....	3-79
Figure 3-34: Hybrid Traffic	3-79
Figure 3-35: Link Deployment Wizard Dialog.....	3-80
Figure 3-36: Wireless Configuration Node Modification of BER.....	3-83
Figure 4-1: Time Sequence Diagram Example.....	4-2
Figure 4-2: Sample Layer 3 Networking Equipment.....	4-5
Figure 4-3: IP Routing Parameters Attribute	4-9
Figure 4-4: Interface Information Attribute	4-10
Figure 4-5: Routing Protocol Attribute Properties.....	4-10
Figure 4-6: End-Device Node Model	4-15
Figure 4-7: Interfacing Modules of “se”	4-16
Figure 4-8: High-Level Functions of the “se_tcp” Module	4-17
Figure 4-9: se_trafgen Process Model	4-21
Figure 4-10: Open Connection State.....	4-21
Figure 4-11: Receive Traffic State.....	4-23
Figure 4-12: Process Message State	4-24
Figure 4-13: IER Handling State	4-26
Figure 4-14: Failure State	4-27
Figure 4-15: Ethernet_wkstn_adv—Node Model.....	4-29
Figure 4-16: Computer—Node Model.....	4-30
Figure 4-17: Sample Workflow Diagram for SE Process Model	4-31
Figure 4-18: Process Model for the SE Module in the Computer	4-32
Figure 4-19: Sample Code 1—Inform OE of the IER Failure, Which Will Then Record the Statistics	4-32
Figure 4-20: Encryptor—Node Model.....	4-33
Figure 4-21: Data Flow for an Encryptor	4-34
Figure 4-22: Process Model for Encryptor	4-34
Figure 4-23: Sample Code 2—Encrypting a Packet.....	4-35
Figure 4-24: Atm_uni_dest_adv Switch—Node Model	4-36
Figure 4-25: Multi-Service Switch—Node Model	4-37
Figure 4-26: Custom Multiplexer Node Model	4-39
Figure 4-27: Circuit Configuration Table	4-40
Figure 4-28: Port Configuration Table	4-41
Figure 4-29: CS_1005_1s_e_sl_adv Router—Node Model	4-44
Figure 4-30: Router with Custom Routing Protocol—Node Model.....	4-45
Figure 4-31: Process Model for a Custom Routing Protocol.....	4-46
Figure 4-32: Phone—Node Model.....	4-47
Figure 4-33: Data Flow for a Phone	4-49
Figure 4-34: Process Model for the SE Module	4-50
Figure 4-35: wlan_station_adv—Node Model	4-51
Figure 4-36: Radio SE model—Node Model	4-52
Figure 4-37: Radio End Device Node Model	4-54
Figure 4-38: fwd Module Process Model	4-56

Figure 4-39: SE Module Interfaces..... 4-58

Figure 4-40: Radio SE Process Model..... 4-59

Figure 4-41: Generic Satellite Terminal 4-63

Figure 4-42: Antenna Aim Process..... 4-65

Figure 4-43: TSSP Satellite Terminal..... 4-67

Figure 4-44: Example Configuration—TSSP Nodal Terminals – Channel Config – Downlink
 Example 4-69

Figure 4-45: Example—Circuit Configuration – Source ports..... 4-70

Figure 4-46: TSSP Process Model..... 4-72

Figure 4-47: Uplink and Downlink Tables 4-79

Figure 4-48: Satellite Switch Process Model..... 4-80

Figure 4-49: Sample Code 3—Adding Signaling Overhead to the Transmission Delay..... 4-83

Figure 4-50: Wireless Configuration Utility Node—Node Model 4-85

Figure 4-51: Wireless Configuration Object—Process Model..... 4-86

Figure 4-52: Wireless Configuration Object—Sample Code 4-86

Figure 4-53: Sample Node Model..... 4-87

Figure 4-54: Selecting “Computer” for equipment_type..... 4-88

Figure 4-55: Adding se_tcp and se_udp 4-88

Figure 4-56: Adding the net_id Extended Attribute 4-89

Figure 4-57: Equipment type attribute example 4-91

Figure 4-58: Equipment type attribute example 4-92

Figure 4-59: Equipment type attribute example 4-93

Figure 4-60: Equipment type attribute example 4-94

Figure 5-1: M&S Overall Problem Solving Process..... 5-3

Figure I-1: Enabling IER DES Reporting Capabilities Globally..... 13

Figure I-2: Enabling IER DES Reporting Capabilities Per-IER..... 14

Figure I-3: Viewing the IER Reports..... 15

Figure I-4: Selecting an IER Report 15

Figure I-5: Enabling Application Delay Tracking Per-IER..... 17

Figure I-6: Viewing Application Delay Tracking Files 18

Figure O-1: JCSS Documentation Set 31

Figure T-1: Self-Description Port Objects 38

Figure U-1: Custom IP Auto Address ID Attribute 40

LIST OF TABLES

Table 2-1: Model Types and Descriptions..... 2-10

Table 2-2: Model Development Activities 2-17

Table 3-1: Traffic Model Use Cases 3-1

Table 3-2: Properties to Determine CP Layer..... 3-20

Table 3-3: Supported Device Classes 3-24

Table 3-4: Device Class Arguments 3-28

Table 3-5: JCSS Attributes for an End-System Device 3-30

Table 3-6: Higher Layer Modules for an End-System Device 3-31

Table 3-7: Lower Layer Modules for an End-System Device 3-31

Table 3-8: Interface Modules for an End-System Device 3-33

Table 3-9: JCSS SE Process Models 3-39

Table 3-10: Attributes for Layer 1 Networking Equipment 3-41

Table 3-11: Attributes for Layer 2 Networking Equipment 3-44

Table 3-12: Modules Needed for Various Layer 2 Protocols 3-44

Table 3-13: Modules Needed by a Multi-Service Switch..... 3-45

Table 3-14: Attributes for Layer 3 Networking Equipment 3-48

Table 3-15: Higher Layer Modules for Layer 3 Networking Equipment 3-48

Table 3-16: Required Modules for Various Interface Technologies 3-49

Table 3-17: Interface Modules for Layer 3 Networking Equipment 3-50

Table 3-18: Required Attributes for a Circuit-Switched End-System Device..... 3-55

Table 3-19: Required Attributes for Circuit-Switched Layer 2 Networking Equipment..... 3-55

Table 3-20: Additional Attributes for Radio Devices..... 3-59

Table 3-21: Pipeline Stage Attributes on a Radio Transmitter 3-60

Table 3-22: Pipeline Stage Attributes on a Radio Receiver 3-60

Table 3-23: Restrictions in Building Radio Devices 3-62

Table 3-24: Required Satellite Device Attributes for Moving Orbits..... 3-64

Table 3-25: Radio Transceiver Pipeline Stages 3-64

Table 3-26: Required Attributes on a Link Model..... 3-69

Table 3-27: Required Attributes for Utility Nodes 3-73

Table 3-28: Optional Attributes for Utility Nodes..... 3-73

Table 4-1: Available IP Common Route Table API Functions 4-11

Table 4-2: Event Description Table 4-17

Table 4-3: Event Communication Mechanisms..... 4-18

Table 4-4: State Description Table 4-18

Table 4-5: Event Feasibility Table..... 4-19

Table 4-6: Event Response Table 4-19

Table 4-7: End-System—Model Attributes 4-30

Table 4-8. Circuit-Switched End-System Device—Model Attributes 4-48

Table 4-9. Radio End-System Device—Model Attributes 4-52

Table 4-10: Event Response Table for “fwd” Process..... 4-56

Table 4-11: Event Response Table for the Radio SE Module 4-58

Table 4-12: Satellite Terminal Settings Table 4-70

Table 4-13: Events of the TSSP Process Model 4-72

Table 4-14: Events of the Satellite Switch Process Model 4-80

Table A-1: Acronyms1

Table C-1: Attributes for Enumerated Data Types4

Table D-1: Packet Formats5

Table E-1: Interfaces and Packet Formats7

Table F-1: Interfaces and Packet Formats9

Table G-1: Typed File Attribute10

Table H-1: Other File Formats.....10

Table I-1: MOPs Reported by OE11

Table I-2: Statistics Groups12

Table J-1: Wired Interface Specifications.....20

Table J-2: Radio Device Interface Specifications.....20

Table J-3: Process Models21

Table J-4: External Files Needed.....21

Table L-1: JCSS APIs and their Locations23

Table N-1: List of JCSS Models (Alphabetic).....26

Table R-1: FAQs.....32

Table S-1: JCSS Wizards.....35

Table T-1: JCSS Port Types and Supported Packet Formats38

Table W-1: JCSS Model Development Guide Checklist.....42

1 EXECUTIVE OVERVIEW

1.1 PURPOSE OF THIS DOCUMENT

The purpose of the *JCSS Model Development Guide* is to provide modeling guidelines and standards for creating communications device and traffic models that are interoperable with the Joint Communication Simulation System(JCSS) System and model suite. The *JCSS Model Development Guide* provides the standards for creating JCSS 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 JCSS and OPNET COTS models within the JCSS 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 JCSS modeling environment has to offer.

1.2 BENEFITS OF MAKING A JCSS-COMPLIANT MODEL

JCSS 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. JCSS 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. The lack of standardization within the modeling community makes it difficult to reuse existing component parts. The use of a common simulation framework such as JCSS 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 JCSS Functionality

Models built according to the guidelines outlined in this *JCSS Model Development Guide* are interoperable not only with other models developed using these standards but also with the majority of the OPNET Standard Library (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), User Datagram Protocol (UDP)
- Routing Protocols such as Routing Information Protocol (RIP), Open Shortest Pathway Forwarding (OSPF), Extended Interior Gateway Routing Protocol (EIGRP), Interior Gateway Routing Protocol (IGRP), Border Gateway Protocol (BGP)
- Application Layer Protocols such as File Transfer Protocol (FTP), and Hypertext Transport Protocol (HTTP)
- Wireless Protocols such as Wireless Fidelity (WiFi) and Worldwide Interoperability for Microwave Access (WiMax); 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).

For a more complete list of protocol models, refer to OPNET Modeler online documentation.

In addition, JCSS 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 and protocol models available in JCSS include but are not limited to the following:

- Encryptors: KIV and KG-series
- Gateways: Juniper CTP, Media Gateway (PSTN to VoIP, VoATM), N.E.T. SCREAM and SHOUTip
- Multiplexers: Prominas, FCC-100
- Satellites and Earth Terminals: AN/TSC series, Standardized Tactical Entry Point (STEP), Teleport, Global Broadcast Service (GBS), UHF DAMA

- Transport Devices and Protocols: Performance Enhancing Proxy (PEP) with SCPS-TP (TAO)
- Tactical Radio Systems: PRC Radios (Falcon II, HAVEQUICK, MBITR), EPLRS, Link-11, Link-16, AN/PSC-5, TRC-170, AN/ARC
- Tactical Voice and Circuit Switches: AN/TTC series, Switch Multiplexer Unit (SMU), Digital Non-Secure Voice Terminal (DNVT), Secure Telephone Units III (STU-III)
- Voice over IP (VoIP): H.323, H.323 Border Element, H.323 Gatekeeper, SIP, SIP Proxy Server, VoIP Phone

More information can be found at the following websites:

- JCSS Program Office Website: <http://www.disa.mil/jcss/index.html>
- JCSS Technical Support Website: <http://www.jcss.disa.mil>

1.3 MODELING BASICS

JCSS is a communications system simulation tool made up of three primary simulation technologies — Discrete Event Simulation (DES), Capacity Planner (CP), and Flow Analysis (FLAN). CP is a customized analytic approach, implemented specifically for JCSS. By convention, models developed for the JCSS environment usually 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 or wireless communication effects. Additionally, because JCSS ships with the full source code to both the OPNET Standard Library (COTS) and JCSS 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. Typically, it is used to study utilization and configuration validation as it supports many of the JCSS devices. Users are encouraged to run a CP analysis before investing larger amounts of time for DES as CP provides the user with high level web reports showing the basic health of the network. However, if the user needs more detailed protocol level information DES should be used instead.

FLAN is an OPNET tool that works similarly to CP. However, the feature is different in that it is an analytical simulation that can utilize protocols such as IP to route traffic. The engine can also provide additional network statistics such as delay and jitter, among others. Refer to the Standard OPNET Documentation for more information on FLAN.

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 JCSS 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 JCSS 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 *JCSS Model Development Guide* takes on greater importance.

There are a number of things that differentiate JCSS 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 JCSS framework. Full explanations of these differences, and how to interact with them, are provided in Section 3.

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

IER Support. JCSS 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. This analytical simulation technology is custom built to handle the JCSS circuit-switched modeling construct and to allow capacity planning workflows that include wireless devices. Many JCSS models operate in both the DES and CP, but it should be noted that not all JCSS models work in both environments. Certain models are designed exclusively for DES as they are complex and detailed models. Conversely, other models are designed as high level planning models, and should only be used with CP as they lack the protocol specific details required by DES. For more information on supported model features, view the Models User Guides included with JCSS.

Classification. JCSS models support the notion of classification, which enables military network planners to build models of different security enclaves.

Interaction with JCSS Model Suites. Newly developed JCSS models must also interact smoothly with the existing device models and technology frameworks that reside within JCSS. These include Prominas and other circuit-switched devices, broadcast networks, Satellite Communications (SATCOM) devices and terminals, and message-based systems such as Link-16.

JCSS itself does not provide a model-authoring framework. Models for use in JCSS are developed using the Modeler development environment, a COTS software package produced by OPNET. This software is not available through the JCSS program office and to acquire it one must contact OPNET directly (www.opnet.com; 240-497-3000). Once a license is acquired, the user has the option of using the Modeler development environment in a standalone version or through the JCSS software. Consult the JCSS User Manual for more information on using Modeler inside the JCSS software.

Prior to beginning JCSS model development, it is important that the developer is familiar with the following materials:

- OPNET Modeler
- OPNET Device Creator
- C/C++ development language
- This *JCSS 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/index.html>) is an excellent place to obtain a background in using the Modeler framework for model development. Look especially at the OPNETWORK sessions for detailed information on OPNET methodologies and user case studies.

1.4 HOW TO USE THIS DOCUMENT

The remainder of this document covers various aspects of JCSS 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 JCSS model development and the *JCSS 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 JCSS 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 JCSS models.

Appendices. The appendices provide associated references, such as JCSS 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 JCSS 9.0

2 TECHNICAL OVERVIEW

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

This section is an overview showing what capabilities exist within a device model and how reuse is possible in JCSS 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 *JCSS 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 JCSS model library. Such models are termed JCSS-compliant models.

This section summarizes the following topics:

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

2.1 INTRODUCTION TO JCSS 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).

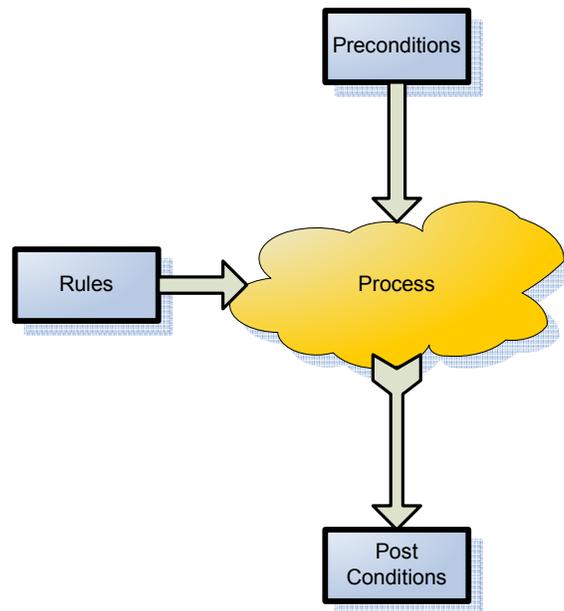


Figure 2-1: A Repeatable Process

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).

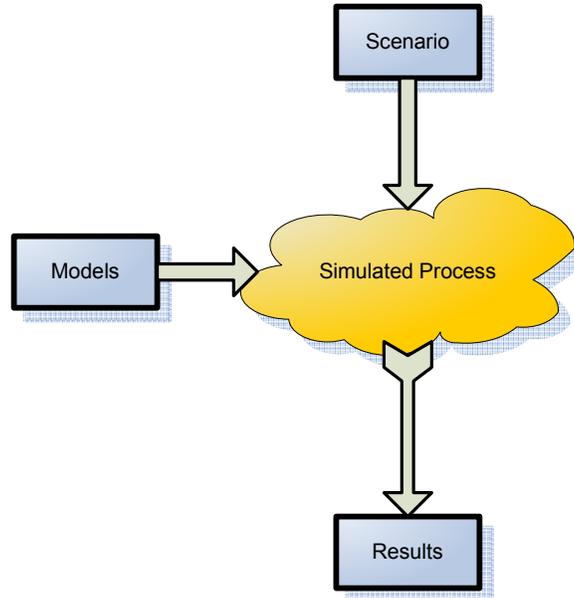


Figure 2-2: A Model of a Repeatable Process

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 JCSS-compliant communications device models. It describes a process to produce and validate these device models so they can be integrated into the JCSS simulation environment.

2.1.2 JCSS Application Architecture

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

1. Military-specific Models (Tactical Radios, Satellites, Encryptors, Multiplexers, etc.)
2. A simple-to-use capacity planning engine that provides analytic simulation
3. Support for DoD Architecture Framework (DoDAF) workflows
4. Usability Enhancements (wizards, reports, PowerPoint briefings)
5. Collaborative Planning workflow for Joint Command, Control, Communications, Computers, and Intelligence (C4I) planning.

The majority of JCSS users are analysts. JCSS 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 JCSS user can press a button to simulate the scenario. The results of the simulation can be viewed within JCSS.

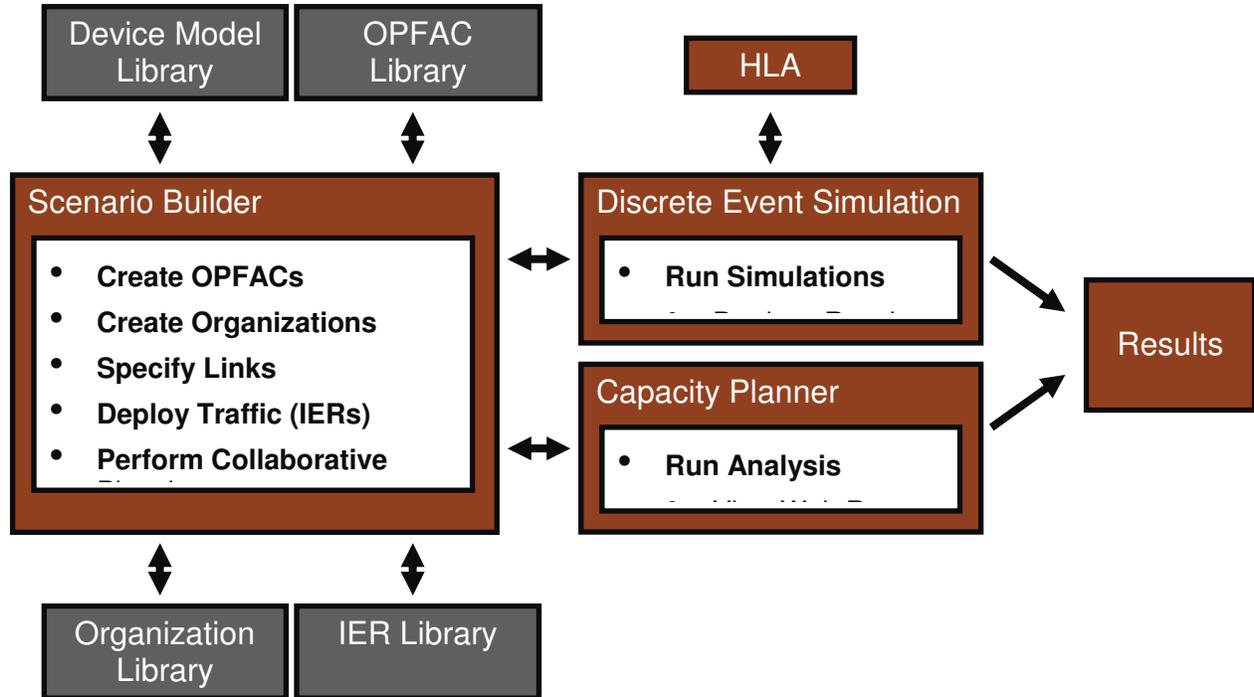


Figure 2-3: The JCSS Architecture

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

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

2.1.2.1 Scenario Builder

The JCSS Scenario Builder is the most recognizable part of JCSS. When most users think of JCSS, they think of the Scenario Builder. Within the Scenario Builder interface, users drag models from the palette 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 JCSS scenario.

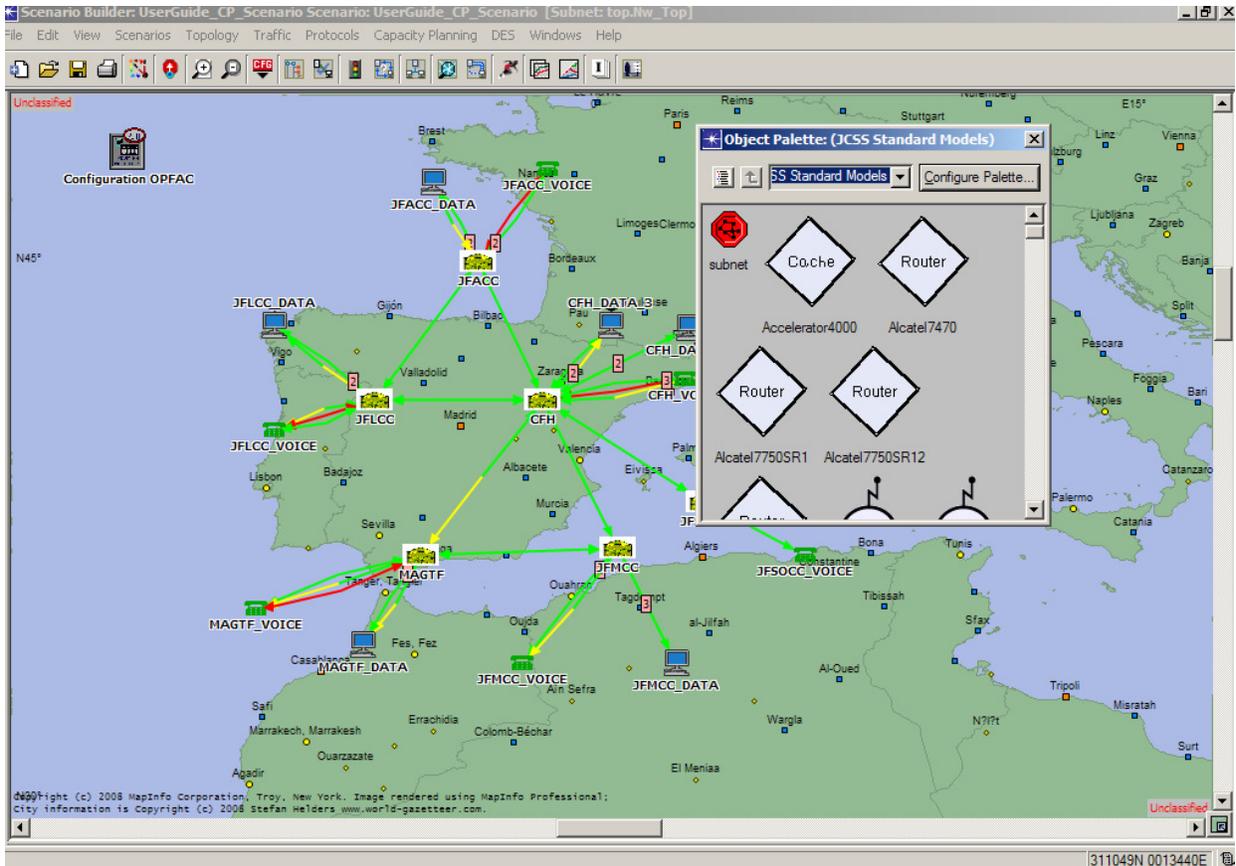


Figure 2-4: Sample JCSS Scenario—the Network-Level Model

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 JCSS routers. The list of attributes exposed is defined by the model developer, but the JCSS user is able to change these values to configure the device for simulation.

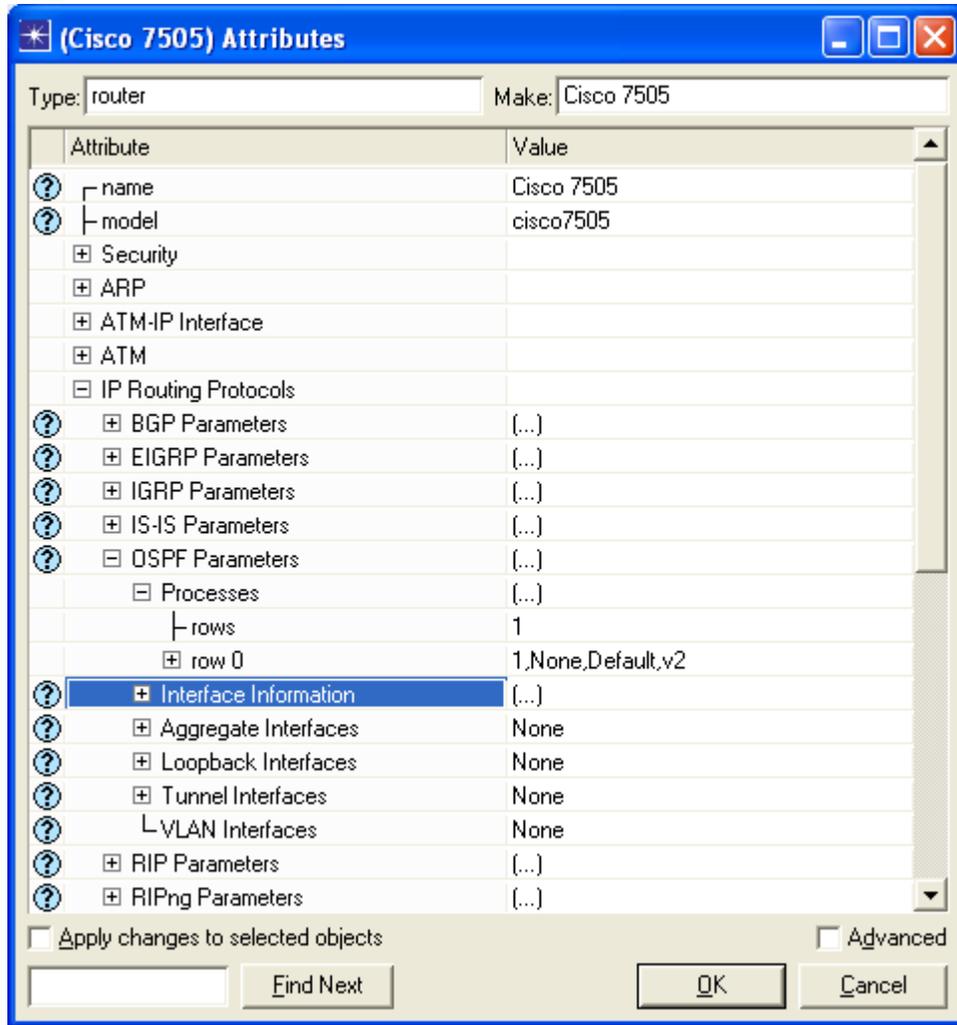


Figure 2-5: Editing Device Attributes

2.1.2.2 Capacity Planner

CP is a JCSS 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 JCSS router.

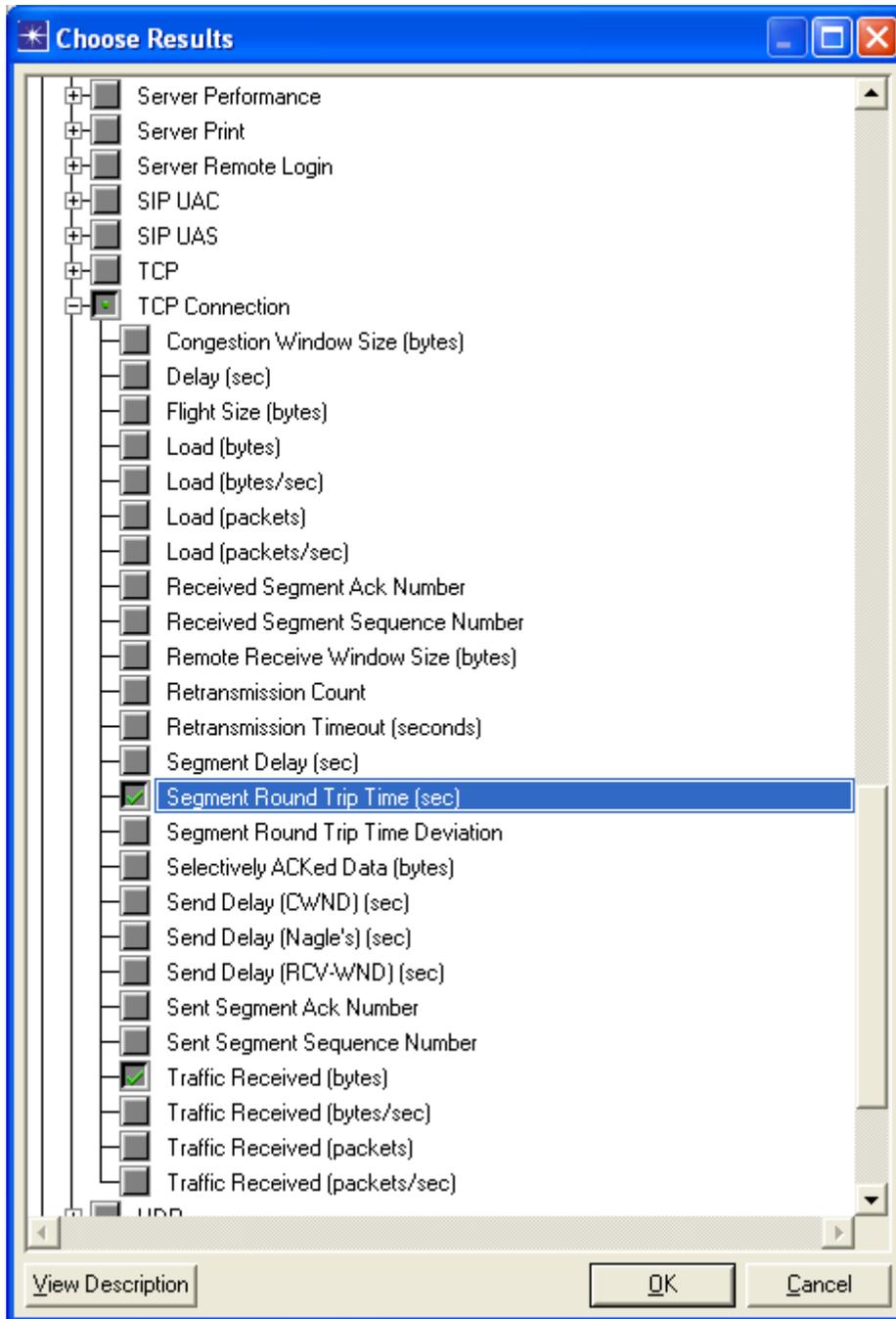


Figure 2-6: An Example of the Statistics Available in DES

With a user-selectable level of statistics granularity, JCSS 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.

To use DES with JCSS, the user is required to have both an OPNET Modeler or IT Guru license and a Simulation Runtime license. Licenses can be obtained by contacting OPNET directly (www.opnet.com) or through the JCSS Program Office (jcss@disa.mil).

2.1.2.4 JCSS Model Library

JCSS 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 JCSS 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)
- Media Gateways (including SCREAM, SHOUTip)
- Military Phone Systems
- Satellite Terminals
- Models to process DoDAF traffic: IERs
- Contributed Models (such as Link-16, TRC-170, UHF DAMA)

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 JCSS model.

2.1.2.5 JCSS Contributed Models

This section describes the necessary information needed when submitting a Device or Network model to the JCSS Program Office. The JCSS Program Office appreciates and accepts all contributed models being developed by its user community. In order to ensure proper use of the model, listed below are the requested requirements when submitting a model.

- **Model Information:** The user is expected to create a new document (in Word or Text File format) with the following information:
 - What is the purpose of the model?
 - What technologies/standards are used by the models?
 - Are there any model limitations?
 - Does this model work with all JCSS functionalities such as Capacity Planner, DES, Link Deployment Wizard, etc.? If not, what are the limitations?
 - If the models include modifications to a JCSS and/or OPNET model, what was changed (self description, ports, attributes, code, etc.) and why?
 - Was the JCSS MDG used to develop these models?
 - What Validation and Verification was performed on these models?
- **Code Information:** If the models are modifications on existing JCSS and/or OPNET models, please provide any code changes to the standard model to be commented accordingly:
 - // JCSS Modification START <model_name or model_group_name>

- // JCSS Modification END <model_name or model_group_name>
 - <model_name or model_group_name> is the name of the model or group of models for which the code change was made.
- Packaging/Sending the model: Include a zip file with all related models/files needed to run the contributed models in JCSS. The zip file should also contain the required documentation above and any additional documentation such as design documents, user guides, etc. Please send the models/project files to JCSS@disa.mil.

2.1.3 JCSS/OPNET Model Hierarchy

JCSS is built upon OPNET COTS technology, and the DES engine used by JCSS 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 JCSS, 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 JCSS uses “device” instead of “node”. Most important, JCSS employs Operational Facility (OPFAC) and Organization (Org) military ideas to create network scenarios.

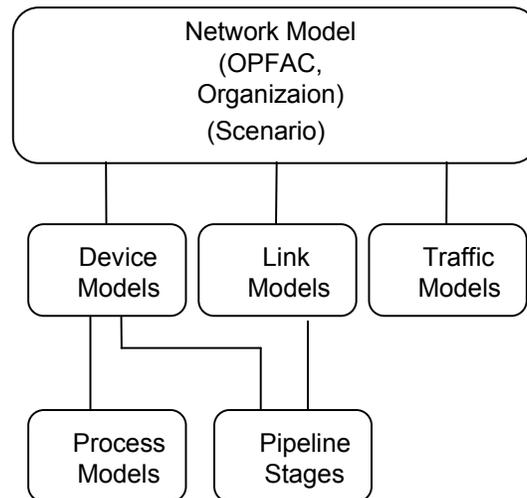


Figure 2-7: The JCSS/OPNET Model Hierarchy

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 Standard Library (COTS) models and for all the JCSS models. The user can copy and modify this code to make the model development tasks easier. For more OPFAC and Org information, please see “JCSS 9.0 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 JCSS by JCSS 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 JCSS 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 JCSS’ Cisco 2514 router. This router is based on the OPNET Standard Library (COTS) Cisco 2514 model, with minor changes to make it compliant with JCSS.

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 JCSS. They do not have to be coded for each device, but simply laid out and connected in the node editor.

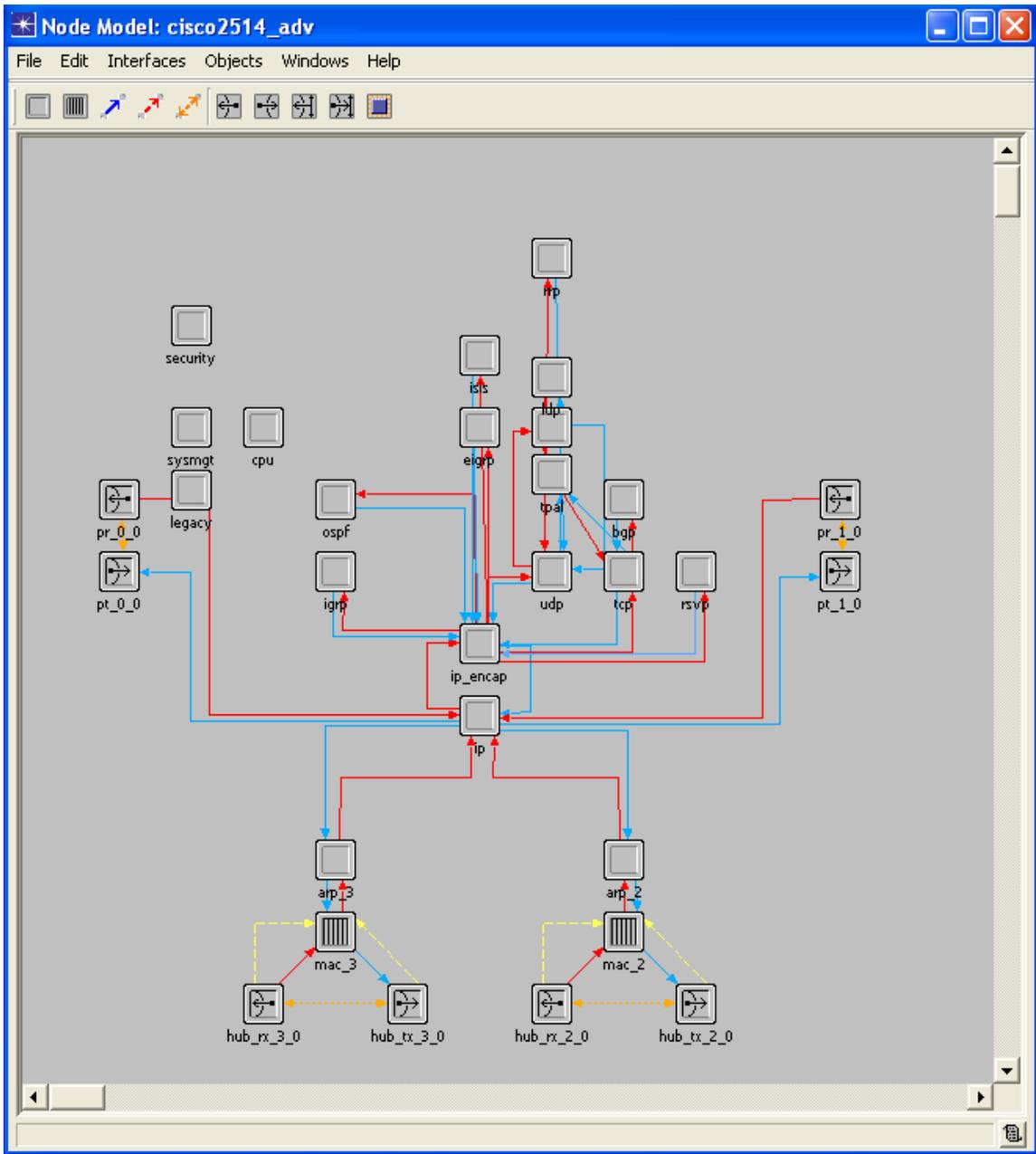


Figure 2-8: Editing a JCSS Cisco 2514 Router Model

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 PRC radio 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.

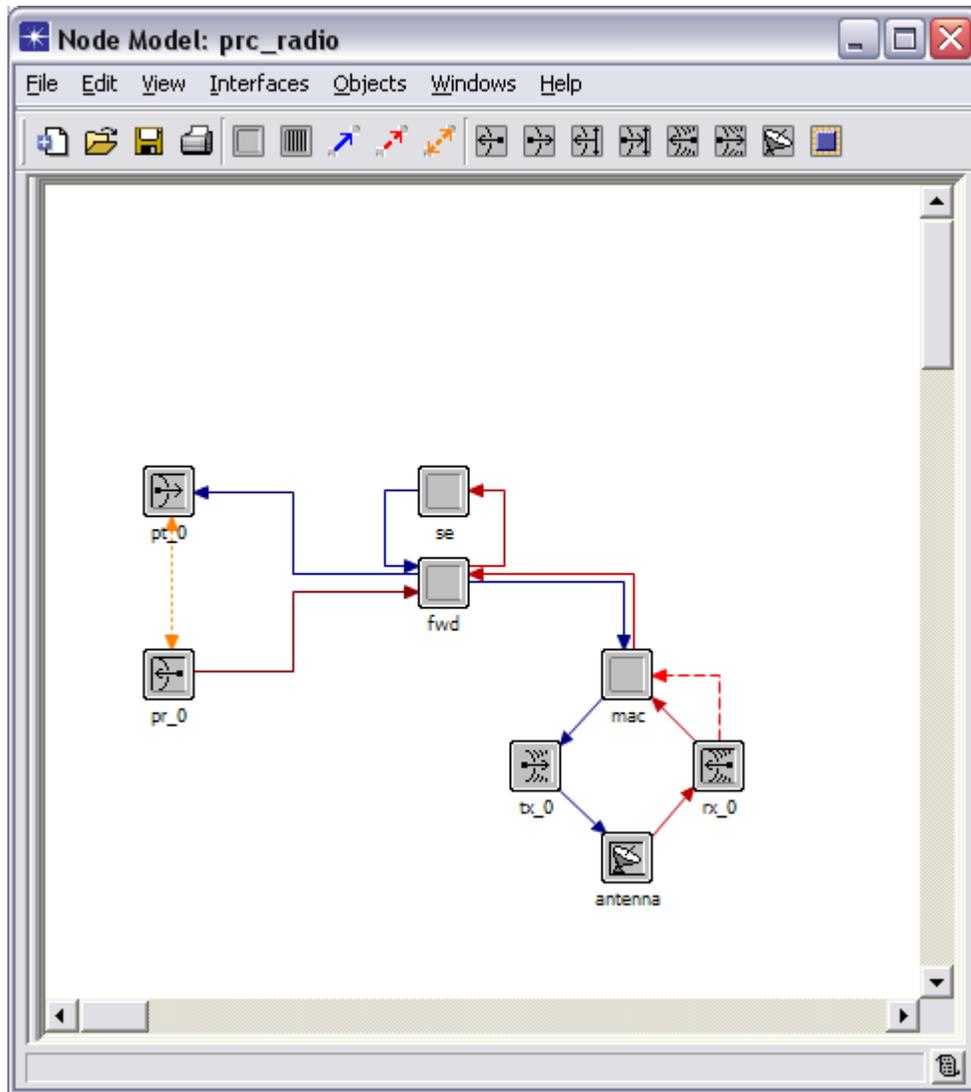


Figure 2-9: The Process Models Within the PRC Radio Model

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 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.

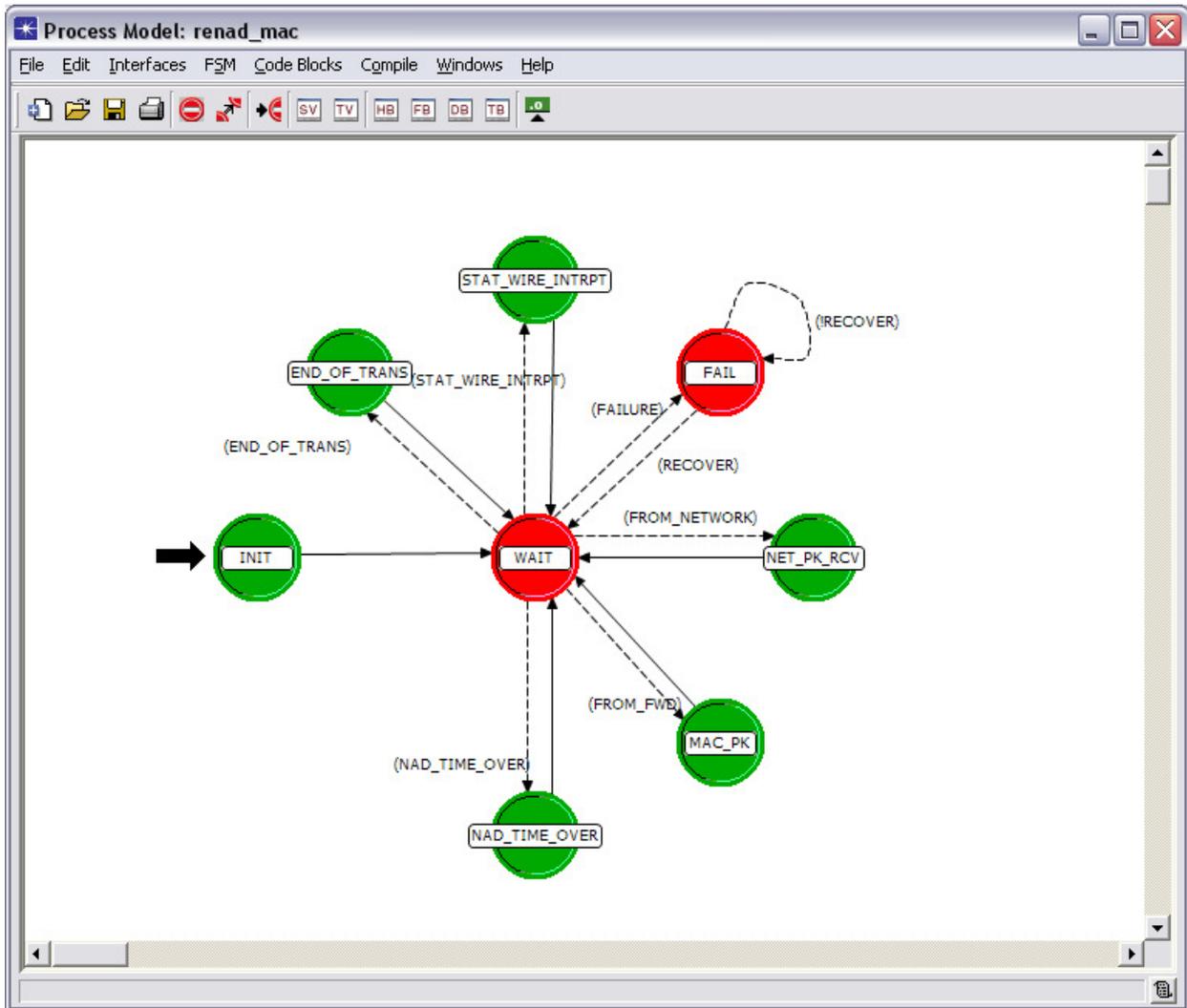


Figure 2-10: The Process Model Editor

There are three places where executable code can be invoked:

1. Upon entry to a state, called the Enter Execs
2. Upon exit from a state, called the Exit Execs

3. 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.

By double clicking into the Enter or Exit Execs, the user can be edit the code for these execs with a text editor (as shown in Figure 2-11).

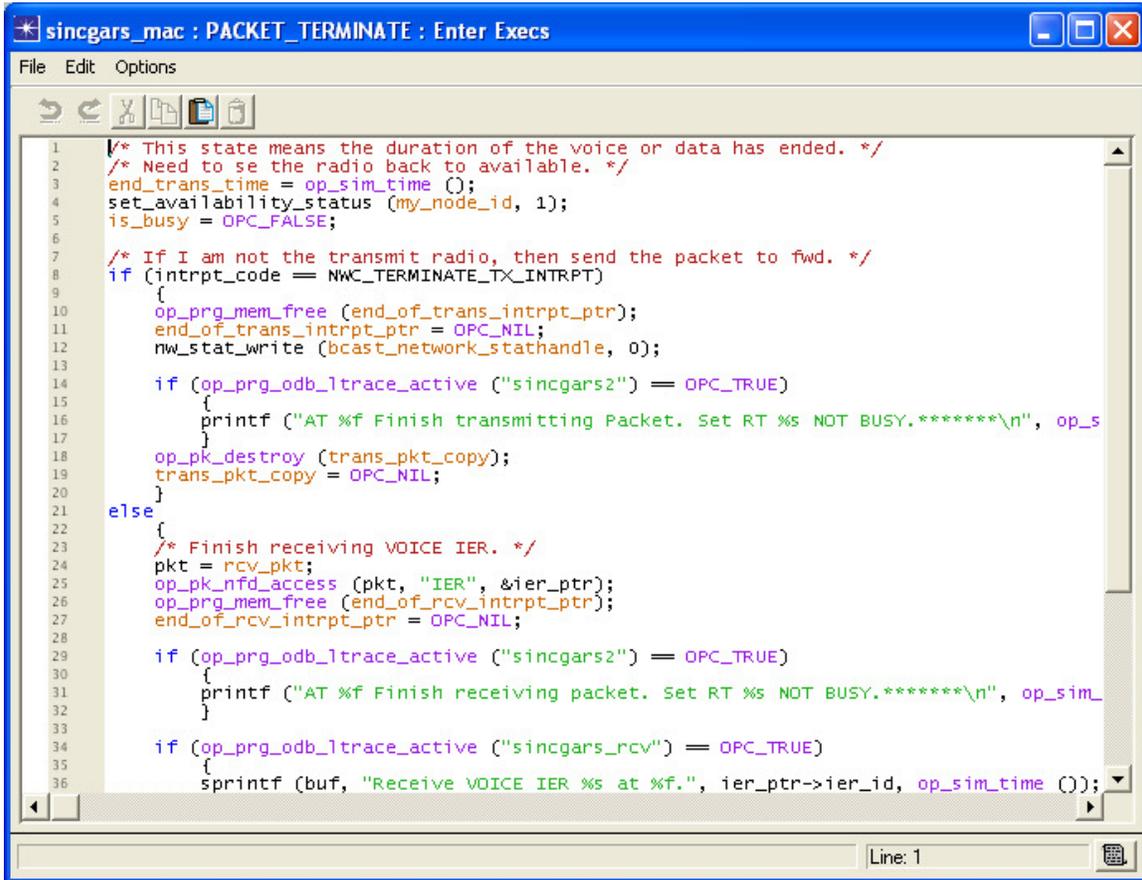


Figure 2-11: Editing C Code in the Process Model Editor

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).

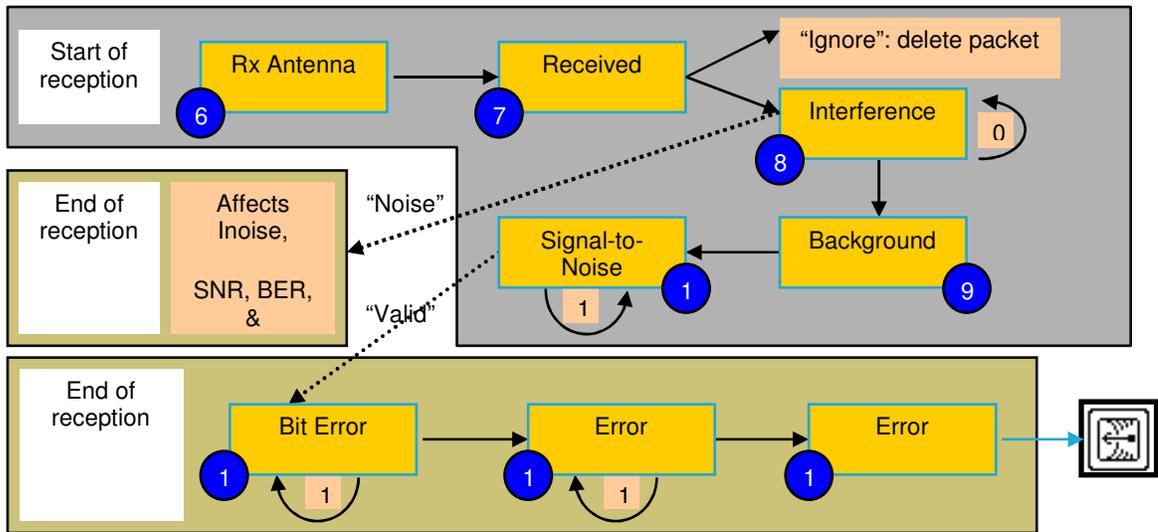


Figure 2-12: Receive Pipeline Stages

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 JCSS requirements for modeling links.

2.1.3.5 Traffic Models

JCSS 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, JCSS 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 JCSS 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.

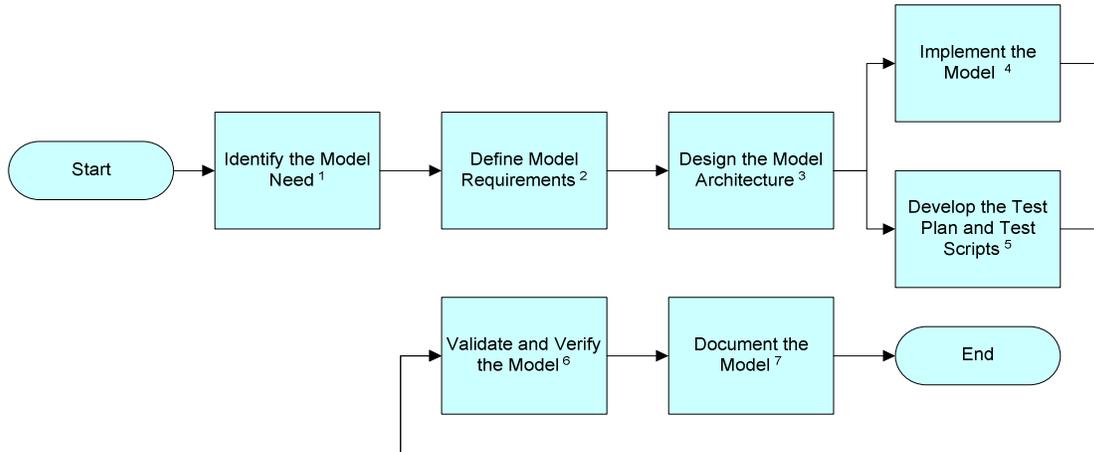


Figure 2-13: JCSS Model Development Life Cycle

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 JCSS MODEL DEVELOPMENT

This section provides the guidance and requirements for creating traffic and communications device models compliant with the JCSS modeling architecture and which can interoperate with models in the JCSS 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 JCSS 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 JCSS model development considerations to the developer; and the third category applies to specific classes of JCSS model development. Each of these specific class subsections explains how to build a JCSS component model, and includes the following:

- Defines a JCSS 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 JCSS and OPNET Standard Library (COTS) classes
- Describes the JCSS and OPNET Standard Library (COTS) failure and recovery
- Describes device model measures of performance (MOP) and how to collect statistics
- Describes the JCSS model documentation standards
- Describes the device model construction process

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

3.1 TRAFFIC MODEL DEVELOPMENT PROCESS

There are five different types of traffic models that can be used in JCSS, each with their own use cases:

Table 3-1: Traffic Model Use Cases

Traffic Type	Typical Use Cases
IERs	<ul style="list-style-type: none"> • Access to deployed traffic in IER form (text files, etc) • Quick and easy way to do reachability analysis for your network • Traceability • Reports • Application packet level tracking
ACE	<ul style="list-style-type: none"> • Application is already deployed • Access to actual network traffic (packet traces) • Most accurate representation
ACE Whiteboard	<ul style="list-style-type: none"> • Know the behavior of application to be deployed • Test before deployment

Standard Application Model	<ul style="list-style-type: none"> • Study behavior of commonly used application models like email, http, etc
Traffic Flows (i.e., Demands)	<ul style="list-style-type: none"> • Load network with background traffic (unrelated traffic) to study your application model • Quick and easy way to do reachability analysis for your network • Capacity planning studies

For ACE, ACE Whiteboard, Standard Application Models, and Traffic Flows, please see the OPNET Modeler online documentation.

3.1.1 IERs

IERs are a Department of Defense (DoD) Core Architecture Data Model (CADM) standard. Essentially, an Operational IER is a single thread of a mission or operation which tells the planner:

- Who is sending the information?
- What information is being sent?
- Why the information is being sent?
- Who is receiving the information?
- How the information must be passed for the warfighter mission, process, or transaction to be completed successfully.

IERs are known as a high level traffic meaning that the user can specify an IER between units such as Organizations and Operational Facilities (OPFACs) instead of actual devices. The idea is to make the process easier and Defense specific by allowing the user to think of the network in military terms. At the highest level, Organizations can be looked at as military organizations (such as DISA, a division, etc.) which are made up of a number of assets. These assets would include common military buildings, vehicles, and communications devices. Organizations are then made up of a number of OPFACs, which can be thought of as the assets for the Organization (i.e., the buildings, tanks, planes, soldiers, etc.). The OPFACs then hold all of the communication devices which make up that asset.

By leveraging this hierarchy, the user can deploy various forms of traffic to interact with these devices through the use of IERs. IERs allow the user to define information about traffic messages (such as interarrival time, classification, start time, end time, size, perishability, etc.) which traverse from a producer OPFAC to a consumer OPFAC. The important concept of IERs is that they also allow the user to deploy traffic at a high level (i.e., between a tank and a plane) without having to fully understand the underlying communication devices inside the OPFAC. The user only needs to define that a message of a certain size needs to be delivered at a certain time between two assets. For example, an “attack” command can be realistically modeled as a message going from the command center to a plane.

Using this workflow, the user determines the capabilities of a particular asset and can make adjustments as necessary based on the load required by the network users. The overall load comes from the databases and lists of IERs which have been created by planners and users.

Also, as everyone is using common terminology, the network planner can share the created network topology with other non-technical users who are familiar with the higher level Organizations and OPFACs. In addition, IERs can be used in other enterprise architectures such as the Department of Defense Architecture Framework (DoDAF). This commonality helps facilitate planning and acquisitions processes.

As JCSS is a Defense oriented tool, a subset of the IER functionality has been added to support IER workflows. This functionality provides traceability, statistics, reports, and packet level tracking. IERs can be defined in the Scenario Builder or can be imported from text files and IER report. Further, the use of the “IER_Demand” traffic flow model represents IER and thread traffic on same lines as COTS application and flow traffic. One traffic flow object is used to represent one IER or one segment of an IER thread. The following screen shot display the IER Demand Model attributes:

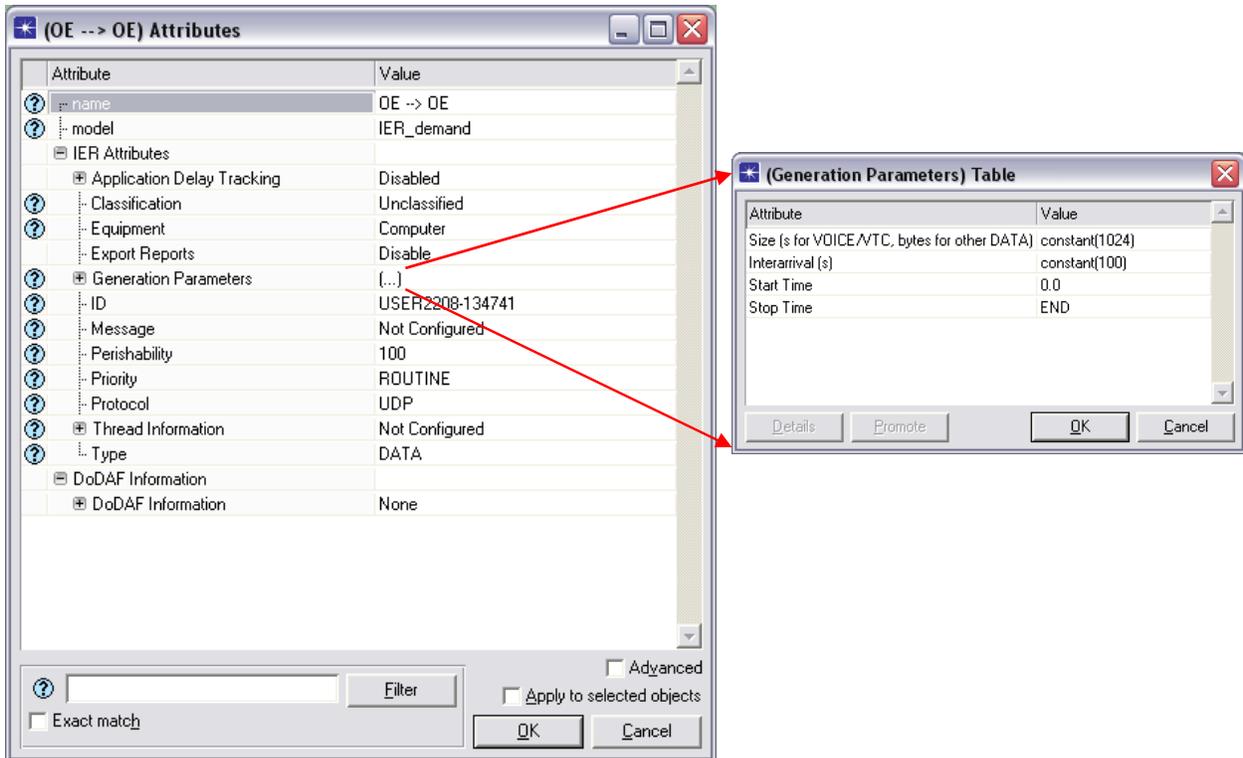


Figure 3-1: IER Demand Model Attributes

Threaded IERs on the other hand are represented as sequence of IERs with each IER having some specific firing conditions. These can be modified in the Thread Information sub-attribute:

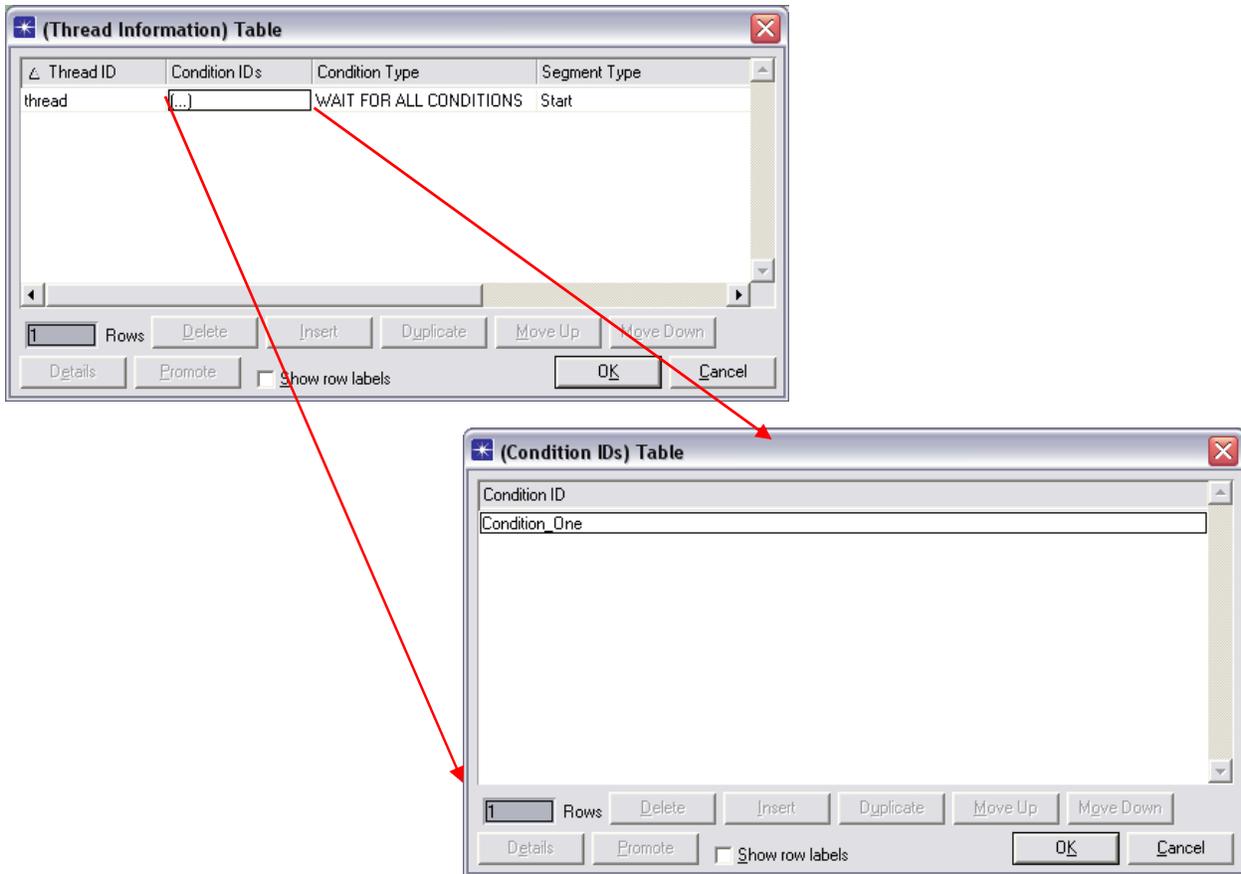


Figure 3-2: IER Thread Information Attributes

3.1.2 Operational Element

The Operation Element (OE), sometimes known as the IER Manager, manages IER and thread traffic for each OPFAC in a scenario. There are several behaviors for the OE node:

- An OE node should be placed in every OPFAC that sends or receives IER/thread traffic in the scenario. This is because the OE node regulates all of the IERs for a particular OPFAC as IERs are a high level traffic. This means that the OE node keeps track of when the IER/Thread is fired, what devices are selected, etc. Also, the OE writes IER/Thread statistics and reports, and interfaces with COTS application packet level tracking (also known as Application Delay Tracking, ADT).
- An OE node can have IER traffic flow objects attached to it (i.e., a user can specify an endpoint of the IER object as an OE). When this is the case, a device is “Auto-Selected” for that endpoint. This means that the OE node will randomly pick a device based on its ability to handle the IER. The rules for this selection can be found in the IER Firing Rules node and are based on decision table rules, device availability, and random selection.
- If no device is available in the network, the OE must retry firing the IER until a device is found or the maximum number of retries is exhausted. The maximum number of retries is configured on the *IER Firing Rules* node.

- The OE translates the IER definition information using the IER traffic flow attributes to an interrupt that can be sent to the System Element (SE) application layer of a device. The device can then read the IER information and translate it into packets.
- The OE controls HLA operations from the HLA Commander software that can be installed with JCSS. This includes OPFAC movement and firing of IERs from a third party simulator.

3.1.3 System Element

Each end-system needs an SE module to support IERs. The functions of an SE module include interfacing with the OE node, initiating/receiving lower layer signaling (such as the transport layer), generating actual application messages and sending them to a lower layer, and receiving application messages meant for the end-system.

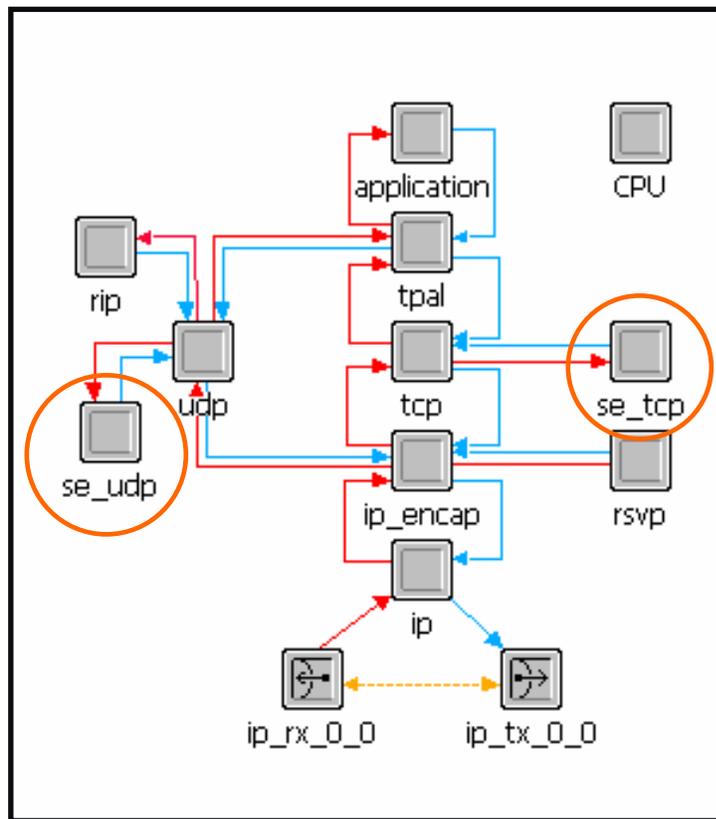


Figure 3-3: Node Model with SE modules

Examples include (but are not limited to): se_trafgen, se_udp, cs_voice_se, prc_se

3.1.4 OE – SE Interaction

The OE node sends remote interrupts to the SE which, in turn, generates application messages to be sent across the network. Also, the SE module informs the OE of traffic reception as it receives other application messages from the network. Using this information, the OE is then able to write

statistics and reports to be analyzed by a user of the model. During custom JCSS model development, interfacing with an OE is only required if a new application layer is needed or an existing application layer is modified. If working on lower protocol layers such as layers 1 through 4, one of existing SE model can be utilized instead.

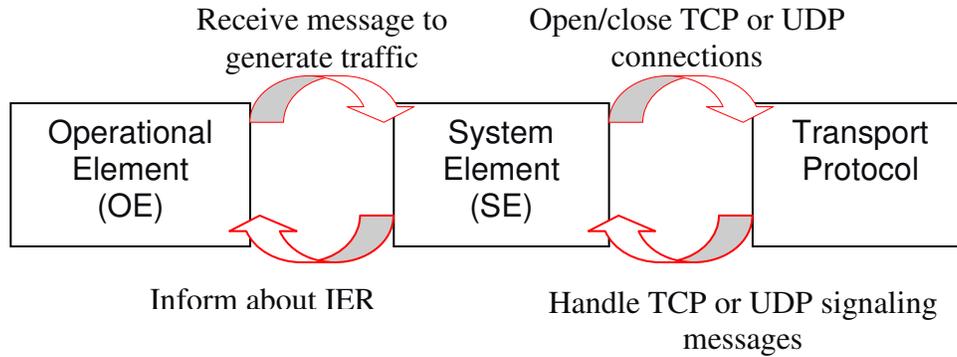


Figure 3-4: OE-SE Interaction

3.1.5 DoDAF Integration

The Department of Defense (DoD) Architecture Framework (DoDAF) provides a standard for description, development, presentation, and integration of systems for the DoD. DoDAF defines a standard way to organize an enterprise architecture or systems architecture into a set of complementary and consistent views. All major U.S. Government DoD weapons and information technology system acquisitions are required to develop and document an enterprise architecture using the views prescribed in the DoDAF. While it is clearly aimed at military systems, DoDAF has broad applicability across the private, public, and voluntary sectors around the world, and represents only one of a large number of systems architecture frameworks.

As JCSS provides a robust set of military communications modeling and simulation tools, it is natural to envision an interface between JCSS and DoDAF that can facilitate building a simulation model of a communication system defined in DoDAF. Also, as many of the JCSS users are military sponsored, they are mandated to use the DoDAF standard to compare and present systems which can be used in the battlefield.

Since DoDAF is a large standard consisting of many products, a JCSS interface to cover all of these products cannot be realized immediately. Therefore, a decision was made to initially focus on integrating a limited number of products. Products were selected based on their relationship with current JCSS features. This led to the first interface supporting DoDAF OV-3 and SV-6 products as they closely resemble the JCSS IERs workflow.

As part of this work, a new *DoDAF Information* attribute was placed on the IER traffic flow object. To modify this attribute directly, the user can right click on the IER traffic flow object and select the **Edit Attributes (Advanced)** menu.

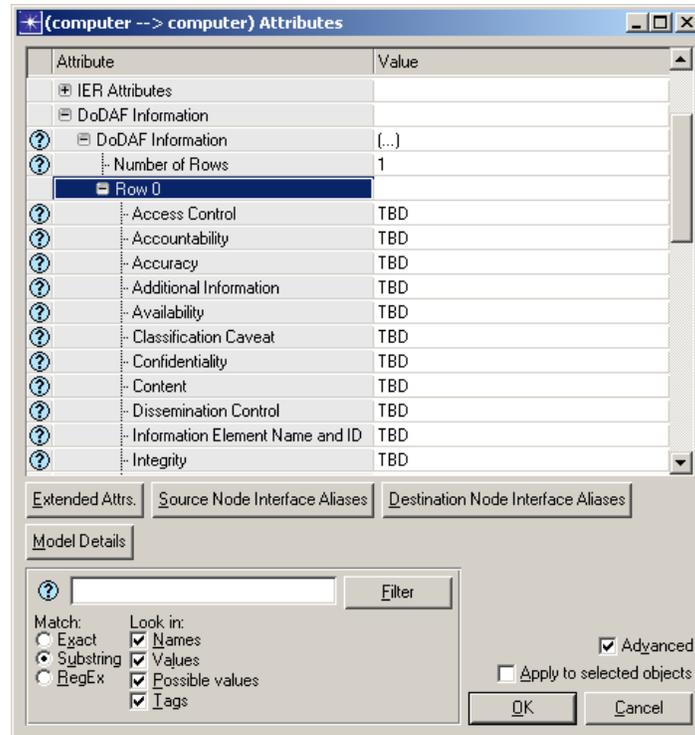


Figure 3-5: Viewing the DoDAF Information Attribute

Each sub-attribute of this attribute corresponds to a DoDAF attribute in an OV-3 or SV-6 View. In some cases, the sub-attributes inside the *IER Attributes* attribute are used instead of adding a new *DoDAF Information* sub-attribute (as there is overlap). To change the value, a user can directly insert a string into the attribute and click **OK** to save the change. However, it is strongly recommended that the user utilize the **DoDAF Integration** dialog box (explained in more detail in the JCSS IER Models User Guide) to help configure the DoDAF attributes properly.

Once the DoDAF information is set, the Capacity Planner (CP) and Discrete Event Simulation (DES) simulations can use these values to help send traffic through the network.

3.2 COMMUNICATIONS DEVICE AND PROCESS MODEL DEVELOPMENT PROCESS

The development process is the second of three phases in the JCSS 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.

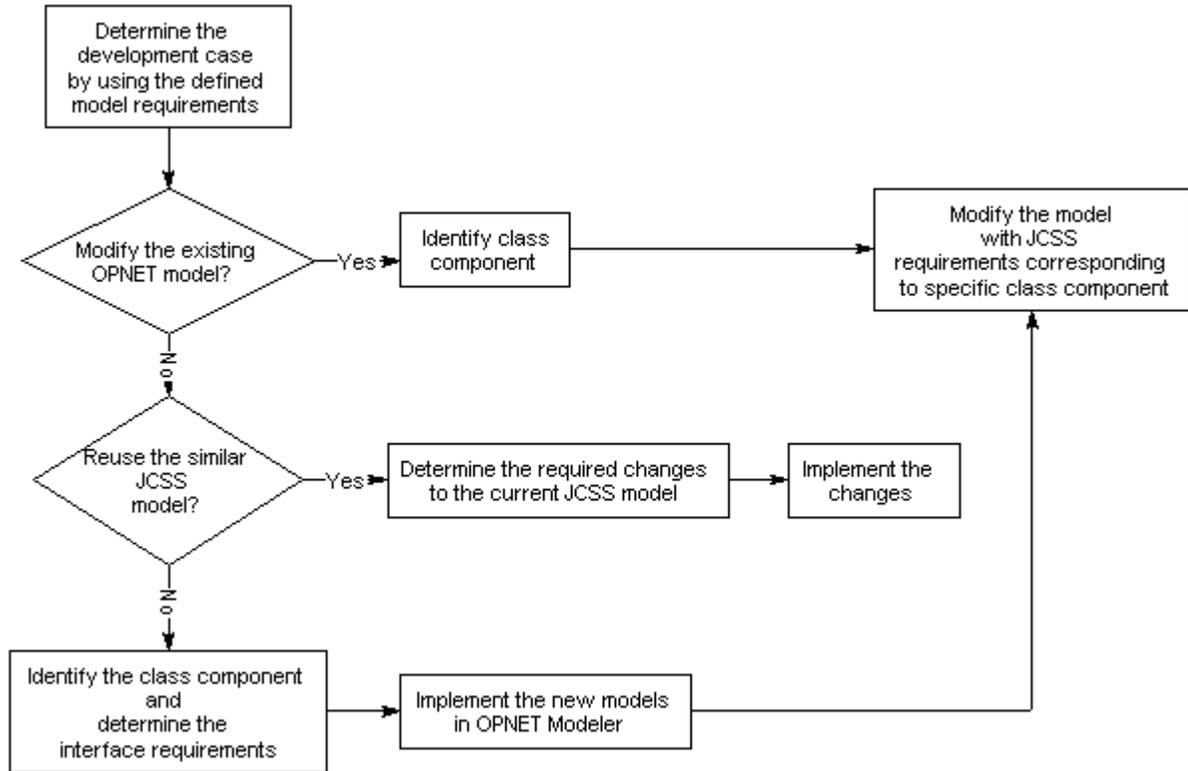


Figure 3-6: High-Level Model Development Process

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 JCSS compatible
- Surrogating from the existing JCSS 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 JCSS Compatible

In this case, the scope is to convert an existing OPNET model into a JCSS 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 JCSS.
- If the device used a COTS traffic model, then it will work as-is in JCSS using DES only.

- If the device “wants” to use the JCSS 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 JCSS 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 JCSS Model

In this step, the developer re-uses a similar JCSS model as the foundation to construct the new model. The key considerations while surrogating from the existing JCSS 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). Regardless, it is recommended that the user deploy Device Creator to create a baseline of any new model created. Refer to section 3.5 for more information on Device Creator.
- Perform all the steps in the “Modifying the Existing OPNET Model to Be JCSS Compatible” subsection.

3.3 MODEL INTEROPERABILITY ISSUES

Before development of any device models in the JCSS 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 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 This statement assumes that the behavior of the OE is similar to the one present in the JCSS 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 JCSS model *nw_ethernet_wkstn.nd.m* has two specialized interfaces—one that supports TCP transport protocol and one that supports UDP. It has a separate implementation of the SE for either of these protocols.

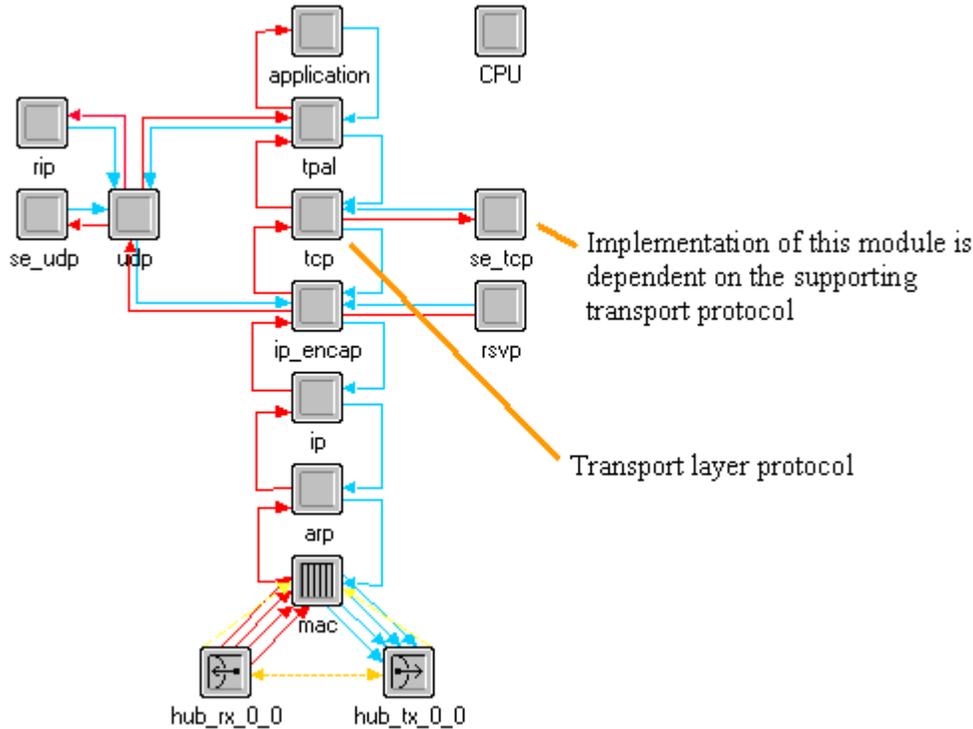


Figure 3-7: Protocol Dependency (e.g., Ethernet Computer Model)

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 JCSS standard models such as Promina, circuit switches, satellite terminals, and the like. Refer to section 3.14 (API and Framework) for more information.

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/processes²). 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 registry³ use can be seen in the JCSS 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.

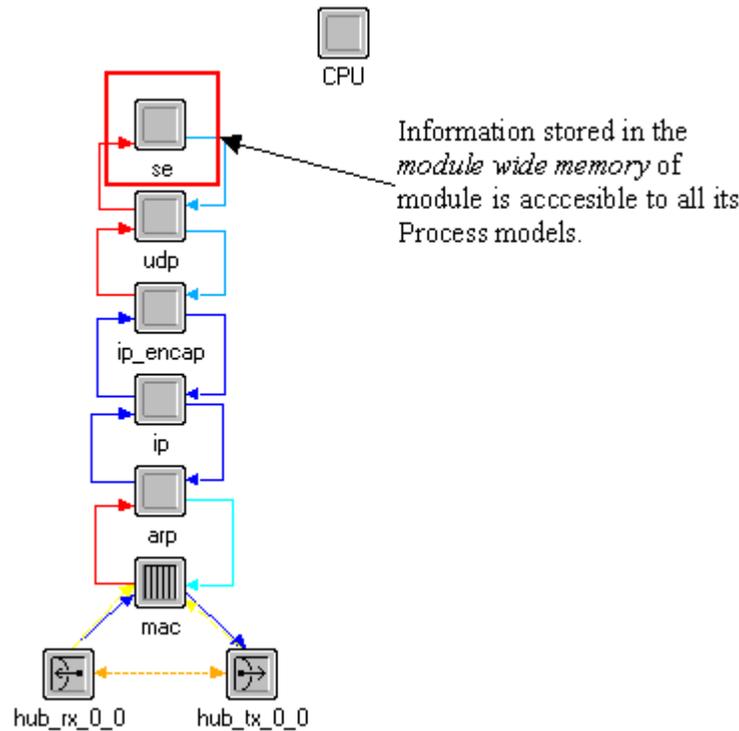


Figure 3-8: Module-Wide Memory (e.g., Ethernet Computer Model)

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 any regular global variables declared in the header block of one process and can be used by other processes. The user is discouraged from using global variables as they are not an ideal or reliable method of sharing information between processes. Global variables also present problems for parallel simulations. It is recommended to use process registries, ICIs, or packets to share the information instead.

However, if it is necessary to create global variables, 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:

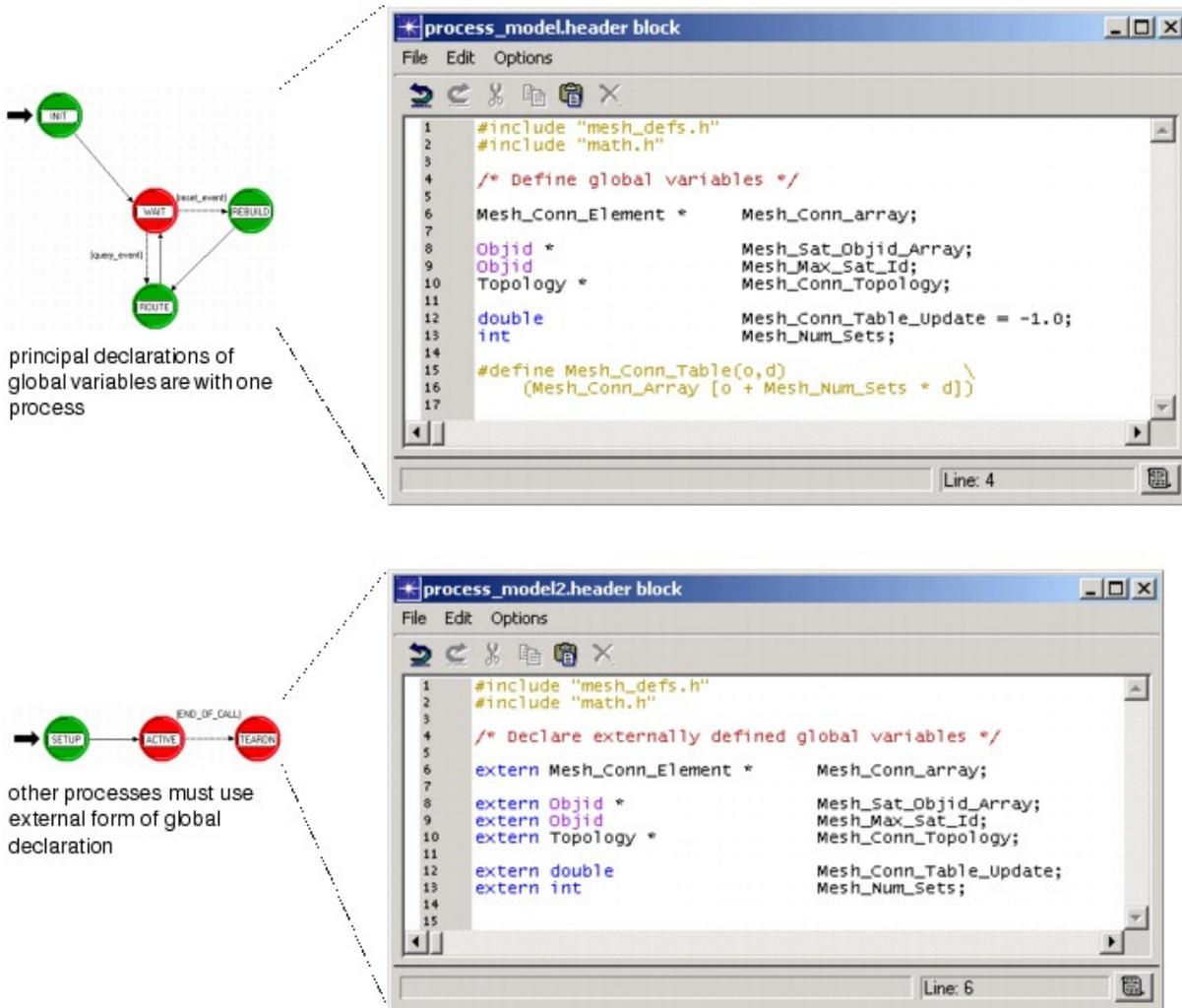


Figure 3-9: Declaration of Global Variable in Two Process Header Blocks

3.3.3 Communication Aspects

This subsection introduces 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/JCSS 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 JCSS 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 `pro_hello_pk` and `pro_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 JCSS environment that is primarily used by the JCSS Scenario Builder to determine if an external link connected between two devices supports the assigned ports (transceivers). Refer to the *JCSS 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 3-10. This will apply to interrupts received at the source state of the transition that the process does not know how to handle.

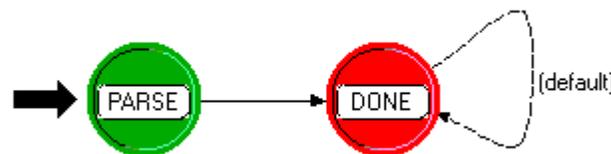


Figure 3-10: Default Interrupt Handling

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 JCSS 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 JCSS standard models.

3.3.4 Self-Description Issues

Every model produced in JCSS holds some information regarding how it can interface with other model types. JCSS 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 JCSS models. If the custom models do not support the same packet format information as JCSS models, then self-description information based on the developed models will have to be developed.

3.3.4.1 Port and Port Groups

The JCSS 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 JCSS models, each port category must have a self-description port object. For example, MRC-142 (JCSS 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).

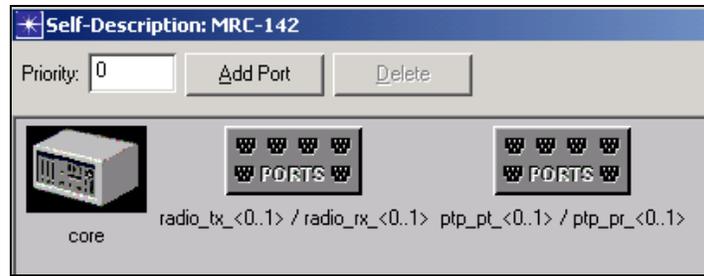


Figure 3-11: Self-Description Port Objects

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 U for details.

3.3.5 Versioning Issues

To upgrade the models to a new JCSS standard model library, users need to force-compile all their models with the new header files. JCSS supports backward compatibility. For example, models developed on Version 14.5 can be applied on Version 15.0, 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 JCSS 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=<JCSS_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 JCSS CAPACITY PLANNING COMPLIANCE REQUIREMENTS

CP in JCSS 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. Because of the use of analytic techniques and shortest-hop routing, the results from CP can be different from those found after running DES. This subsection is of interest when:

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

3.4.1 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 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, or substitute model, 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 3-12. 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 3-2.

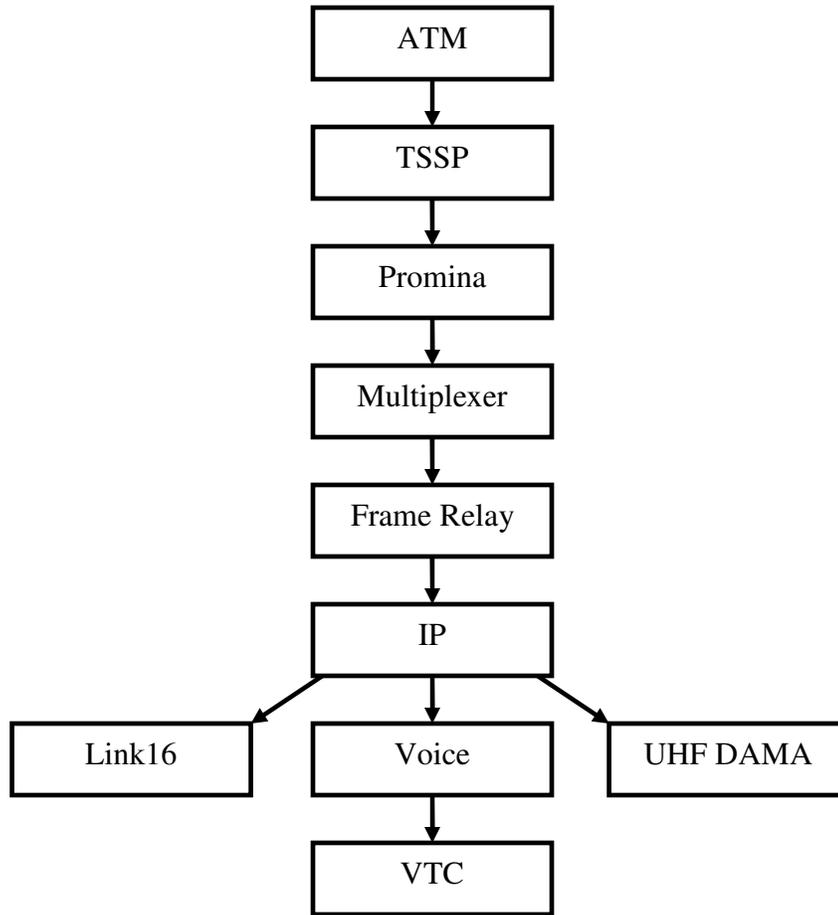


Figure 3-12: CP Layers

Table 3-2: Properties to Determine CP Layer

Layer	Attribute	Attribute Location	Acceptable Value
ATM	equipment type	on device	generic
	interface type	self-description	contains atm:
	machine type	self-description	router or switch
	interface type	self-description	contains atm:
TSSP	equipment_type	on device	Promina
	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
Link16	JRE Links	on device	n/a (attribute must simply exist)
	equipment type	on device	JTIDS
UHF DAMA Terminal	Platform ID	on device	n/a (attribute must simply exist)
	Platform Terminal Logon Frame	on device	n/a (attribute must simply exist)
UHF DAMA SRAP	model	on device	UHF_SATCOM_SRAP
Voice	equipment type	on device	generic or phone or radio or JTIDS or Media Gateway
	interface type	self-description	contains circuit_switched:Voice_LAN or contains circuit_switched:Voice_WAN
	equipment type	on device	is not Promina and is not Encryptor and is not Multiplexer
Video Teleconferencing (VTC)	equipment type	on device	generic or VTC Terminal
	interface type	self-description	contains circuit_switched:Voice_LAN or contains circuit_switched:Voice_WAN
	equipment type	on device	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 3-2 will cause unroutable demands.

An example of some of the information that the CP will use in JCSS 9.0 is as follows:

- “eplrs” interface type on EPLRS ports
- “promina:WAN” interface type on Promina-100

- “multiplexer:mux_aggregate” interface type on multiplexer ports like FCC100v7
- “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

During CP routing, failure/recovery of a subnet, device, or link will be taken into account. To use failure/recovery on an object, the user should use the “Edit JCSS Attributes” right-click menu or manually configure the “Failure Recovery Configuration Node” located in the Configuration OPFAC. Essentially, Failure/Recovery centers around setting the “condition” toggle attribute located on all scenario objects. In a simulation, if the “condition” attribute is set to “disabled” on an object, the object is considered failed. Therefore, the user does not need to make any additional modeling changes for Failure/Recovery to work in CP. For more information on using the Failure/Recovery feature, consult the JCSS User Manual.

3.4.3 Logical Views

Logical Views provide the ability to filter the network so that a user can look at particular layers or technologies more closely. Logical Views will work for many of the JCSS device technologies such as IP, Tactical Radios, Satellites, etc. and utilizes CP graphs and self-description information (using the interface class and machine type) to determine the logical connections between devices. Therefore, as CP is used to create the Logical Views, when a model is created that works with CP it usually will work with Logical Views also. Logical Views can then show the logical connection information visually to the user.

It is important to note the orange, logical links which are displayed the example Logical View below (Figure 3-13). In the example case, this logical link is abstracting the layer-2 layer and showing IP connectivity between end-stations and routers. What this link shows to the user is that these routers can “talk” to each other through IP even though they are not physically connected to each other. Also, logical links show information such as the physical links which make up the connection, the overall data rate for the entire logical link, etc.

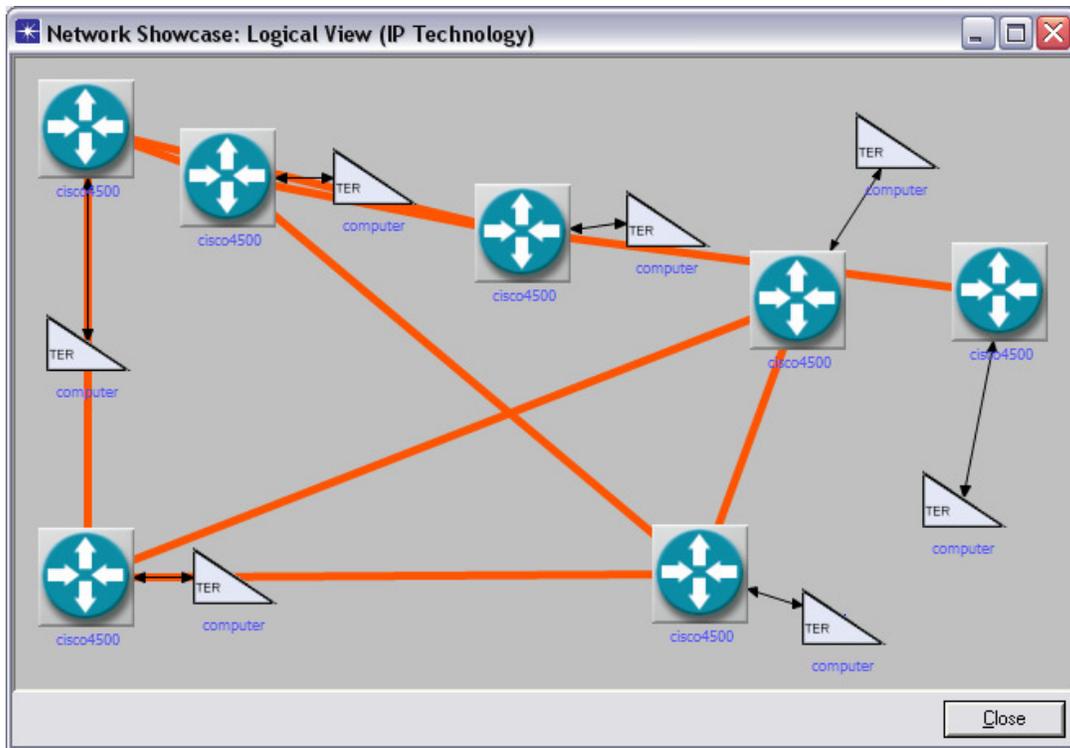


Figure 3-13: Example Logical View

3.4.4 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. This is not the case, for example, if the device adds a certain amount of overhead, then the use of a surrogate is warranted. To surrogate a model means that the user should take an existing model that works with the required feature, and derive a new version of that model to behave similarly to the newly created device model. This derived model will load the network properly in the analytical simulation, but should not be used in detailed simulations such as DES. For DES, the user should use the newly created device model instead. In the case of message size overhead, 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.”

3.4.5 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 the Mobile Subscriber Equipment (MSE), 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. 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.

3.5 METHODOLOGIES FOR CREATING DEVICES FOR JCSS WITH DEVICE CREATOR

In addition to using the built-in model objects and creating derived models, the user can create new device models using the Create Custom Device Model operation.

Device Creator allows the user to create several different types of network components, including routers, bridges, hubs, workstations, servers, and switches. The following table lists the device categories and the technologies or device types supported by Create Custom Device.

Table 3-3: Supported Device Classes

Device Category	Supported Technologies or Device Types
Bridge	<ul style="list-style-type: none"> • Ethernet (10BaseT, 100BaseT, 1000BaseX) • FDDI • Token Ring
Cloud	<ul style="list-style-type: none"> • ATM • Frame Relay • SLIP • Multiprotocol

Device Category	Supported Technologies or Device Types
Fibre Channel Devices	<ul style="list-style-type: none"> • Hub • Switch • Endnode • Multihomed Endnode
Firewall	<ul style="list-style-type: none"> • ATM • Ethernet (10BaseT, 100BaseT, 1000BaseX), Ethernet LANE, and EtherChannel • FDDI • Frame Relay • SLIP • Token Ring and TR LANE
Hub	<ul style="list-style-type: none"> • Ethernet (10BaseT, 100BaseT, 1000BaseX) • FDDI • Fibre Channel • Token Ring
LAN Model	<ul style="list-style-type: none"> • Shared • Switched
Mainframes	<ul style="list-style-type: none"> • ATM • Ethernet (10BaseT, 100BaseT, 1000BaseX), Ethernet LANE, and EtherChannel • FDDI • Frame Relay • Fibre Channel • SLIP • Token Ring and TR LANE • Wireless LAN
Multihomed Host	<ul style="list-style-type: none"> • Client • Server
Multiplexer	<ul style="list-style-type: none"> • Promina
Multi-Service Switch	<ul style="list-style-type: none"> • Circuit Switched • Ethernet (10BaseT, 100BaseT, 1000BaseX) and EtherChannel • FDDI • Frame Relay • SLIP • Token Ring • Voice over ATM
Router	<ul style="list-style-type: none"> • ATM • DOCSIS • Ethernet (10BaseT, 100BaseT, 1000BaseX), Ethernet LANE, and EtherChannel • FDDI • Frame Relay • SLIP • Token Ring and TR LANE • Wireless LAN

Device Category	Supported Technologies or Device Types
Switch	<ul style="list-style-type: none"> • ATM • Ethernet (10BaseT, 100BaseT, 1000BaseX) and EtherChannel • FDDI • Fibre Channel • Frame Relay • LANE • Multiprotocol Switch • SSP • Token Ring • UMTS RNC
Vendor Device	<ul style="list-style-type: none"> • 3Com • Avici Systems • Bay Networks • Cabletron • Cisco • eXtreme Networks • Fore Systems • Foundry Networks • Hewlett-Packard • Juniper Networks • Lucent (Ascend) • Newbridge • NEC • Nortel Networks

Models created with the device creator have several advantages:

- They make it easy to configure new models for specialized needs and integrate them into your network models.
- They create both a derived model and a base model, so that you may use either depending on the modeling requirements.
- They support inheritance, so that changes to the original (parent) models can automatically affect all models derived from it.
- Can be used as a baseline model so additional customizations (such as new modules, process models, etc.) can be easily added. All interfaces, modules, self-description, etc. will be properly configured for the user.

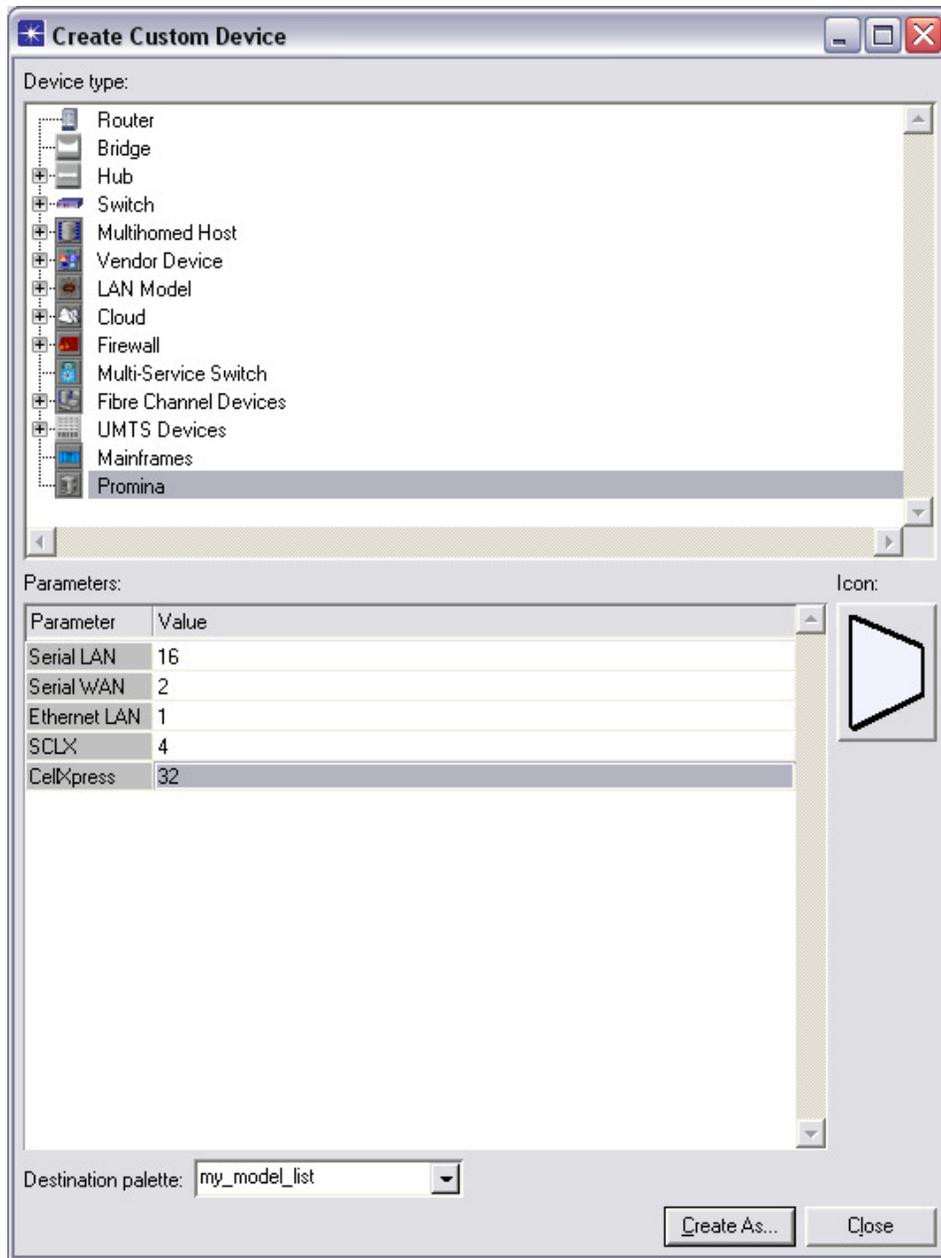


Figure 3-14: Create Custom Device Dialog

3.5.1 Model Names

As with derived models, the user can give a model any name that does not duplicate the name of an existing base or derived model of the same type. For more information, see the OPNET Modeler online documentation.

3.5.2 Creating a Custom Device

When creating a new model with Device Creator, you must set several argument options to specify your model appropriately. Many of these arguments (such as model name, model list, and icon name) are the same for all model types, but depending on the model class and the protocol you select, you may need to specify other arguments. These arguments, their values, and a description of each are listed in the following table.

Table 3-4: Device Class Arguments

Argument	Value	Description
ATM	ports	Maximum number of physical ATM links the model can support
ATM QoS A, B, C or D Buffer Capacity	cells	Buffer capacity for class A, B, C, or D traffic
Bridge Frame Service Rate	bits/second	Rate at which a bridge can process frames
Bridge Protocol Data Unit Service Rate	packets/second	Rate at which a bridge can process BPDUs
Buffer Capacity	bits	Maximum queue size of a port
Client Custom Application	Off Sample Load	Client custom application load generated by the model
Client Database Application	Off Low Load Medium Load High Load	Client database application load generated by the model
Client E-mail	Off Low Load Medium Load High Load	Client e-mail application load generated by the model
Client FTP	Off Low Load Medium Load High Load	Client FTP application load generated by the model
Client HTTP	Light Browsing Heavy Browsing Searching Image Browsing	Client HTTP application load generated by the model
Client Remote Login	Off Low Load Medium Load High Load	Client remote login application load generated by the model
Client Start Time	seconds	The time the client process begins to send traffic
Client Video Conferencing	Off Low Load Medium Load High Load	Client video conferencing application load generated by the model

Argument	Value	Description
Client X-windows	Off Low Load Medium Load High Load	Client X-windows application load generated by the model
Ethernet	ports	Generic ethernet device that can be used for 10BaseT, 100BaseT, and 1000BaseX ethernet links
FDDI	ports	Maximum number of physical FDDI links the model can support
Frame Relay	ports	Maximum number of physical Frame Relay links the model can support
Icon Name	N/A	User-defined icon name
IP ATM Maximum Data Rate	bits/second	Data rate specification for creating traffic contract used to set up the ATM connections
IP Forwarding Rate	packets/second	Number of packets an IP router can forward per second
Model Name	N/A	User-defined model name. This value defaults to my <device_type>.
Model List	N/A	User-defined model list. This value defaults to my_model_list.
Port Count	ports	Number of ports on the model
Server Configuration Table	N/A	Services supported by this server
SLIP	ports	Maximum number of SLIP links the model can support
Switch Backplane Speed	packets/sec	The rate at which switch backplane (the internal switching bus) operates. It governs the time it takes for an incoming packet to reach the switch processor.
Switching Rate	frames/sec	Number of incoming frames that can be forwarded
Switch Port Switching Speed	packets/sec	The rate at which packets are switched from the switch processor to the appropriate output port
Technology	Ethernet FDDI Token Ring	The protocol that the device type is based on
Token Ring	ports	Maximum number of physical Token Ring links the model can support
TPAL ATM Maximum Data Rate	bits/second	Data rate specification for creating traffic contract used to set up the ATM connections
Transport Address	TPAL address	TPAL address of the node. This address must be unique for each node

3.6 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 3.9. 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. A global repository stores all IER and thread related information (parsed from the IER demands) which is accessed by keys. Each device (including all Oes) can refer directly to each IER by keys located in the *ier_info* ICI which is received through interrupt everytime an IER is generated. The keys are simple integer values that uniquely identify a given IER and IER instance. 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.

3.6.1 Attributes

Table 3-5 gives the minimum set of attributes that an end-system device must have.

Table 3-5: JCSS Attributes for an 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; JCSS ships with a <i>classification.ad.m</i> 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

3.6.2 Self Descriptions

JCSS uses values from model Self Descriptions in many of its features. JCSS uses the value of the “machine type” characteristic in a node self-description to categorize the node. For example, the “machine type” is used by the Link Deployment Wizard and the Circuit Deployment Wizard. It is also used by the Logical Views to determine the technology used by the node (IP, radio, etc).

The Link Deployment Wizard uses the values of the “interface type” characteristic in node self descriptions to determine if nodes may be connected to each other and what types of links the node supports. An “interface type” characteristic is usually defined on each port group in a node self-description.

JCSS uses the values of various other characteristics in self-descriptions. For example, the “Nodal Mode” characteristic specifies some capabilities of TSSP satellite models. JCSS also extracts the names and numbers of ports from node model self-descriptions for use in many features.

For more information about Self Descriptions, refer to the standard online OPNET documentation and Appendix U.

3.6.3 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 3-6 specifies the higher layer modules for a certain technology, and Table 3-7 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.

3.6.3.1 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 3-6. Please note this table is not exhaustive and additional modules can be found in the OPNET Modeler online documentation.

Table 3-6: Higher Layer Modules for an 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

3.6.3.2 Lower Layer Modules

The OPNET Standard (COTS) protocols shown in Table 3-7 can be used as lower layer modules. The process model in a module is specified in parentheses next to the name of the module. Please note this table is not exhaustive and additional modules can be found in the OPNET Modeler online documentation.

Table 3-7: Lower Layer Modules for an 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

Protocol Type	Required Modules
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 Protocol (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 3-6 and the modules needed for Ethernet from Table 3-7. The types of transmitters and receivers to be used depend on the physical layer of the device. Transmitters and receivers can be one of two types:

- Point-to-point
- Radio

Such an end-system device with TCP/IP over Ethernet point-to-point transceivers would appear as illustrated in Figure 3-15.

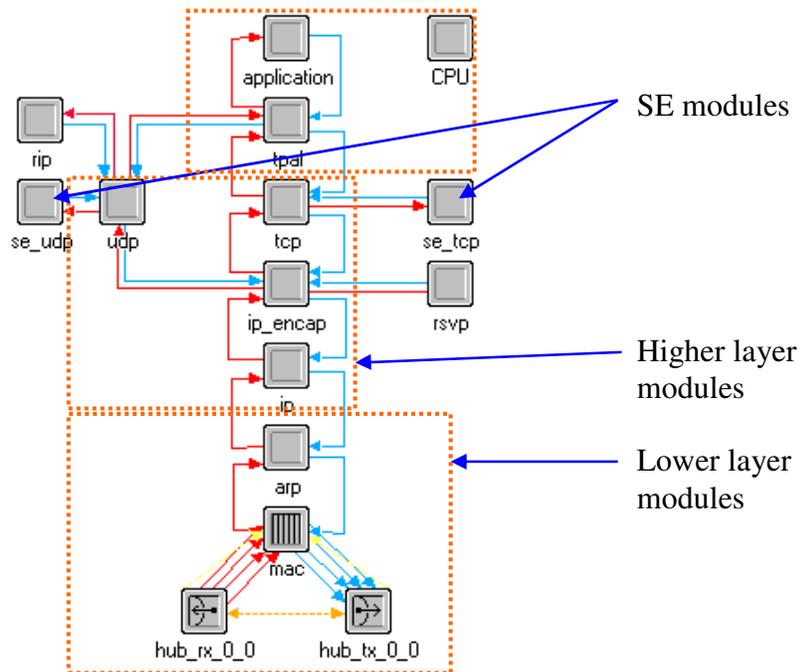


Figure 3-15: An Ethernet End-System Device—Node Model

For end-system devices with radio interfaces, refer to Subsection 4.10.

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 3-6 and Table 3-7. When combining modules from the two tables, sometimes it is necessary to connect them by an interface module, shown in Table 3-8. Please note that this table is not exhaustive and additional modules can be found in the OPNET Modeler online documentation.

Table 3-8: Interface Modules for an End-System Device

Higher Layer Protocol Stack	Lower Layer Protocol Stack	Interface Module Needed
TCP, UDP, IP	ATM	IPAL (ams_ipif_v4)
	ATM (with LANE)	arp (ip_arp_v4), LANE_IF (lms_lane_if_v3), LANE (lms_lec_v3)
	Ethernet	ethernet (ethernet_mac_v2), arp (ip_arp_v4)
	Frame relay	FRIPIF (frms_fr_ipif_v3)
	Serial	IP (ip_dispatch)

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 3-16.

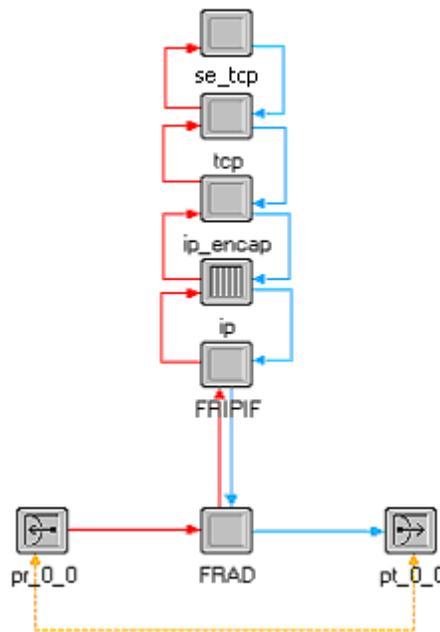


Figure 3-16: End-System Device with Frame Relay as the MAC Technology—Node Model

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 3-17. 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.

3.6.4 End-System Devices Categories

3.6.4.1 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.

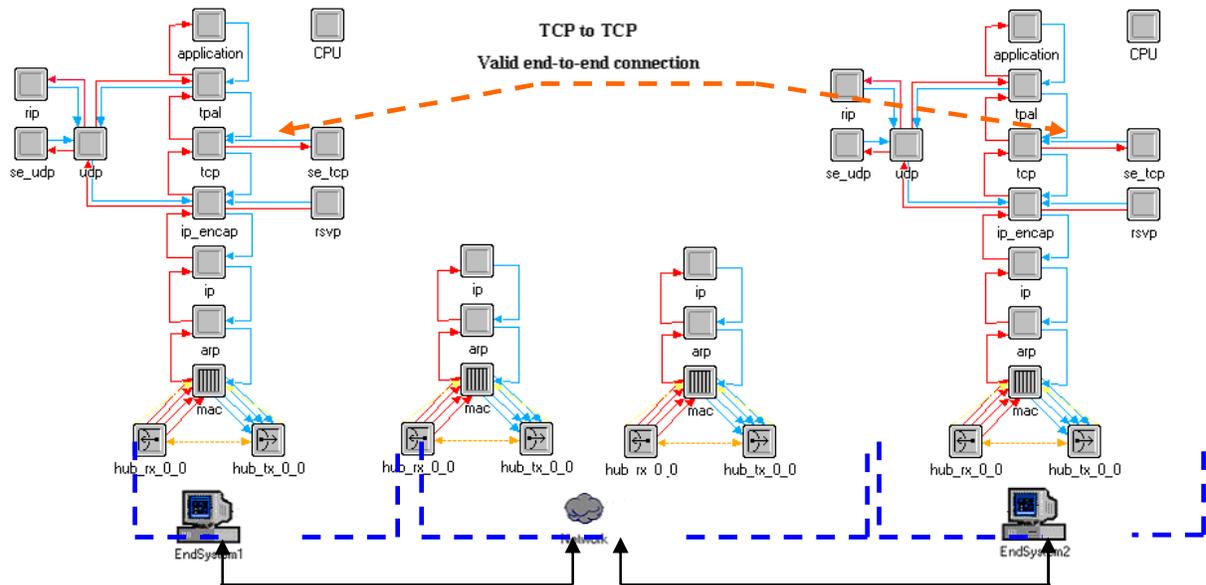


Figure 3-17: A Valid End-System to End-System Connection

3.6.4.2 Circuit-Switched Voice Traffic Only

If the device supports only voice calls, it does not need the network protocol stack. In JCSS end-system circuit-switched devices (e.g., phone), it sends out a call-setup packet (packet format *ctkswpkt*) that may cause intermediate network devices to reserve bandwidth on the links and intermediate devices for the duration of the call. Refer to Subsection 3.10.

If such a purely circuit-switched device connects to other packet-switched devices, such a configuration requires use of multi-service switches (see Figure 3-18). Again, refer to Subsection 3.10 for more details.

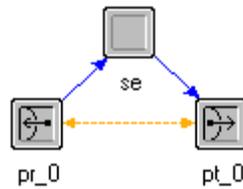


Figure 3-18: Example of a Circuit-Switched End-System Device—Node Model

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 3-19).

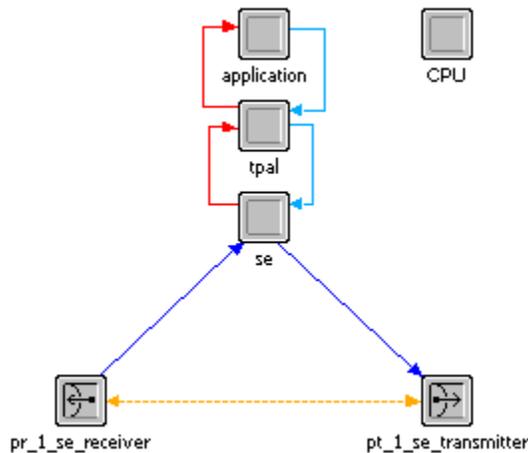


Figure 3-19: Example of a Circuit-Switched End-System Device That Handles Voice Applications As Well As Voice IERs

3.6.5 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 JCSS standard model, then the developer needs to adhere to the packet formats on the MAC of the JCSS standard model. Refer to “Appendix D: Packet Formats” for a list of the packet formats in the JCSS standard models. Interfaces can also support custom packet formats created by a model developer.

3.6.6 Interfacing with Other Classes

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

3.6.6.1 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

NwC_Ier_Key_Remote_Intrpt_Code_Fire_Ier_Instance and an ICI of type *ier_info* (see Appendix E: Standard OPNET Interfaces and Packet Formats), the SE will retrieve the IER information from the keys in 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 3-20 shows how the OE sends a remote interrupt to the SE.

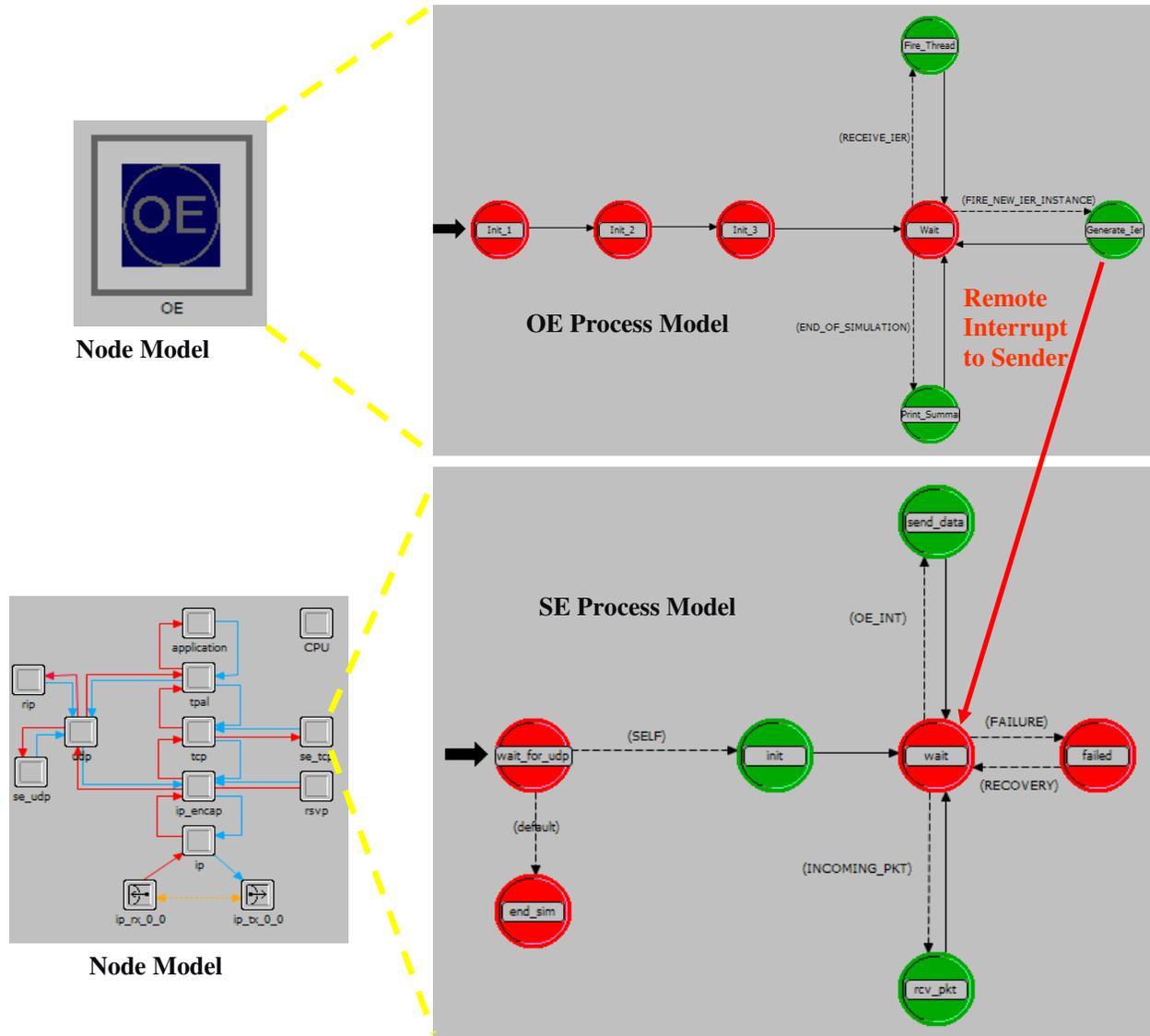


Figure 3-20: Remote Interrupt from the OE to the SE

3.6.6.2 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: Standard OPNET Interfaces and Packet Formats), the SE will generate a call for the duration specified in the ICI.

3.6.6.3 *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.

3.6.7 *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. 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 OPNET Modeler online documentation for more information.

3.6.7.1 *Creating Custom Transport Concerns*

In order to create a custom transport model that can be integrated into JCSS device models, there are several concerns that developers should be aware of. First, the developers must modify current SE models or develop a new SE model to interface with the new transport model. Currently, JCSS 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 are 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 pay attention to 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 JCSS, 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 JCSS PMO before modification! Also, it is a good practice to backup the current OE model before modification.

3.6.8 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 the Modeling Concepts manual, “Network Domain” chapter, and “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.

3.6.8.1 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 node about the failure of the IERs generated by itself

3.6.8.2 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.”

3.6.9 Collecting Statistics

IER statistics are written in the Output Vector (OV) format as well as the Output Table (OT) format. To enable the OE to do so, the IER Application Programming Interface (API) is used by the end devices for reporting all statistics. Therefore, this means that Ses should inform their own

OE (i.e., the OE that sent the fire IER interrupt to the end station) about the status of a particular IER so that statistics can be updated. For example, the “rcv_pkt” state of the *se_udp* process model uses the following API call with the IER keys and the destination node’s object id for the successful IER reception:

```
nw_ier_support_inform_ier_received ()
```

In the case of an IER failure, another example can be found in the *se_trafgen* process model, in the *process_message* state’s Enter Executives:

```
nw_ier_support_inform_ier_failure ()
```

The following are example cases when an end system should use the IER API:

- 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
- 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

3.6.10 JCSS Standard SE Models

The JCSS standard models include several SE models that can be used as a basis for most required device modeling. The most commonly used versions are shown in Table 3-9. They provide all of the required functionality for many of the JCSS devices and make use of the provided APIs. Therefore, development of a new SE process model may not be required.

Table 3-9: JCSS SE Process Models

Process Model	Description
<i>cs_voice_se</i>	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.”
<i>prc_se</i>	Generates <i>radio</i> packets in response to VOICE IERs. The parent module of this process should have the name “se.”
<i>se_link16_host</i>	Generates <i>radio</i> packets in response to VOICE IERs. IERs are translated into J-Series messages based on MIL-STD-6016C. The parent module of this process should have the name “se.”
<i>se_proc_mod</i>	Generates <i>data</i> packets in response to DATA IERs for the JTIDS radio. The parent module of this process should have the name “se.”
<i>se_trafgen</i>	Generates <i>data</i> 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.”

Process Model	Description
Se_udp	Generates <i>data</i> 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."
Voip_se	Generates the various Voice over IP signaling packets (H.323 and SIP) in response to VOICE IERs and voice standard applications. The parent module of this process should have the name "se."

If a new end-system model is expected to interface with existing JCSS 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.

3.6.11 Example: Constructing a Computer Model

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

3.7 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.

3.7.1 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	-- <i>Inherent</i> --	Specifies name of device
model	String	-- <i>Inherent</i> --	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; JCSS ships with a <i>classification.ad.m</i> file that developers can use for their models.
Equipment_type	Enumerated	Switch, router	Identifies the device type

3.7.2 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.

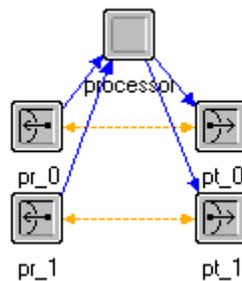


Figure 3-21: Example of Layer 1 Networking Equipment—Node Model

3.7.3 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.

When creating these devices, the user should use the same interface types and packet formats that are used by the devices it can directly connect to through the use of links. Refer to “Appendix D: Packet Formats” for a list of the packet formats in the JCSS standard models and refer to “Appendix E” for a list of standard OPNET interface types.

3.7.4 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 JCSS 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.

3.7.5 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.

3.7.5.1 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 itself can do the cleanup. In cases where logical links are viewed by the network (like in JCSS), the edge devices (devices at the ends of a logical link) can detect and perform the cleanup automatically for the networking device. For example, 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.

3.7.5.2 Handling Recovery of Self

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

3.7.6 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 the OPNET Modeler online documentation for detailed examples of how to accomplish this. 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 JCSS 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 JCSS 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 JCSS, 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.

3.7.7 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.

3.8 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.

3.8.1 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	-- <i>Inherent</i> --	Specifies name of device
model	String	-- <i>Inherent</i> --	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

3.8.2 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. Please note this table is not exhaustive and additional modules can be found in the OPNET Modeler online documentation.

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 (JCSS)	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

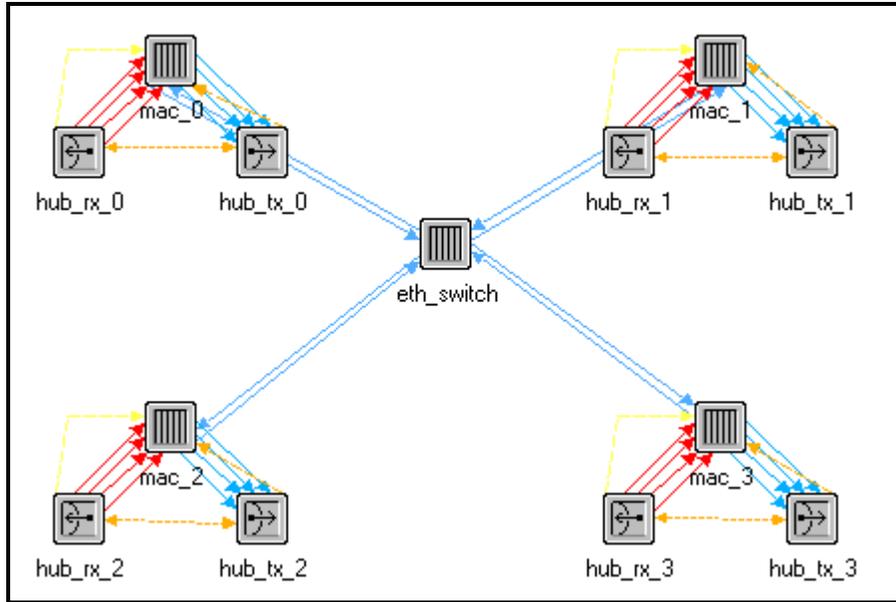


Figure 3-22: Example of Layer 2 Networking Equipment—Node Model

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 a 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

3.8.3 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.

3.8.4 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 the Generic Circuit API Section (3.14.1).

When creating these devices, the user should use the same interface types and packet formats that are used by the devices that it can connect to. Refer to “Appendix D: Packet Formats” for a list of the packet formats in the JCSS standard models and refer to “Appendix E” for a list of standard OPNET interface types.

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 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)
- Wireless (802.11 WLAN, EPLRS, SATCOM, MIL-STD-188-220D, UHF DAMA, TDMA, etc.)

3.8.6 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.

3.8.6.1 *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).

3.8.6.2 *Handling Recovery of Self*

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

3.8.7 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 the OPNET Modeler online documentation for detailed examples of how to accomplish this.

3.8.8 Example: Constructing a Multi-Service Switch

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

3.9 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 JCSS 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.

3.9.1 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	-- <i>Inherent</i> --	Specifies name of device
model	String	-- <i>Inherent</i> --	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.

3.9.2 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 3-23.

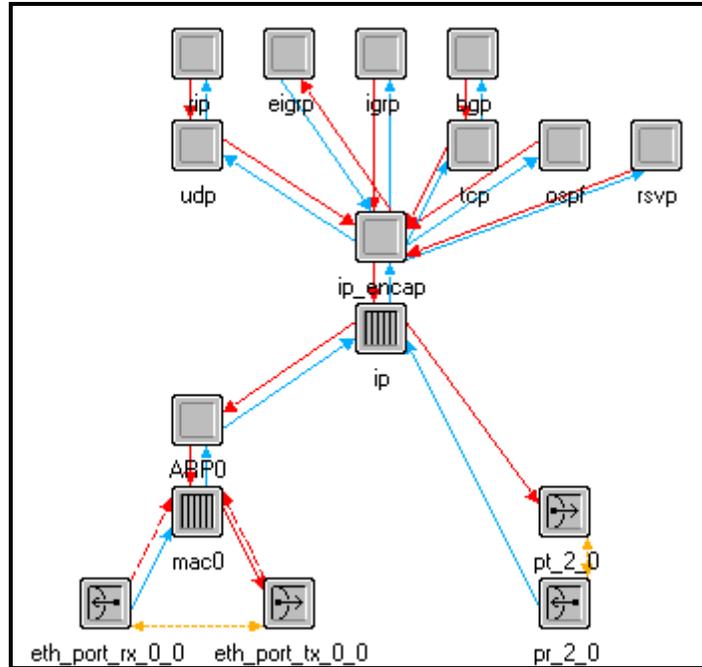


Figure 3-23: Example of Layer 3 Networking Equipment—Node Model

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 (fdi_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	FRIPF (frms_fr_ipif_v3)

3.9.3 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 3-24). 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.

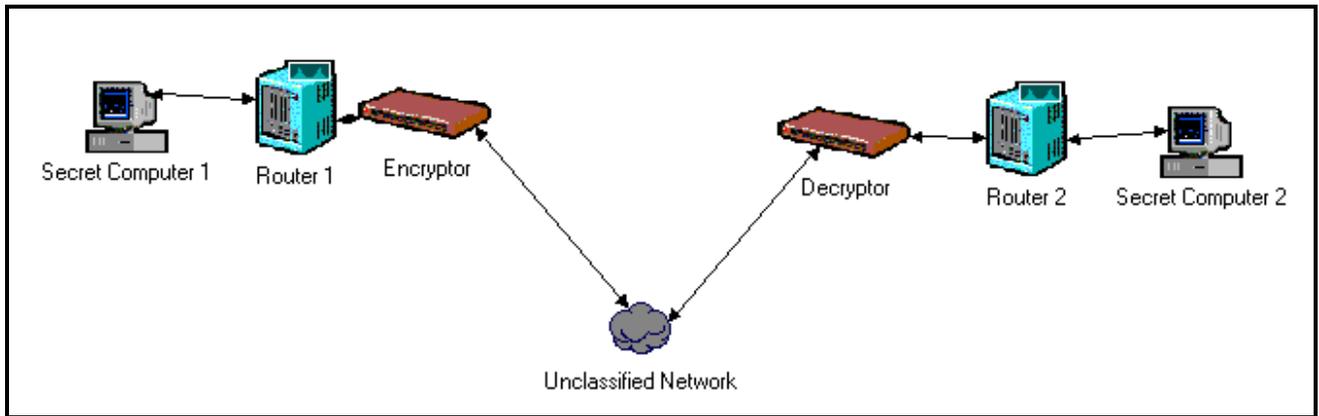


Figure 3-24: Networks with Different Security Classification Levels

3.9.4 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 ()
```

When creating these devices, the user should use the same interface types and packet formats that are used by the devices that it can connect to. Refer to “Appendix D: Packet Formats” for a list of the packet formats in the JCSS standard models and refer to “Appendix E” for a list of standard OPNET interface types.

3.9.5 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
- Wireless (802.11 WLAN, EPLRS, SATCOM, MIL-STD-188-220D, UHF DAMA, TDMA, etc.)

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.

3.9.6 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.

3.9.6.1 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., 15.0.A). For more information, refer to OPNETWORK Session 1510: Understanding IP Model Internals and Interfaces.

3.9.6.2 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.

3.9.6.3 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:

1. 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`

2. Running the custom routing protocol on all interfaces in the network.

3.9.6.4 *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.

3.9.7 *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.

3.9.7.1 *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.

3.9.7.2 *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.

3.9.7.3 *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.

3.9.7.4 *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 redibilit routes to all destinations through the recovered node and update the routing tables if the new route is better than the existing routes.

3.9.8 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 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 the OPNET Modeler online documentation for detailed examples of how to accomplish this task.

3.10 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 JCSS 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 JCSS.

3.10.1 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 for a 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 for 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

3.10.2 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 JCSS-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.

3.10.3 Routing in Circuit-Switched Devices

This subsection describes the JCSS 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 and sends out a CONNECT packet. If the CONNECT packet reserves bandwidth on all the links to the destination and potentially bumps calls based on priority. If bandwidth is not available on the link, then the switch writes an IER Failure statistic for the call trying to be set up. If the switch bumps a lower priority call, an IER Failure statistic is written for the bumped call. Once the call has been established, a global voice manager is used to synch all devices along the path of the call. Once the call has been completed or failed, the global voice manager will notify all devices that the call has ended. The global voice manager is implemented using the *global_voice_mgr.pr.m* and *nw_voice_mgr.ex.c*.

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*.

3.10.4 Circuit-Switched Links

JCSS 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.

3.10.5 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:

1. 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. Also the gateway registers with the global voice manager discussed in the previous section.
2. 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 3-25. 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.
3. 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.
4. 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.
5. The multi-service switch acts as the destination phone inside the circuit-switched network. The job of the multi-service switch is to translate the different signaling protocols during the course of the call.
6. 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.
7. Once the call is completed or failed/bumped, multi-service switches in the route are notified to stop generating voice packets through the use of the global voice manager.

For the actual models used with media gateways and Voice over IP, refer to the <JCSS Installation Directory>\Sim_Domain\op_models\netwars_std_models\voip directory.

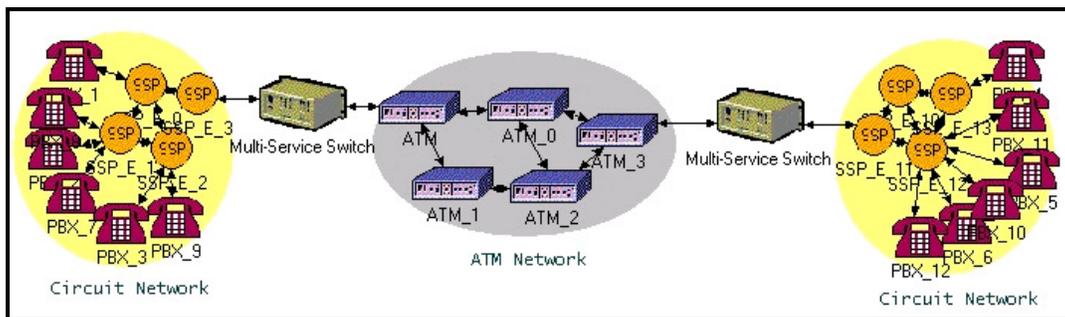


Figure 3-25: Circuit-Switched and Packet-Switched Network Intercommunication

3.10.6 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.”

3.10.6.1 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 JCSS Layer 2 networking device with circuit-switched capabilities handling voice calls fails, it informs the global voice manager and writes out the IER statistics. The global voice manager then interrupts all devices along the path of the call and informs those devices that the call has ended. Those devices then free their bandwidth for the call.

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

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

3.10.7 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

3.11 COMPLIANCE FOR WIRELESS INTERFACES

The end-system or network equipment devices in JCSS 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.

3.11.1 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 BER 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 <i>channel</i> 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 stage⁵ 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 required by OPNET’s radio pipeline stages. Most of the attributes defined in Table 3-20 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.

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

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.

3.11.1.1 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 a 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.

3.11.1.2 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 a 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 BER for the incoming transmission.
Error model	dra_error	Computes the number of bit errors in a segment of the incoming transmission.
Ecc model	dra_ecc	Determines the acceptability of an incoming transmission.

3.11.2 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.

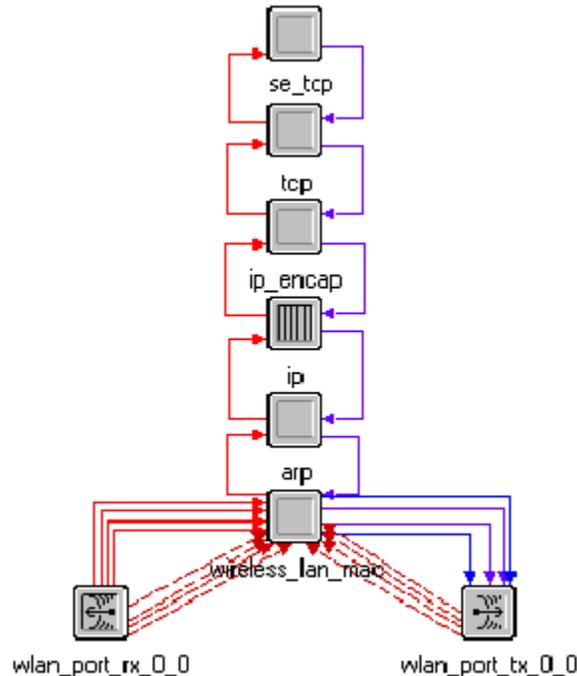


Figure 3-26: Example of a Radio End-System Device—Node Model

3.11.3 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 “3.6. Compliance for End-System Devices” as this section specifically deals with building end-system devices.

3.11.4 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.

3.11.5 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.)

3.11.6 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.

ATM device with radio interface

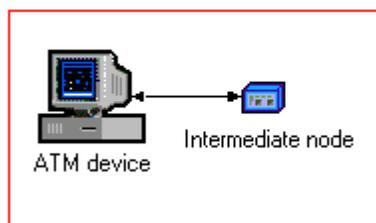


Figure 3-27: An ATM Device with a Radio Interface

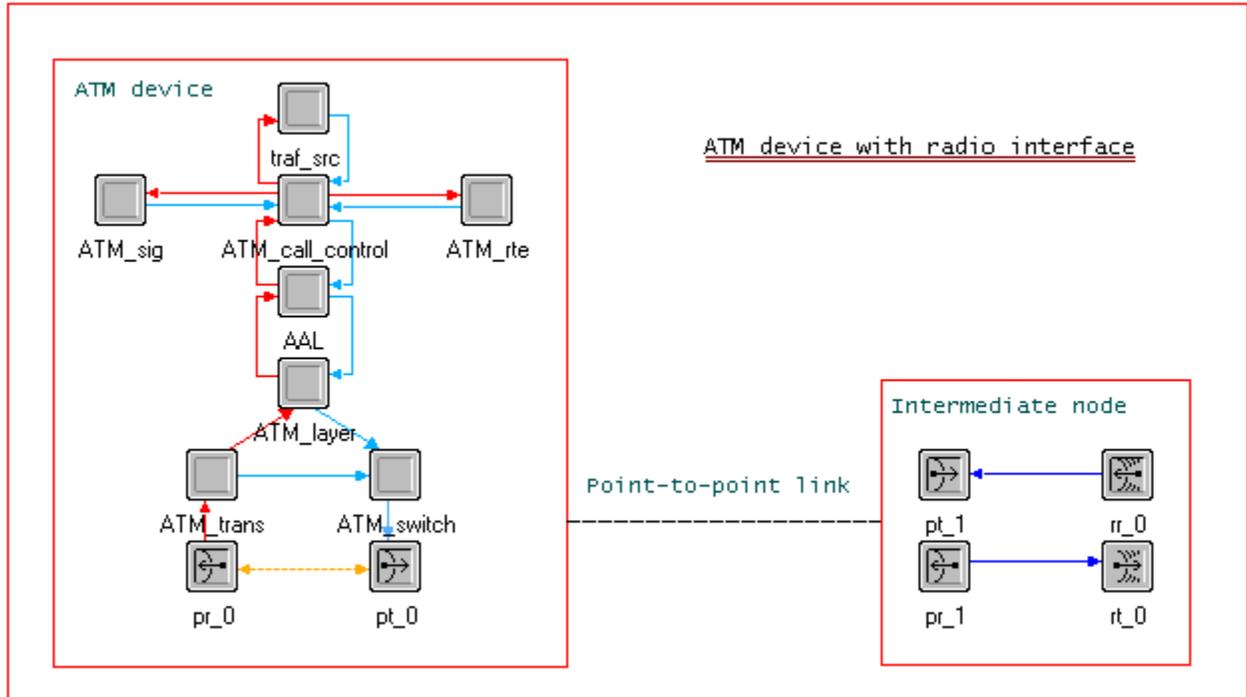


Figure 3-28: Internal Representation of an ATM Device and Intermediate Node

3.11.7 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 JCSS models, currently the devices connected to the radio devices perform the IER cleanup operations in case the radio device fails.

3.11.8 Collecting Statistics

Broadcast network utilization statistics are collected for broadcast radios.

3.11.9 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.

3.11.10 Satellite Considerations

A satellite device can be modeled as a networking device with radio interfaces, as documented above. The current JCSS 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.

3.11.11 JCSS Standard Geostationary Satellite Communications System Models

A satellite communications system can be modeled in two ways, either based on the JCSS 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 JCSS 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.

Note that the ground terminal device model that can communicate with the JCSS 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
7	Power Calculation	Rx	nwra_power_tirem.ps.c

Stage	Function	Module	File
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 4.12, Satellite Terminal Generic Example.

3.11.12 Generic Satellite Device Model (for Bent Pipe Links)

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

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

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

3.11.14 TSSP Satellite Terminal Device Model

To learn how to create a device of this type, refer to subsection 4.13, Satellite Terminal with TSSP Example.

3.11.15 Broadcast Radio Considerations

Integrating a custom radio with the broadcast network framework involves modifying some files that define this framework. JCSS 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>\JCSSrules\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*.

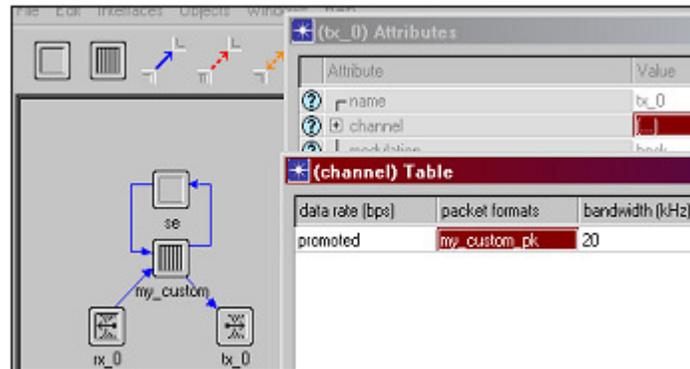


Figure 3-29: (Channel) Table

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).

3.11.16 EPLRS Radio Considerations

Integration of custom radio models with EPLRS radios in the first place requires using the customized pipeline stages used by EPLRS radios. EPLRS radios use a Packet Error Rate (PER) model as opposed to the standard Bit Error Rate (BER) model used in other radios. The following is the list of pipeline radios used in EPLRS:

- rxgroup model: *eplrs_rxgroup*
- txdel model: *eplrs_txdel*
- closure model: *dra_closure*
- chanmatch model: *eplrs_chanmatch*
- tagain_model: *NONE*
- propdel model: *dra_propdel*
- ragain model: *NONE*
- power model: *eplrs_power_no_rxstate*

- bkgnoise model: *dra_bkgnoise*
- inoise model: *dra_inoise*
- snr model: *dra_snr*
- ber model: *NONE*
- error model: *NONE*
- ecc model: *eplrs_per*

EPLRS radios detect each other through exchange of Hello messages which are sent using *eplrs_hello* packet format. Actual data is transmitted using *eplrs_pdu* packet format. An *eplrs_pdu* packet can carry different number of payload bits depending on which waveform from the set of 18 available waveforms is used by each needline. A custom radio model should use the same waveform format as the one used by the waveform of the needline it is communicating with.

EPLRS Radios use the 420-450MHz frequency range which can be divided in 5, 6, or 8 channels based on the *Channel Set* attribute of the EPLRS_ENM node (Figure x). Therefore, if a custom radio model is required to operate in all of the above modes, it should set the channels of its transmitter module in software to accommodate different frequencies based on different channel set options. Power level for all radios in a division is set in the *Network Power Level* attribute of the EPLRS_ENM. In general, any custom radio node that needs to communicate with the EPLRS radios in a division needs to read most of the information about the division from the attributes of the EPLRS_ENM node corresponding to that division.

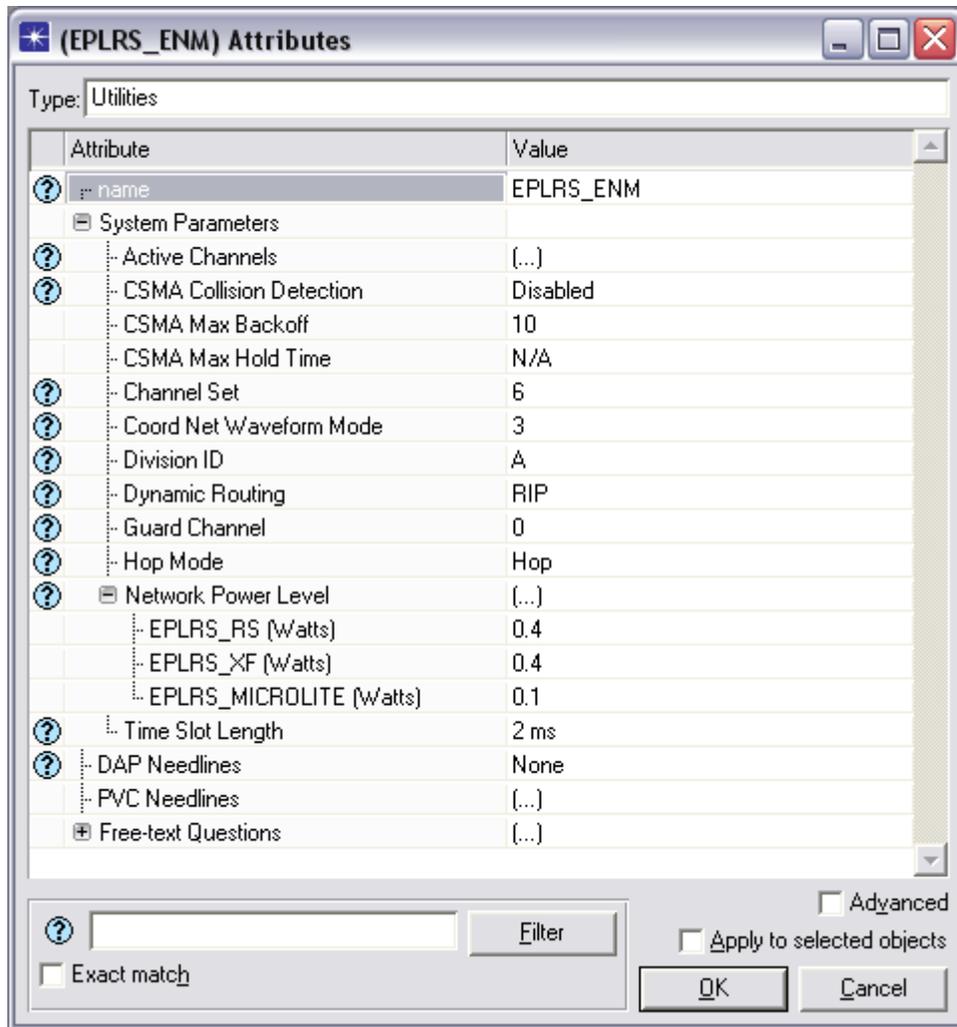


Figure 3-30: EPLRS ENM System Parameters Attribute

3.12 COMPLIANCE FOR LINK MODELS

Links connect devices in a network. JCSS 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 JCSS 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 JCSS 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.

3.12.1 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 a Link Model

Attribute Name	Attribute Type	Default Value	Description
name	String	-- <i>Inherent</i> --	Specifies name of link
model	String	-- <i>Inherent</i> --	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.

3.12.1.1 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 JCSS' Link Deployment Wizard (LDW). The LDW uses this file to match links to appropriate ports. JCSS maintains this file under <JCSS 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.

3.12.2 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

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

3.12.3 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.

3.12.4 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.

3.12.5 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.

3.12.6 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.

3.12.7 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: Provide 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).

3.13 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.

3.13.1 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.

3.13.2 Self Description

Self Description must be added to any Utility Node used in the network. Many of the Scenario Builder features will read the Self Description and ignore the Utility Node when performing operations. This is ideal because the Utility Node is only used for configuration and is not an actual device. To signify that a node is a Utility Node, the user should put the value *Utilities* inside the *machine type* attribute in the core Self Description of the Utility node model.

3.13.3 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.

3.13.4 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

⁶ For example, if a utility node should act only on particular end nodes, then it may have the attributes *end node (1)* and *end node (2)*. The values of these attributes will be the full hierarchical names of *ed1* and *ed2*.

models using the utility nodes should be designed to work with multiple instances of the utility node.

3.13.5 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-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 JCSS 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 JCSS Scenario that supports packet format “cs_special.” In this way, the user would have an easy way to set up “cs_special” circuits.

3.14 API AND FRAMEWORK

3.14.1 Generic Circuit GUI API

3.14.1.1 Purpose

JCSS features a wide array of circuit models that require circuit configuration. These include Prominas, FCC-100, TSSP, and N.E.T. SCREAM and SHOUTip devices, among others. In previous versions of JCSS, users were required to use several different circuit configuration wizards for different types of models (and therefore circuits). This process was not very extendable and users had a difficult time using circuit devices in JCSS. Because of this, JCSS 7.0 introduced the Generic Circuit Wizard which also contains a generic API and workflow. This feature allows any node model with circuit requirements to easily integrate with the Scenario Builder, Capacity Planner and DES features. It also provides a common workflow between all circuit devices no matter the type of circuit. The section below provides specific information on this API and how users can intergrate with it. It should be noted to work with the API and functionality, certain attributes and settings will be required in the circuit based node models. Also, an external data file describing the circuits used by the node models must be created so the device can work with the Generic Circuit Wizard.

3.14.1.2 Node Model Requirements

In order for the circuit based node model to function, a value for the “machine type” must be specified in the self description. This attribute will associate the device with the external data description files. For each node model, the user will also need to define a node-level compound attribute called “Circuit Configuration”. The attribute must define two sub-attributes which include “Local Port” and “Circuit Speed”. Also, it is recommended that the user should implant active attribute handlers for the “Local Ports” attribute to prevent users from inadvertently changing the circuit configurations. All circuit configurations should be performed through the Generic Circuit Wizard.

3.14.1.3 Path Model

A new path model “nw_circuit” is used to represents circuits in JCSS. Inside the “nw_circuit” path model, there are the following attributes:

- Circuit Type: Describes the circuit type and is critical since certain devices may support more than one type of circuits.
- Port A & B: Source device port and Destination device port name.

The *Port A/B* attributes will contain values mapping to the *Circuit Configuration > Local Port* attributes for each of the end devices that are connected to the path. Unlike OPNET link objects, OPNET path objects attach only to nodes and not to their ports. This means that the specified attributes must be correctly populated for the proper operation of the circuits. Also, as each end device does not contain attributes that specify the other remote port on the remote device, the *Port A/B* attributes alone must record the port assignment for the circuit.

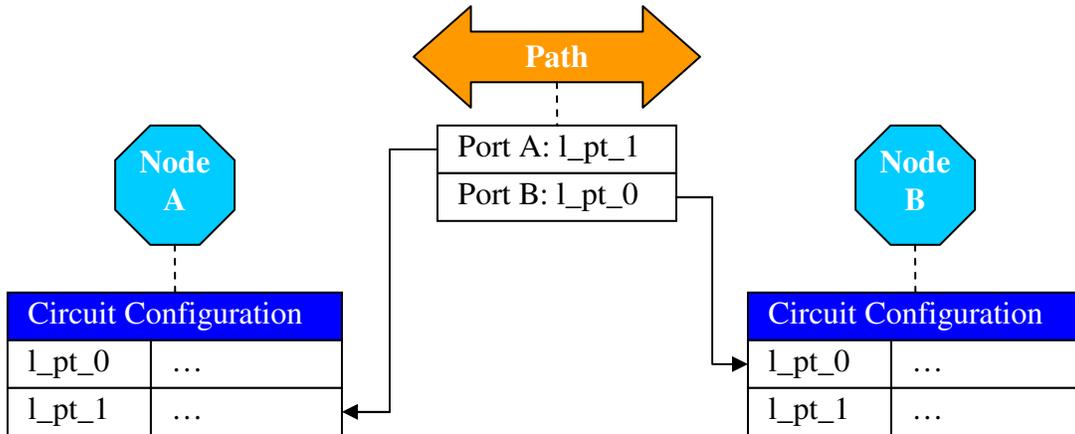


Figure 3-31: Path Port Associations Diagram

The ‘A’ and ‘B’ designations refer to the first and last nodes listed for the circuit by the OPNET topology functions. These functions list the nodes of a circuit in a consistent order throughout the life of the object, so *Port A* and *Port B* can be consistently associated to the correct end devices without using the names of the devices.

3.14.1.4 Data Description Files

The user also needs to create a data description file for each “machine type” value specified by the circuit based devices. These files will allow the Scenario Builder to support new devices by listing the circuit types supported by the devices and describing the node attributes that are important to circuit configuration. Some of the node attributes will be associated with particular circuit types. The Scenario Builder will allow circuits between two devices only when the devices define at least one common circuit type in their data description files.

Each file will have an XML format similar to the following:

```
<MachineType>
  <CircuitTypes>
    <CircuitType>
      <Name>Promina</Name>
      <Color>blue</Color>
      <Attributes>...</Attributes>
    </CircuitType>
  </CircuitTypes>
  <Attributes>
    <Integer>
      <Name>Call priority</Name>
      <DefaultValue>7</DefaultValue>
      <LowerBound>0</LowerBound>
      <UpperBound>15</UpperBound>
    </Integer>
    <String>
      <Name>Call type</Name>
      <DefaultValue>Permanent</DefaultValue>
    </String>
  </Attributes>
</MachineType>
```

```

    <AcceptedValues>
      <Value>Permanent</Value>
      <Value>Demand</Value>
    </AcceptedValues>
  </String>
</Attributes>
</MachineType>

```

The file must declare at least one circuit type, but attributes are optional. The circuit type values will be stored in the *Circuit Type* attribute of the circuit's path object. The optional *Color* element is a special attribute that allows circuit types to have predefined colors.

All attributes specified in the data description file must be sub-attributes of the *Circuit Configuration* attribute, so the attribute names should be relative to that attribute. For example, *Call priority* above is actually the *Circuit Configuration > Call priority* attribute on a Promina node. When a circuit in the Scenario Builder accesses these node attributes, it will access the row of the "Circuit Configuration" attribute that corresponds to itself by matching its *Port A/B* attribute to the node's *Local Port* attribute.

The files are organized around machine types rather than circuit types primarily because the attributes in the description are more likely to be specific to the node model than to the circuit. However, this approach will require all devices with the same machine type to support the same set of circuit types and the same set of attributes.

A data description file may be located in any directory specified in the "mod_dirs" preference and must have a name of the form "<machine type>.circuit.xml". This requirement restricts machine type values to be valid in filenames.

3.14.1.5 *Circuit API*

The Circuit API is a set of utility functions designed to get miscellaneous information in DES. This API can be used by any circuit based device which uses the `nw_circuit` path objects to define circuits.

Circuit API functionality includes:

- Getting remote device
- Getting remote port
- Remote IP or ATM address
- Getting circuit speed
- Getting selected circuit path (if applicable)
- Getting current voice load

Circuit API functions are declared in `nw_circuit_api.h` and defined in `nw_circuit_api.ex.c`. Use case examples include: circuit interface module (`cir_intf.pr.m`), CTP (`ctp_dispatch.pr.m`), SCREAM (`bbs_dispatch.pr.m`), Promina (`pro_node_manager.pr.m`), and others.

3.14.1.6 Circuit Interface Module

The Circuit Interface Module is an optional plug and play module for circuit based devices that are used in DES. If used, one module is required for each circuit port.

This module provides support for:

- Interface with JCSS circuit switched voice calls
- Explicit/hybrid packet generation to represent voice call load on the circuit
- Collection of circuit level statistics (circuit level utilization, throughput)
- Generation of hybrid packet

The Circuit Interface Module is currently used by CTP and SCREAM device to load the network for voice calls. The following is an example of the *Port Configuration Table* attribute on the Circuit Interface Module. This attribute is used to configure the module for simulation.

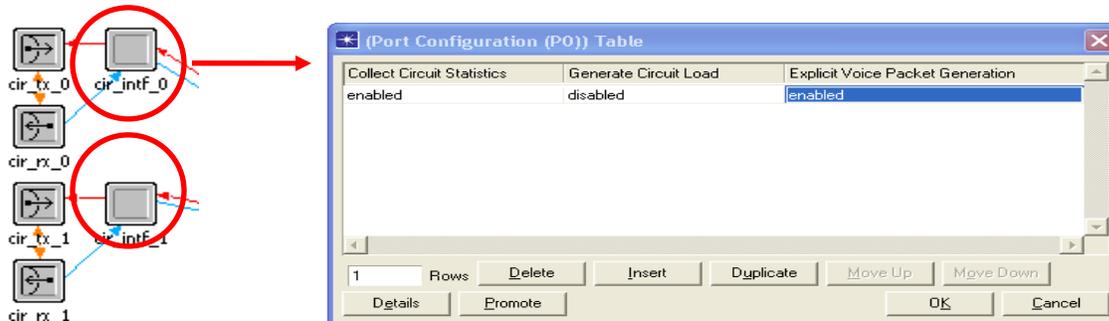


Figure 3-32: Circuit Interface Module Port Configuration Table

3.14.2 IP Auto Addressing

IP Auto Addressing is a run-time IP subnet establishment that assigns an IP address to each IP interface during simulation startup. Each IP device will graph-walk over each active IP interface to discover all of its neighboring IP interfaces to be placed inside the same IP subnet. The majority of this implementation comes from standard OPNET library and are declared in *ip_auto_addr_sup_v4.h* and defined in *ip_auto_addr_sup_v4.ex.c*. There is additional support for JCSS devices in *nw_custom_ip_auto_addr.ex.c*, and *nw_ip_modification_support.ex.c*.

3.14.3 Hybrid API

Hybrid packets are used to represent a large number of explicit packets during simulation. Instead of sending thousands of packets to model a network load, a single tracer packet can be sent on a network to represent a large amount of packets. The advantage is that the simulation runs faster as there are fewer events and packets even though the network is still loaded properly (such as the device queues, links, etc.). Also, many statistics (such as queuing delays, end-to-end delay, utilization, etc.) will be updated accurately in comparison with sending explicit packets as the tracer packet will continue to use the actual protocols and processes defined on the devices in the network. Some of the disadvantages are that in high loss environments or failure studies, the tracer packet could be lost and the network may not take into account the load the tracer packet

represented. Also, as each packet is not sent, certain sensitive statistics such as jitter cannot be calculated accurately.

Regardless, the Hybrid API can be used by any model in DES which has an IP/ATM stack to generate tracer packets dynamically. The API has functions which are similar in nature to the standard Packet Creation APIs provided by OPNET. This module has functions to:

- Generate a single tracer packet with given bits/sec (bps) and packets/sec (pps)
- Generate a train of tracer packets with given bps, pps for given period of time with given tracer generation interval
- Change the bps/pps of a defined train of tracer packets
- Change the bps/pps of a defined train of tracer packets to account for explicit packets

The API functions are declared in *nw_oms_basetraf_src.h* and defined in *nw_oms_basetraf_src.ex.c*. The following screen shots display how SHOUTip (using the *fsr_to_voip* and *voip_to_voip* process models) uses explicit and hybrid packets to model VoIP traffic in a network:

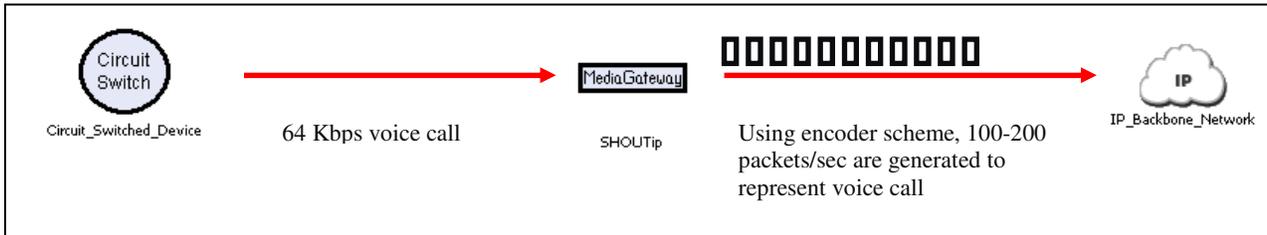


Figure 3-33: Explicit Traffic

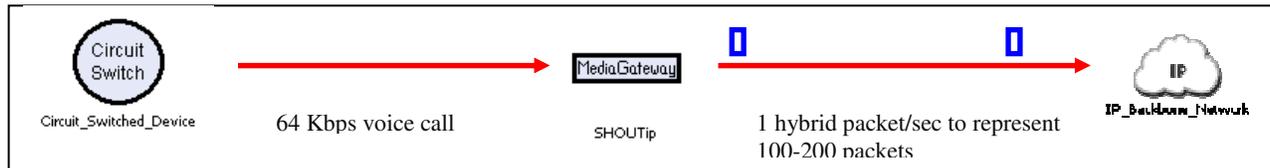


Figure 3-34: Hybrid Traffic

3.14.4 Link Deployment Wizard

The Link Deployment Wizard (LDW) provides a quick and easy way to deploy links such as wired, radio, SHF satellite links, and GBS satellite links. The objective is to have the user utilize a consistent workflow for all device types where the user is given correct link defaults to avoid mis-configuration. This includes selecting two devices and initiating the wizard through the menu options or shortcut. Depending on the devices selected, different information is used to display potential link configurations to the user:

- For both wired and radio links, the wizard uses the device self-description (interface type and machine type), as well as, the LinkTypeMap.gdf file to give possible link selections. For wired and radio based custom models, the user can easily integrate the

model with this functionality. Refer to Appendix T for more details on how to use self-description properly.

- The SHF/GBS satellite links use a combination of device attributes (such as Equipment Type, Home Satellite, Channel Config, and Nodal Mode) to populate the wizard. These wizards are specialized to cater to the JCSS satellite model suite. Users will have a more difficult time integrating with these wizards unless they use identical attribute names and values.

To integrate wired and radio devices with LDW, the user will need to configure their custom devices to use the proper self-description attributes and values. Secondly, the user must make sure that the LinkTypeMap.gdf file has at least one compatible link defined. The file contains a list of relevant links, their appropriate data rates, and other additional information which is used by the wizard primarily to filter the links and populate the default data rates for the device pair.

During LDW operation, the wizard uses the following workflow:

- After the user has selected two OPFACs or two devices, the wizard will match the interface type attribute of the port group self-description to find common technologies.
- Based on the common technologies (which are displayed in the *Select Port Group* drop down menu), the wizard will find the corresponding matching links. Depending upon the selected technology, the wizard will show valid links, source and destination ports, and the default data rates. For each technology, the wizard:
 - Uses the LinktypeMap.gdf file to get a list of wired or radio links
 - Checks each link using the interface type attribute of self description of the link model to see if it can support this technology
 - Finds matching/free ports (on each device) for each common technology

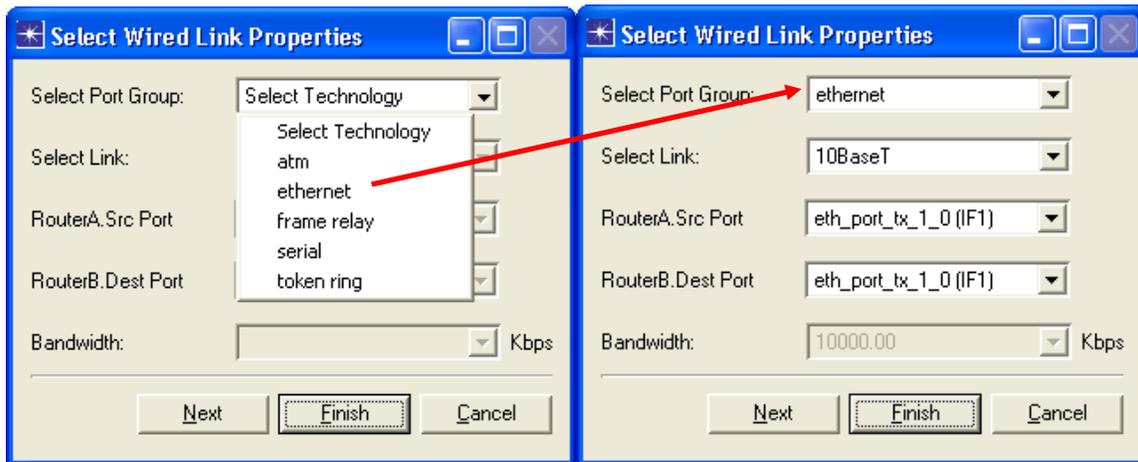


Figure 3-35: Link Deployment Wizard Dialog

3.14.5 Broadcast Network Framework

The Broadcast Network Wizard allows the user to correctly configure radio models that have broadcast network functionality. As part of this feature, the wizard performs the following operations:

- By specifying an XML file for each type of radio, the user can set the correct defaults, configurations, etc. for a given radio.
- Once the user deploys a group of radios into a scenario and uses the Broadcast Network Wizard, the wizard will check for common models of operation and look at the appropriate XML file for information.
- This wizard will then display a table of attributes relevant to each common mode (with default values and ranges for acceptable values). As these are specified in the XML file, the values are easily changeable and will only show the correct settings to the user to avoid mis-configuration.
- The Broadcast Network Wizard will then use the values set by the user inside the wizard to correctly configure each device in the scenario and deploy a broadcast network.

Currently, all JCSS radios use this functionality to deploy Broadcast Networks.

In the Broadcast Network Framework, the XML file contains the following information:

- Modes of operation (waveform used by radio e.g. HAVEQUICK I/II, etc)
- Attributes which needs to be configured
- Default value for each attribute
- Range of acceptable values

Each file will have an XML format similar to the following example shown below. In this case, the example below is for the SINCGARS radio model and the XML file is called SINCGARS.xml. The naming of the XML file should use the following schema <Radio Model Name>.xml.

```
<ModesofOperation>
  <Mode>
    <Name>sinCGARS</Name>
    <Attributes>
      <Double>
        <DisplayName>Default Frequency</DisplayName>
        <InternalName>min frequency</InternalName>
        <DefaultValue>30</DefaultValue>
        <DisableValue>promoted</DisableValue>
        <LowerBound>30</LowerBound>
        <UpperBound>88</UpperBound>
        <UnitTag>MHz</UnitTag>
      </Double>
      <Double>
        <DisplayName>Channel Bandwidth</DisplayName>
        <InternalName>bandwidth</InternalName>
        <DefaultValue>5</DefaultValue>
        <DisableValue>promoted</DisableValue>
        <AcceptedValues Open="true">
          <Value>5</Value>
          <Value>6.25</Value>
        </AcceptedValues>
        <UnitTag>KHz</UnitTag>
      </Double>
      <Double>
        <DisplayName>Data Rate</DisplayName>
```

```

    <InternalName>data rate</InternalName>
    <DefaultValue>16000</DefaultValue>
    <DisableValue>promoted</DisableValue>
    <AcceptedValues Open="true">
      <Value>75</Value>
      <Value>150</Value>
      <Value>300</Value>
      <Value>600</Value>
      <Value>1200</Value>
      <Value>2400</Value>
      <Value>4800</Value>
      <Value>16000</Value>
    </AcceptedValues>
    <UnitTag>bps</UnitTag>
  </Double>
</Attributes>
</Mode>
</ModesofOperation>

```

3.14.6 Wireless Configuration Node

The Wireless Configuration Node inside the Configuration OPFAC can modify the BER, PER, ECC, and Antenna Pattern behavior using the Node Groups attribute. This attribute allows the user to select profiles to represent the given Wireless pipeline stage (i.e., BER, PER, etc.). The functionality of the Wireless Configuration Node includes:

- User can import values from text file
- Allows for “playback” of live exercise data
- Values can change dynamically during the simulation
- Changes only effect the Member radios for this attribute

All JCSS radios work with this new functionality. For custom radios, the user can utilize the APIs in the *jcss_wireless_config_support.ex.c* external file to modify the custom pipeline stages. For examples, see the *dra_ber*, *dra_ecc*, and *dra_error* pipeline stages.

The example below shows how to modify BER:

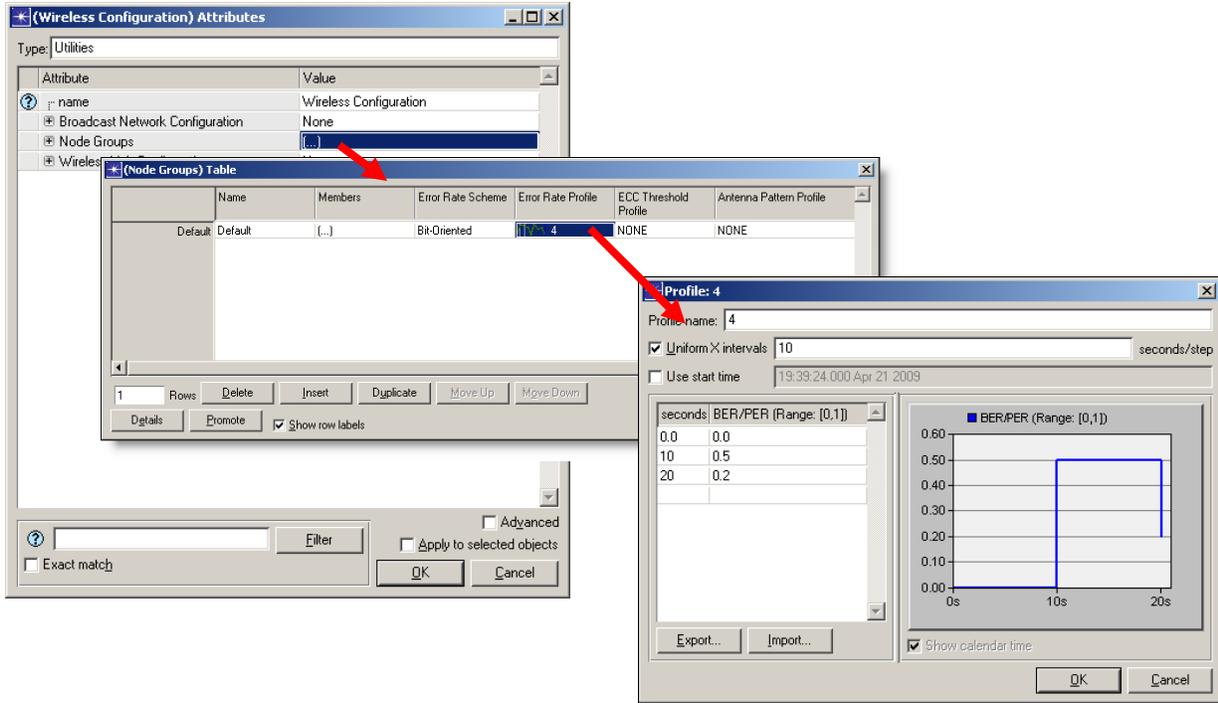


Figure 3-36: Wireless Configuration Node Modification of BER

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 editors⁷ 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.

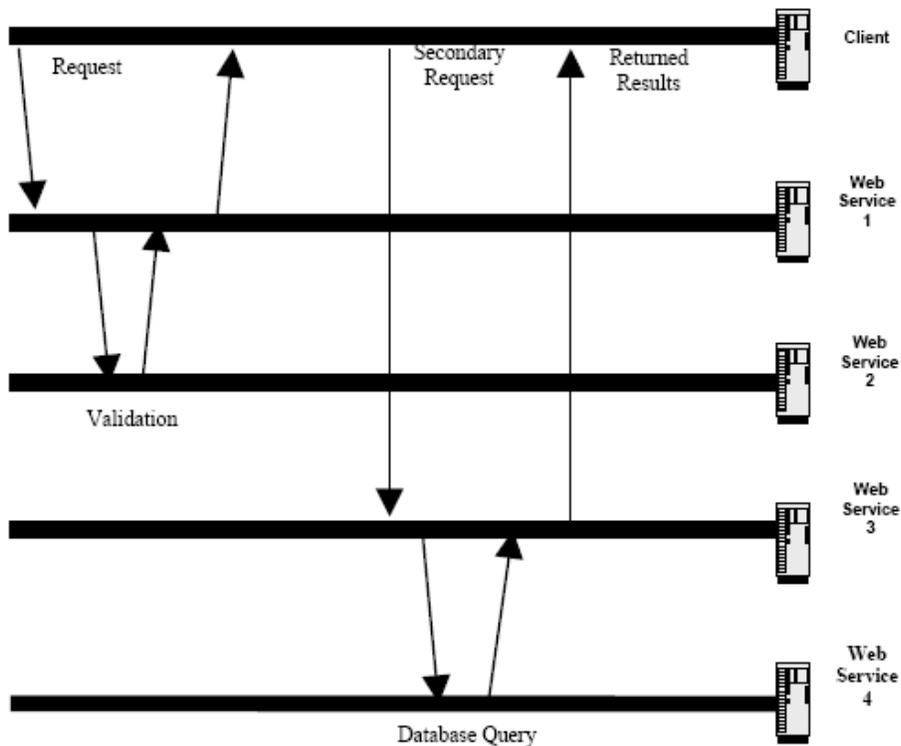


Figure 4-1: Time Sequence Diagram Example

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-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 decisions⁹ 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 Element¹⁰ “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.

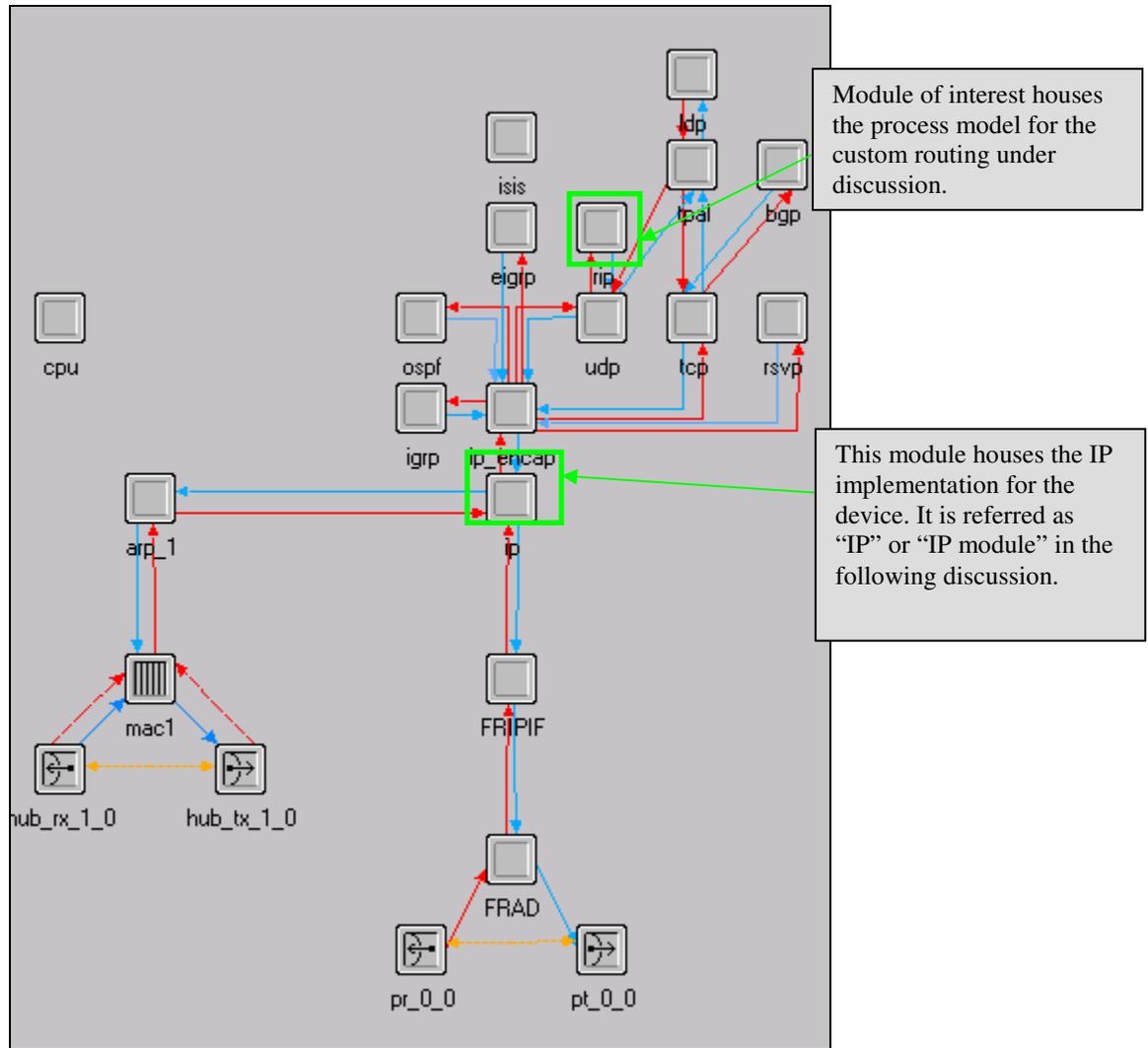


Figure 4-2: Sample Layer 3 Networking Equipment

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 API¹⁴ 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 process in 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 protocol 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:

1. 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.

2. 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 route¹⁷ (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_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/JCSS 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.

Attribute	Value
Router ID	Auto Assigned
Autonomous System Number	100
Interface Information (12 Rows)	(...)
Aggregate Interfaces	None
Loopback Interfaces	(...)
Tunnel Interfaces	None
VLAN Interfaces	None
BVI Interfaces	None
Default Gateway	Unassigned
Default Network(s)	None
Static Routing Table	(...)
Static Routes Across VRFs	Enabled
Load Balancing Options	Destination-Based
Multipath Routes Threshold	Unlimited
Administrative Weights	(...)
OS Version	Not Set
Standard ACL Configuration	None
Extended ACL Configuration	None
Damping Configuration	None
AS Path Lists	None
Community Lists	None
Extended Community Lists	None
Prefix Filter Configuration	None
Route Map Configuration	None
Local Policy	None
Forwarding Table Policies	Not Configured
Fate Sharing	Not Configured
IPv4 Configurations	Default

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).

Name	Status	Operational Status	Address	Subnet Mask	Secondary Address Information	Subinterface Information	Routing Protocol(s)	MTU (bytes)	Protocol MTUs	Metric Information	Layer 2 Mappings	Packet Filter	Policy Routing	Routing Instance	Compression Information	Interface Speed (Kbps)	Aggregation Parameters	Encapsulation	VRF Sitemap	Interface Configurations	Description
IF0	Active	Infer	204.0.10.1	Class C (natural)	Not Used	None	RIP	Ethernet (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF1	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	Ethernet (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF2	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	Ethernet (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF3	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	Ethernet (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF4	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF5	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF6	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF7	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF8	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF9	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF10	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A
IF11	Active	Infer	Auto Assigned	Auto Assigned	Not Used	None	RIP	IP (...)	Default	None	None	None	None	None	None	10 Mbps	None	Not Configured	None	Default	N/A

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 edited¹⁸ to include the custom routing protocol (IP Routing Parameters | Interface Information | Routing Protocol(s)). A new symbol map must be added for this attribute¹⁹ (as shown in Figure 4-5).

Attribute: Routing Protocol(s)

Data type: string

Attribute properties: Private Public

Symbol map:

Symbol	Value
RIP	RIP
IGRP	IGRP
OSPF	OSPF
BGP	BGP
EIGRP	EIGRP
IS-IS	IS-IS
TORA	TORA
ADDV	ADDV
OLSR	OLSR
CRP	Custom Routing Protocol

Default value: RIP Auto. assign value

Units:

Comments: Specifies the dynamic routing protocol(s) running on this interface.

Buttons: Move Up, Move Down, Delete, Add, ETS Handlers...

Figure 4-5: Routing Protocol Attribute Properties

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.

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, int proto, int admin_distance)	Adds a route entry to the common route table. This function checks for an already existing entry.
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,	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

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
IpT_Address next_hop, int proto, int new_metric)	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 hop²¹

proto: The unique protocol that entered this route²²; 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 Attributes²³

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

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

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.

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 build²⁶ 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 JCSS 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 v3.1, Subsection 3, Compliance for End System Devices, on the approach and methodology for creating an end-device model.

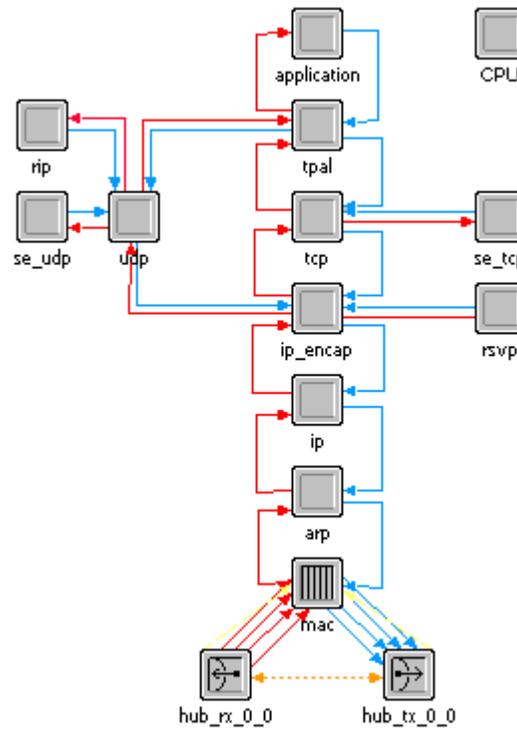


Figure 4-6: End-Device Node Model

The node model in Figure 4-6 is the actual node model of the JCSS standard node models; “NW_ethernet_wkstn_adv”²⁷

For details on how to design the node model for an end-device model, please refer to the “3.6. Compliance for End-System Devices” subsection. The following discussion about the process model development assumes that the minimum required attributes²⁸ 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 v3.1.

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).

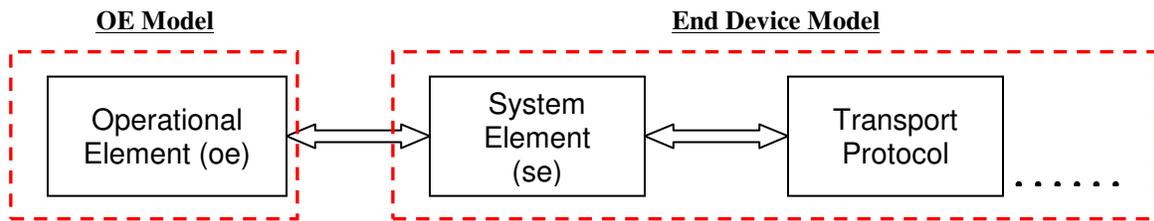


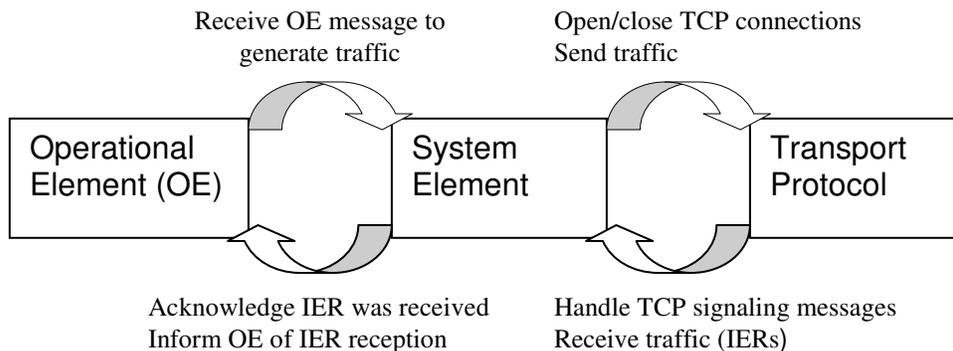
Figure 4-7: Interfacing Modules of “se”

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



³⁰ Please refer to Figure 4-17: Sample Workflow Diagram for SE Process Model of the Model Development Guide v3.1.

Figure 4-8: High-Level Functions of the “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.

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 <i>Oes</i> informing the <i>se_tcp</i> 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 <i>tcp</i> handshake messages that are sent from the <i>tcp</i> module.
Device failure	FAILURE	This is the failure information sent to the <i>se_tcp</i> module from the simulation kernel.
Device recovery	RECOVERY	This is the recovery information sent to the <i>se_tcp</i> 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 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 “Unforced” 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.

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).

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

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:

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

Current State	Logical Event	Condition	Action	Interim State (Forced State)	Next State
wait	Generate traffic	None	Open TCP connection	open_conn	wait
	Receive traffic	None	Put the packet in the reassembly buffer and close the TCP connection	rcv_pkt	wait

Current State	Logical Event	Condition		Action	Interim State (Forced State)	Next State
	Receive TCP signaling	To open a new connection		Open a new server connection	process_message	wait
		Informing the status of an existing connection	established	Send the packet and then close the connection	process_message	wait
			close	Inform the OE that the IER is received successfully	process_message	wait
			aborted	Inform OE of the IER failure that the connection aborted	process_message	wait
	Device Failure	None		Free up the IER and TCP connection related memory. Set the availability of the device as <i>non-available</i> .	None	failed
failed	Device Recovery	None		Set the availability status of the device <i>available</i> .	None	wait

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

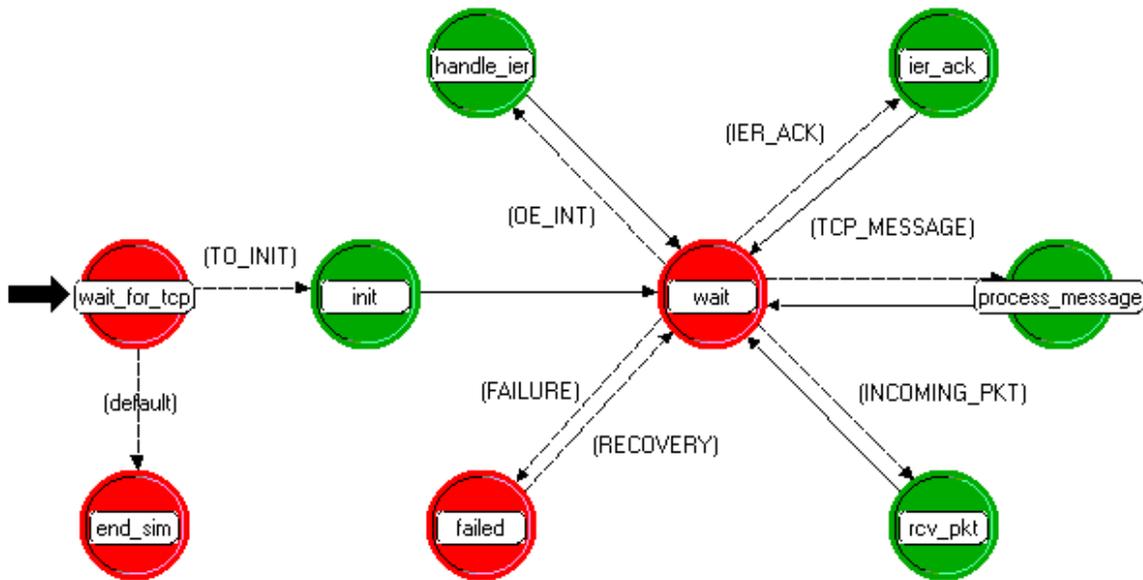


Figure 4-9: se_trafgen Process Model

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

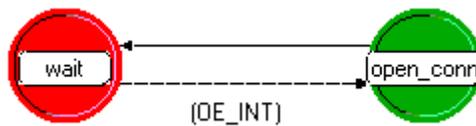


Figure 4-10: Open Connection State

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 (!strcmp(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 stores 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 ());
```

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 active³⁴ 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

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.

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

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

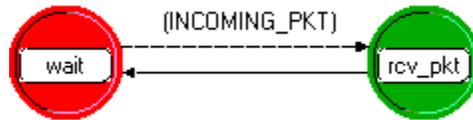


Figure 4-11: Receive Traffic State

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.

```

/* 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

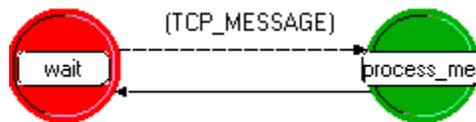


Figure 4-12: Process Message State

The execution should come to the *process_message* state from the *wait* state, on reception of a stream interrupt from the *tcp* module. The transition³⁵ 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.

```

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 further 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 */
}

```

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 icitype is "ier_ack").

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 IER³⁶ 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;
}

```

³⁶ Please refer to “Appendix G: Modeling ” of the Model Development Guide v3.1 on details on the codes used by the OE to communicate with SE and vice-versa.

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

```

case TPC_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 IER Handling State Implementation

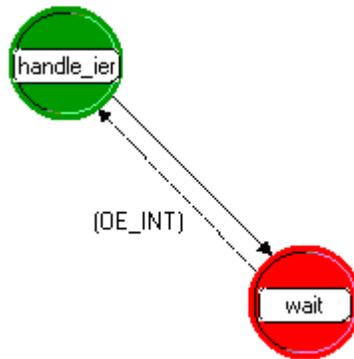


Figure 4-13: IER Handling State

The execution should come to the *handle_ier* state from the *wait* state upon reception of a remote interrupt from the OE. The transition for this state is *OE_INT*, defined in the header block as:

```

#define OE_INT ((intrpt == OPC_INTRPT_REMOTE) && !strcmp (icitype, NWC_Ier_Info_Ici_Name))

```

The following functions are performed in this state:

1. IER framework API is used to obtain the traffic type:

```

/* Get IER keys and consumer information from ICI sent by IER manager */
op_ici_attr_get (iciptr, NWC_Ier_Key_Field_Name, &ier_key);
op_ici_attr_get (iciptr, NWC_Ier_Instance_Key_Field_Name, &ier_instance_key);
op_ici_attr_get (iciptr, NWC_Consumer_Objid_Field_Name, &dest_node);

/* Use IER framework API to get traffic type of this IER */
traffic_type = nw_ier_support_get_traffic_type (ier_key);

```

2. IER framework API is used to track this application layer message:

```

/* Use IER framework API to track this application level packet */
nw_ier_support_tag_apprtracking_info (ier_key, ier_instance_key, pkptr);

```

Additionally, in the Enter Execs of the *rcv_pkt* state, the IER framework API is used to obtain the IER and IER instance keys from the application level packet and informing IER manager and COTS ADT support about reception of this IER. This is done by adding the following code:

```

/* Get the IER keys from the packet */
op_pk_nfd_get (pkptr, NwC_Ier_Key_Field_Name, &ier_key);
op_pk_nfd_get (pkptr, NwC_Ier_Instance_Key_Field_Name, &ier_instance_key);

/* USE IER Manager API to inform about successful reception of the IER */
nw_ier_support_inform_ier_received (ier_key, ier_instance_key, my_node_id);

/* Use COTS Aptracking API to record reception of the application packet */
aptrack_pk_reception_record (pkptr);

```

4.3.4.5 Failure State Implementation

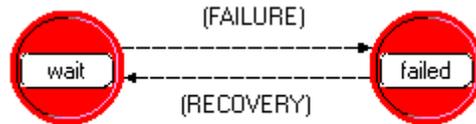


Figure 4-14: Failure State

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)
```

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 = (Trafigent_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 transmit 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:

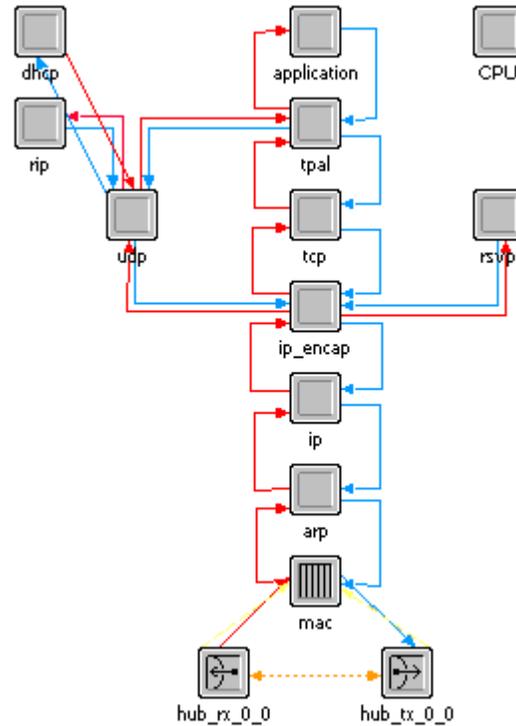


Figure 4-15: Ethernet_wkstn_adv—Node Model

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 4-16.

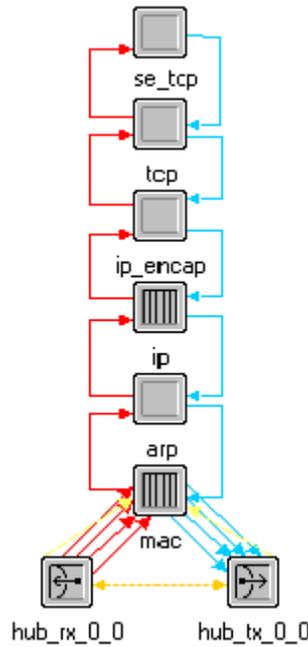


Figure 4-16: Computer—Node Model

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 JCSS 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 4-17 contains a workflow diagram of a simple SE process model.

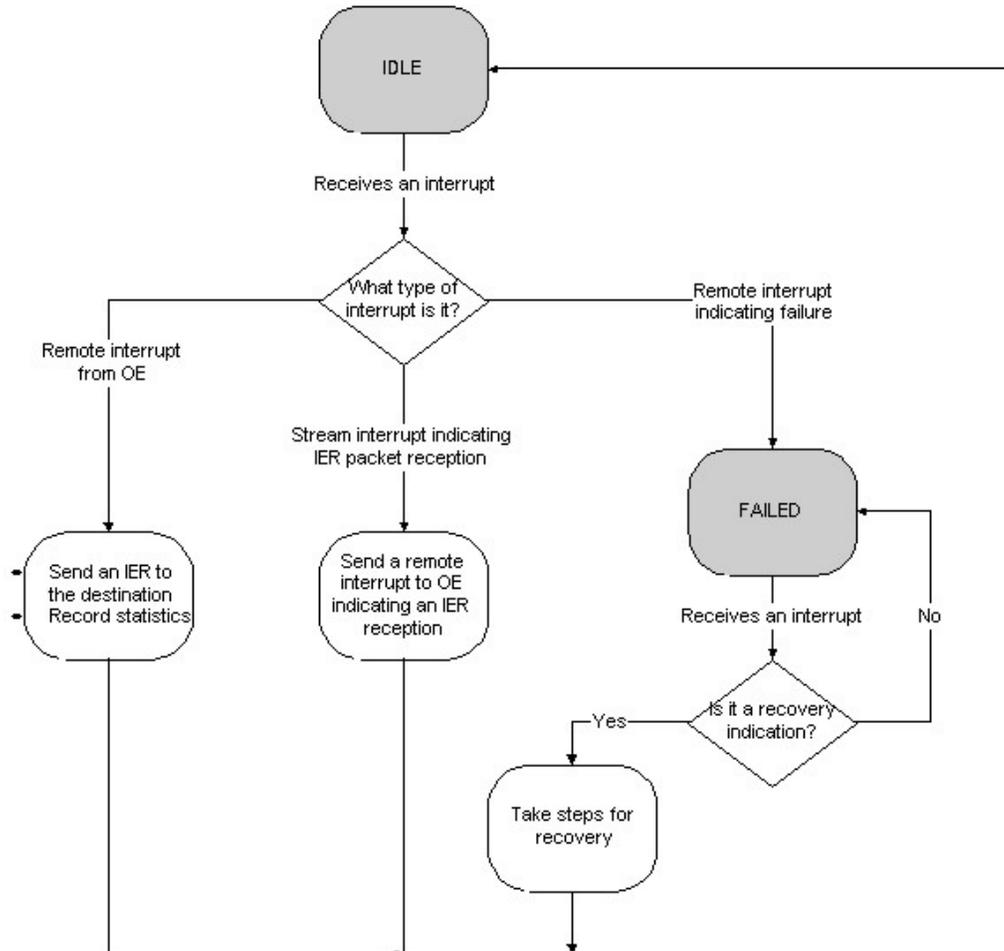


Figure 4-17: Sample Workflow Diagram for SE Process Model

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 4-17, 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.

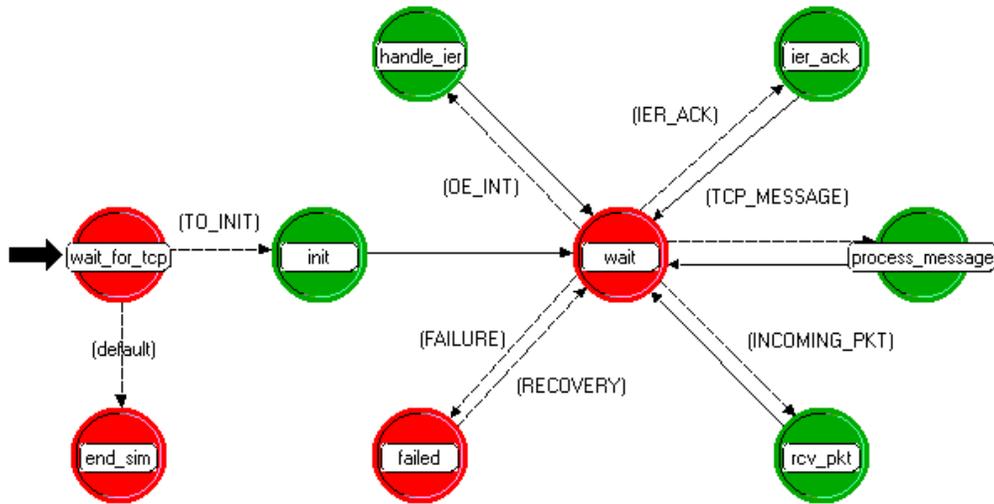


Figure 4-18: Process Model for the SE Module in the Computer

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)) != SRC_NTL) {
        notifyOE(ierInfo->ier_id, NWC_INFORM_SRC_OE_FAIL, "Unable to transmit TCP message????");
        destroyIerInfo(ierInfo);
    }
    destroyConnection(connIndex);
    break;

```

Figure 4-19: Sample Code 1—Inform OE of the IER Failure, Which Will Then Record the Statistics

Interfacing with the statistics, such as writing success and failure statistics, is normally accomplished through the APIs. Refer to the OPNET Modeler online documentation for detailed examples of how to accomplish this.

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 4-20.

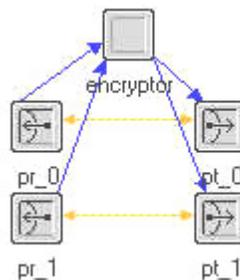


Figure 4-20: Encryptor—Node Model

4.5.3 Process Model

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

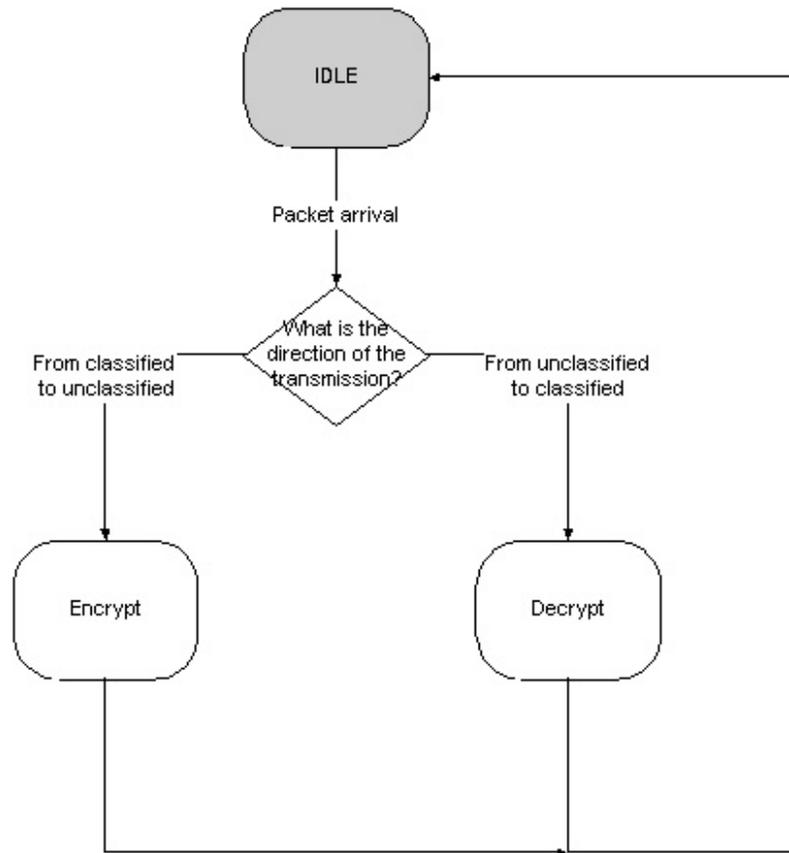


Figure 4-21: Data Flow for an Encryptor

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.

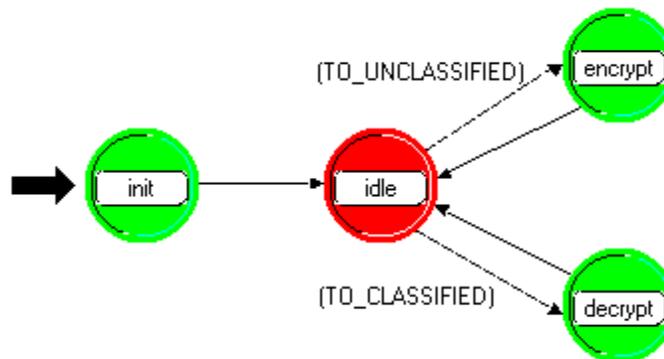


Figure 4-22: Process Model for Encryptor

Figure 4-23 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 4-23: Sample 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 JCSS version 3.1).

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 4-24:

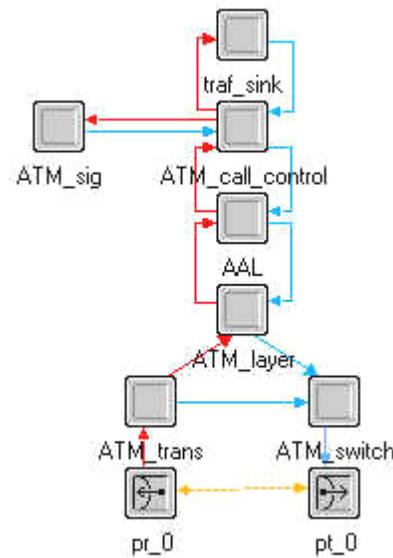


Figure 4-24: *Atm_uni_dest_adv* Switch—Node Model

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 4-25.

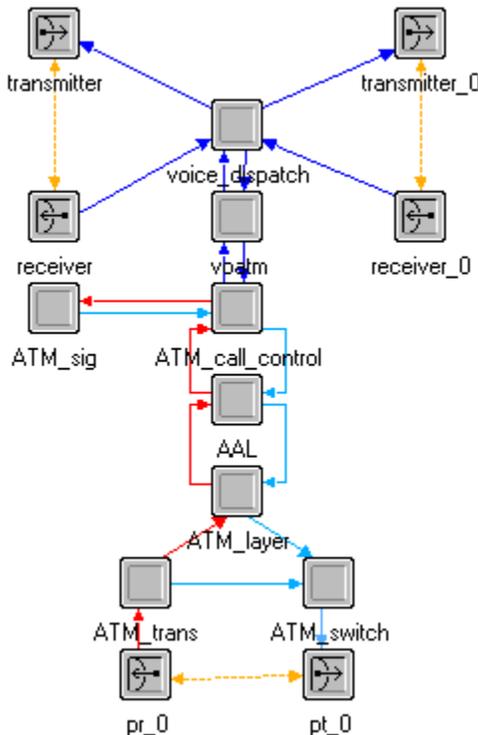


Figure 4-25: Multi-Service Switch—Node Model

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 JCSS 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. This means on the “source” side, there needs to be flooding on the other circuit switch interface.

4.7 LAYER 2 DEVICE EXAMPLE: MULTIPLEXER DEVICE USING CIRCUIT API

4.7.1 Overview

The example considered here is a circuit based multiplexer device. The objectives of this example include adding self descriptions and data description files to a custom multiplexer, so that circuits for the device can be created using the Generic Circuit Wizard.

The example networking equipment is a custom node model (found in the OPNETWORK 2008 Session 1952 Lab 2), but it is similar in nature to the *CTP_1012* node model in the JCSS device suite. The node model is shown in the following figure:

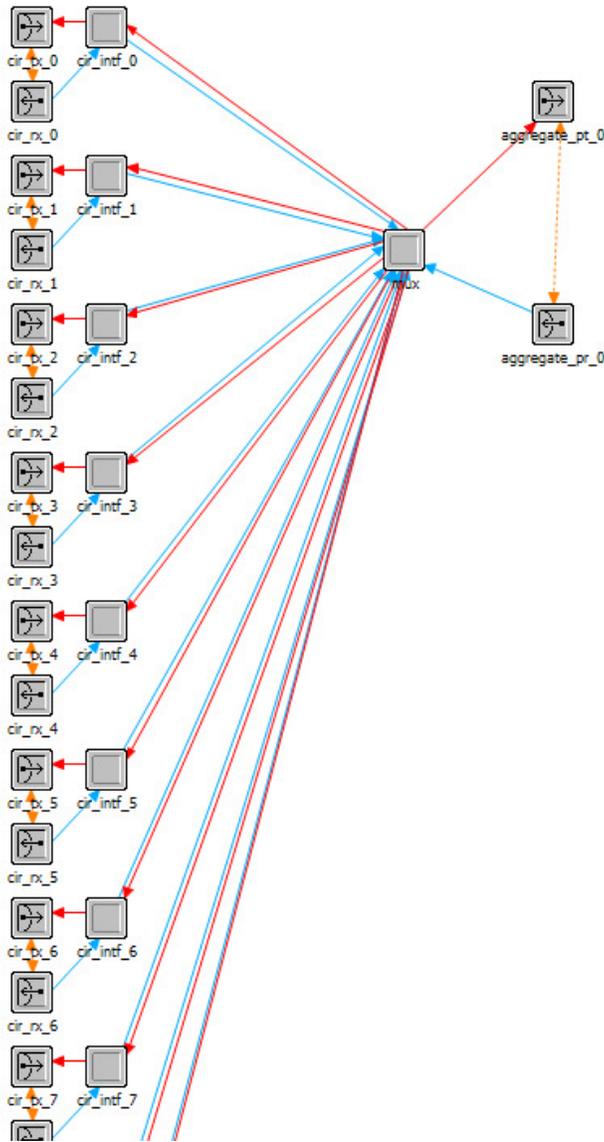


Figure 4-26: Custom Multiplexer Node Model

- The node contains 10 ports (transmitter- receiver pairs) that are named cir_tx_<n> and cir_rx_<n> to denote circuit side ports. It is recommended that the user utilize this type of naming scheme, so that users can easily determine the functionality of the port by looking at the model. These ports take in a serial bit stream which will eventually be aggregated together with other data streams.
- The node also has one aggregate port (named aggregate_pt_0 and aggregate_pr_0) which sends out multiplexed (aggregate) frames.
- The *mux* module is responsible for performing the multiplexing of serial bit streams coming from the circuit side ports and demultiplexing the aggregate frames coming from the aggregate port.
- The input of each circuit side port is fed in to the circuit interface module (one module per circuit side port). This module is an optional plug and play module which is responsible for optional features like handling circuit switched voice, generating explicit packets for voice, collecting circuit level statistics and generating full circuit load. The recommended way of naming this module is cir_intf_<n>.

4.7.2 Attributes and Process Model Code

Step 1. Choose Interfaces > Node Interfaces, observe that this node model has an attribute named *Circuit Configuration*. This is a required attribute for every circuit based device and is used by the Generic Circuit Wizard. Click on (...) to open up the (Circuit Configuration) Table (not all the rows are shown).

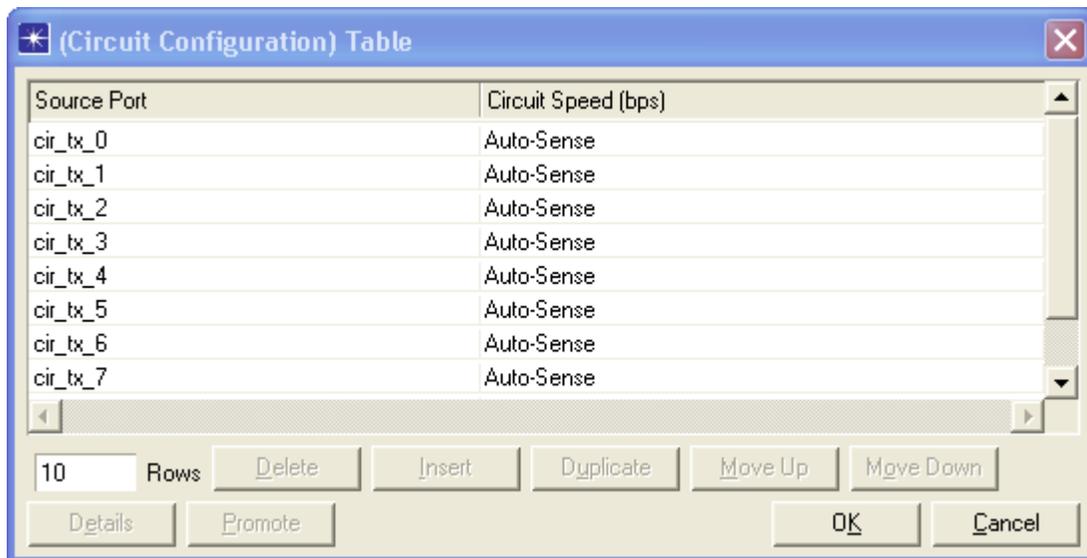


Figure 4-27: Circuit Configuration Table

- This compound attribute has one row for each circuit side port.
- Each row should have two required attributes named as Source Port (attribute data type is string) and Circuit Speed (attribute data type is double). It can also have advanced attributes (optional) for your model, but it is not needed for this example.
- The Source Port attribute should be set to the transmitter name of the circuit side port.

- The Circuit Speed attribute is set to the default value Auto-Sense (i.e., it uses the data rate of connected link).

Step 2. Observe in the Node Interface dialog box that there is a Port Configuration (P<n>) attribute for each circuit side port. Click on (...) to open up the (Port Configuration (P<n>)) Table.

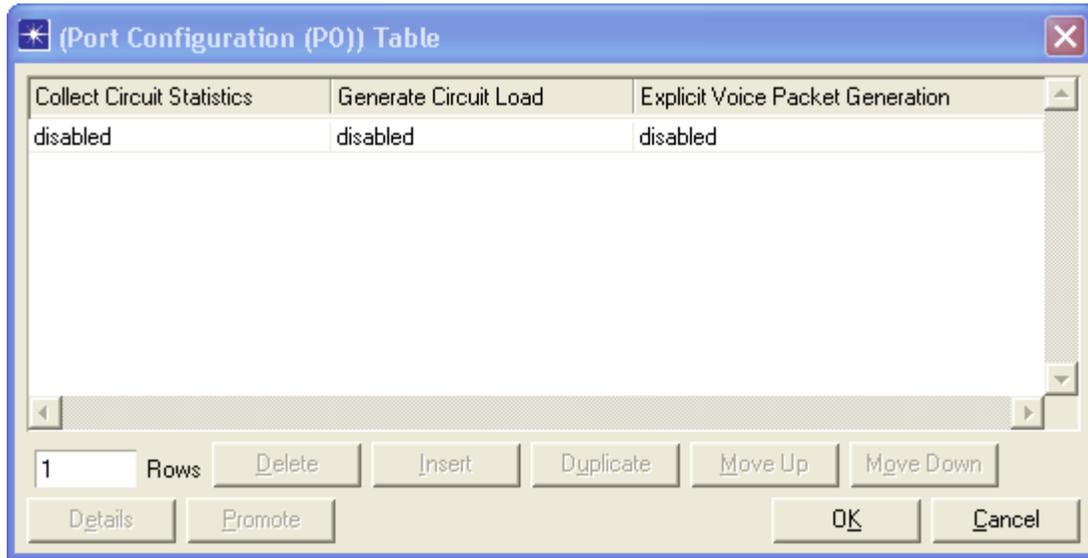


Figure 4-28: Port Configuration Table

- This table has attributes to turn ON/OFF the optional features supported by circuit interface module.

Step 3. In the Exit Execs of INIT state, observe the use of the DES circuit API:

```

/* This is the init state of the mux process that will read the port and circuit configuration for this device */

/* Obtain information about this process */
my_module_id = op_id_self();
my_node_id = op_topo_parent(my_module_id);

/* setup segmentation buffers */
segbuf = op_sar_buf_create (OPC_SAR_BUF_TYPE_SEGMENT, OPC_SAR_BUF_OPT_DEFAULT);
rsmbuf = op_sar_buf_create (OPC_SAR_BUF_TYPE_REASSEMBLY, OPC_SAR_BUF_OPT_DEFAULT);

/* Initially the busy flag */
busy = OPC_FALSE;
packet_id_count = 1;

/* Get the streams to aggregate port */
tx_obj = op_topo_assoc(my_module_id, OPC_TOPO_ASSOC_OUT, OPC_OBJTYPE_PTTX, 0);
stream_id = op_topo_assoc (tx_obj, OPC_TOPO_ASSOC_IN, OPC_OBJTYPE_STRM, 0);
op_ima_obj_attr_get(stream_id, "src stream", &aggregate_outstrm);

rx_obj = op_topo_assoc (my_module_id, OPC_TOPO_ASSOC_IN, OPC_OBJTYPE_PTRX, 0);
stream_id = op_topo_assoc (rx_obj, OPC_TOPO_ASSOC_OUT, OPC_OBJTYPE_STRM, 0);
op_ima_obj_attr_get (stream_id, "dest stream", &aggregate_instrm);

/* Create the mapping for instrm_to_remote_port and port_to_outstrm */

/* Loop through all the circuit ports */
for(index = 0; index != MAX_CIRCUIT_PORTS; index++)
{
    /* obtain the circuit interface object */
    sprintf(temp_str, "cir_intf%d", index);
    circuit_obj = op_id_from_name(my_node_id, OPC_OBJTYPE_PROC, temp_str);

    /* get the transmitter for the circuit interface */
    tx_obj = op_topo_assoc(circuit_obj, OPC_TOPO_ASSOC_OUT, OPC_OBJTYPE_PTTX, 0);
    circuit_ptr = nw_circuit_object_get(my_node_id, tx_obj);
    /* We are only interested if there is a circuit configured for this port */
    if(circuit_ptr)
    {
        /* Get my local port number */
        local_port = nw_circuit_local_port_number_get(tx_obj, circuit_ptr);

        /* Get my destination port number */
        remote_port = nw_circuit_remote_port_number_get(tx_obj, circuit_ptr);

        /* Get the streams to the circuit side */
        stream_id = op_topo_connect(my_module_id, circuit_obj, OPC_OBJTYPE_STRM, 0);
        op_ima_obj_attr_get(stream_id, "src stream", &out_strm);
        stream_id = op_topo_connect(circuit_obj, my_module_id, OPC_OBJTYPE_STRM, 0);
        op_ima_obj_attr_get(stream_id, "dest stream", &in_strm);

        instrm_to_remote_port[in_strm] = remote_port;
        port_to_outstrm[local_port] = out_strm;
    }
}

```

Above usage illustrates that any custom model which uses an nw_circuit path object to define circuits can take advantage of the functions which are part of circuit API in DES.

4.8 LAYER 3 DEVICE EXAMPLE: CUSTOM ROUTER

4.8.1 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.

4.8.2 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 4-29:

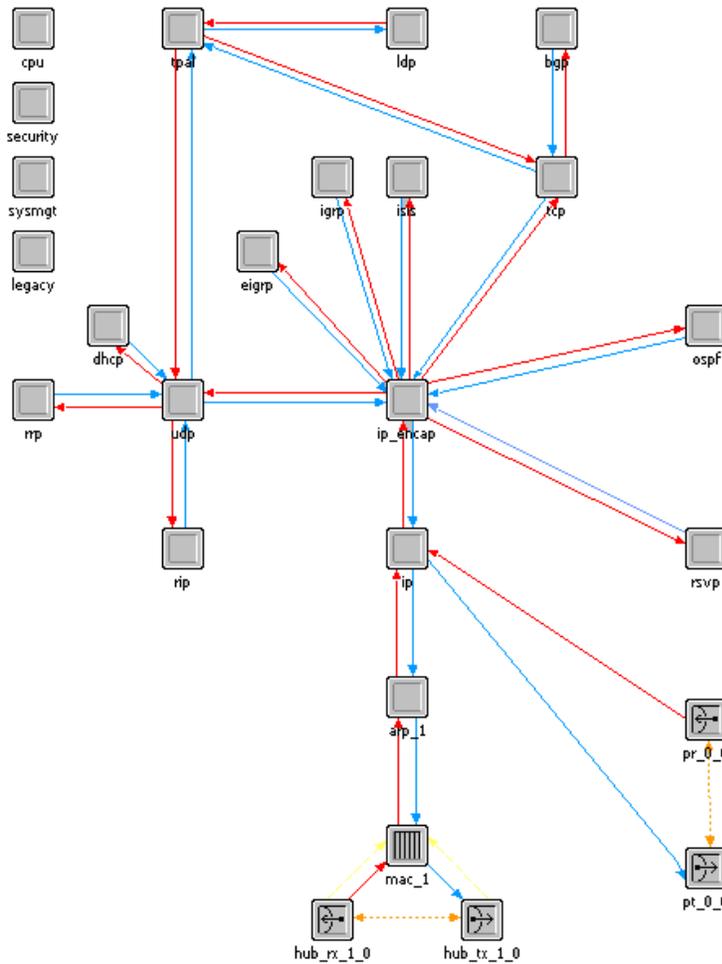


Figure 4-29: CS_1005_1s_e_sl_adv Router—Node Model

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 4-30.

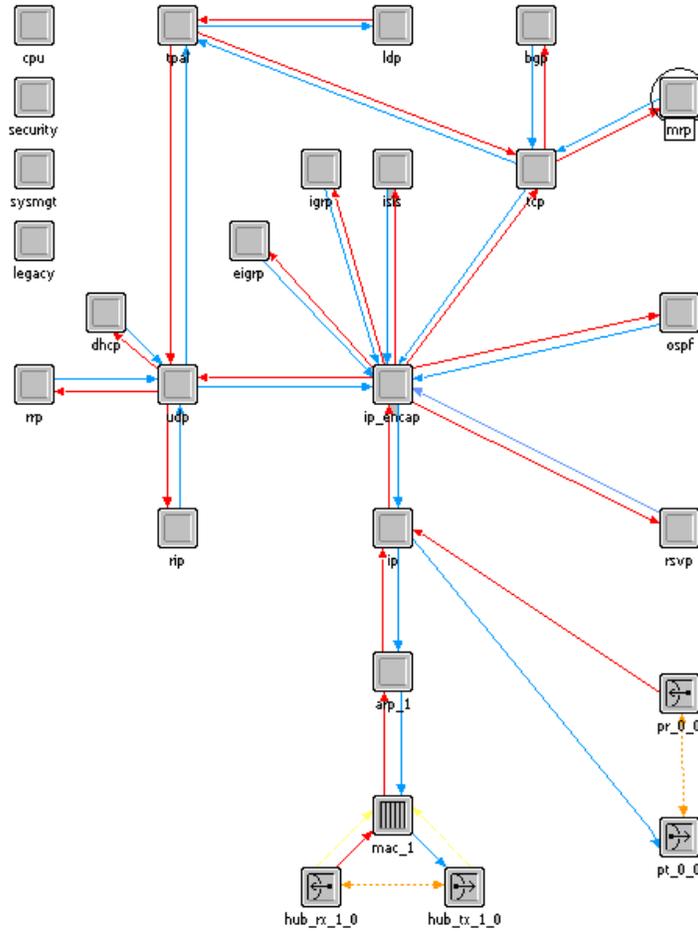


Figure 4-30: Router with Custom Routing Protocol—Node Model

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

4.8.3 Process Model: Custom Routing Protocol

The process model that implements the custom routing protocol looks like Figure 4-31.

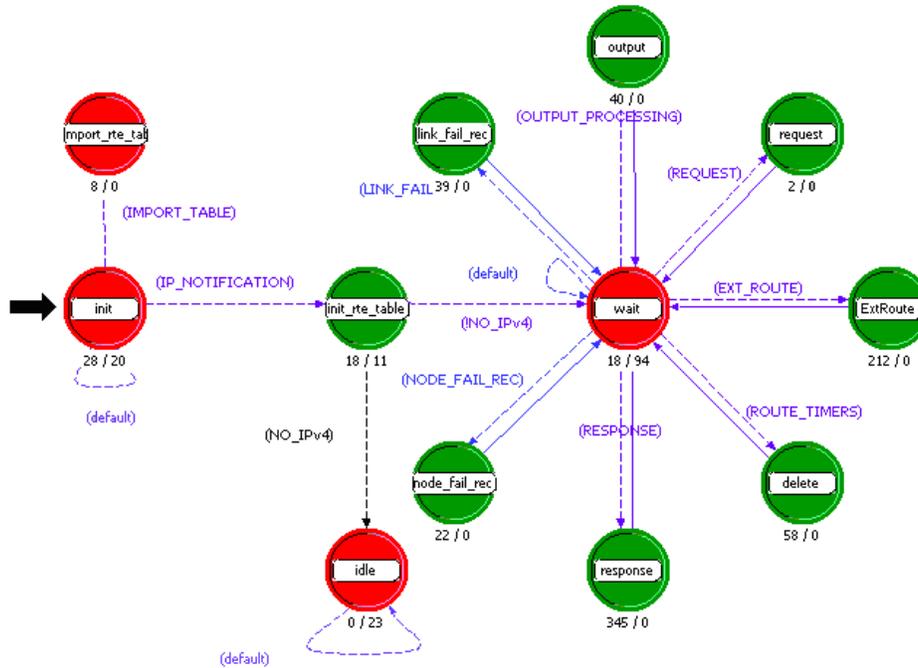


Figure 4-31: Process Model for a Custom Routing Protocol

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., 15.0.A).

4.9 CIRCUIT-SWITCHED DEVICE EXAMPLE: END SYSTEM

4.9.1 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.

4.9.2 Steps

The phone has three modules, as shown in Figure 4-32:

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

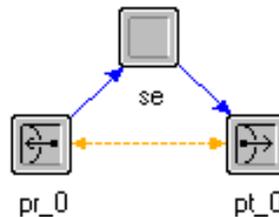


Figure 4-32: Phone—Node Model

The transmitter and the receiver are connected to the SE module by packet streams, as shown in Figure 4-32. 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.

4.9.3 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.

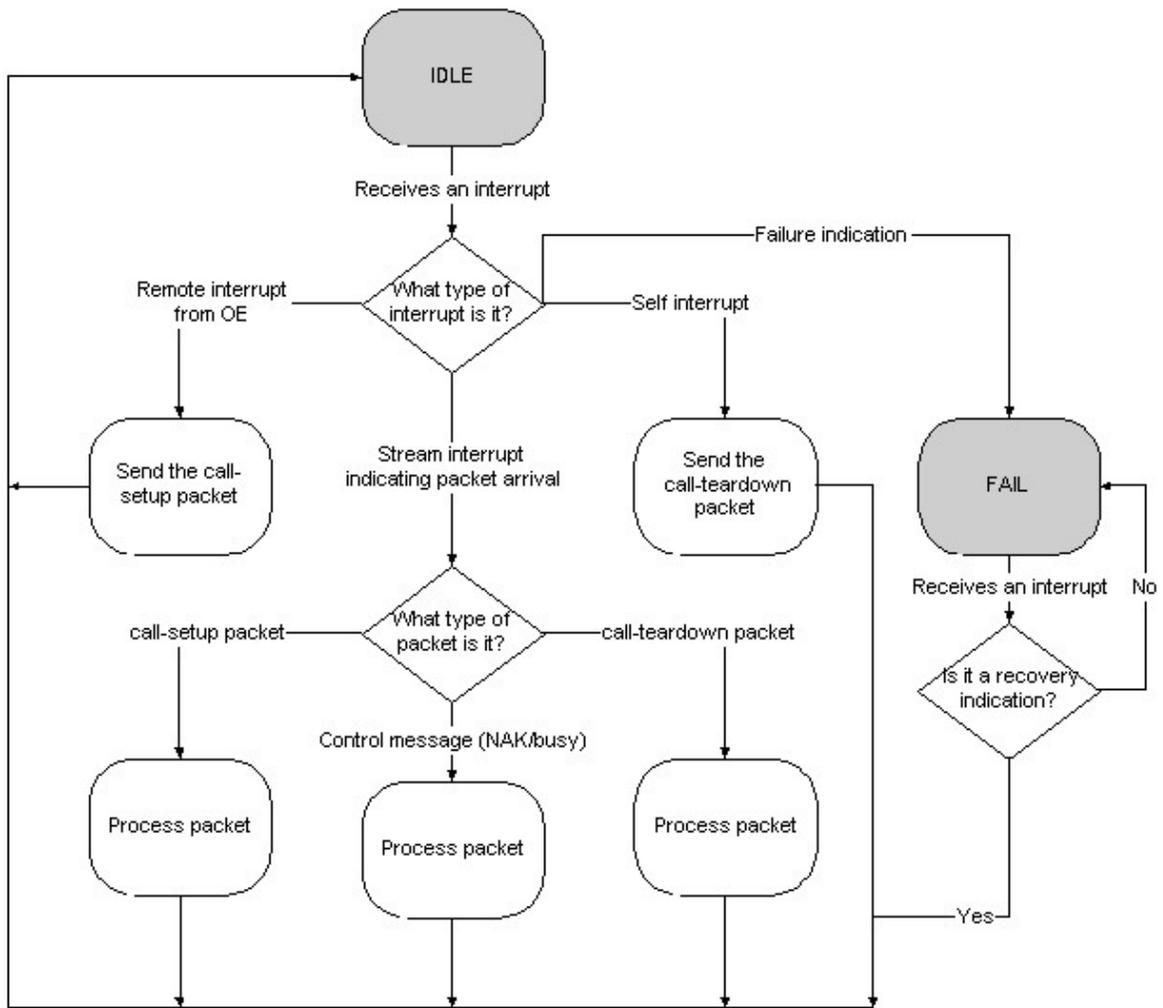


Figure 4-33: Data Flow for a Phone

Step 2. The process model for the *se* module might look like that shown in Figure 4-34.

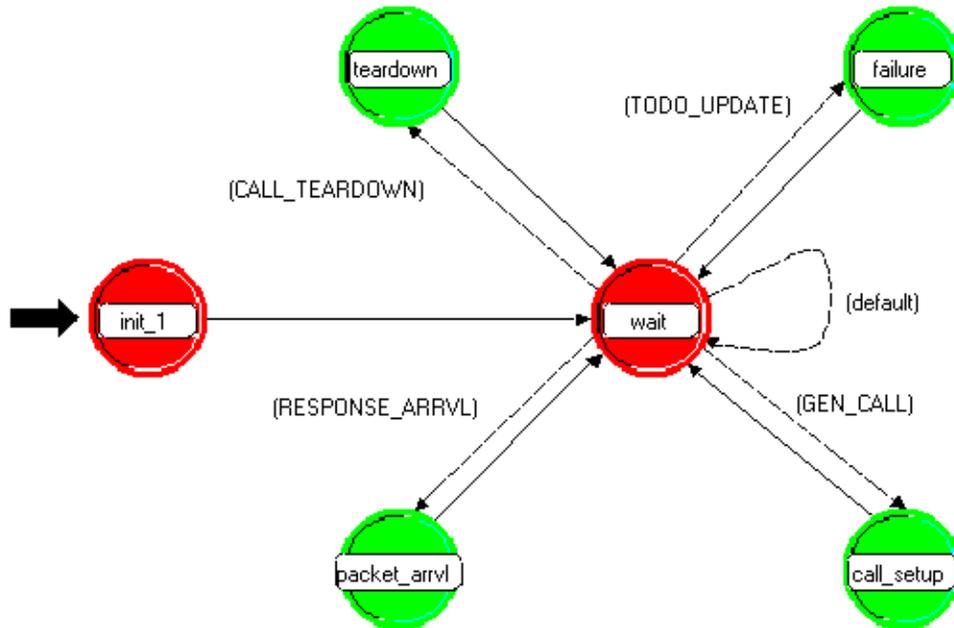


Figure 4-34: Process Model for the SE Module

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 *ier_info* 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.

4.10 WIRELESS DEVICE EXAMPLE

4.10.1 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 4-35.

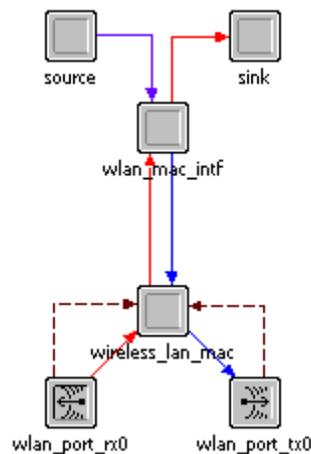


Figure 4-35: wlan_station_adv—Node Model

4.10.2 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 4-36.

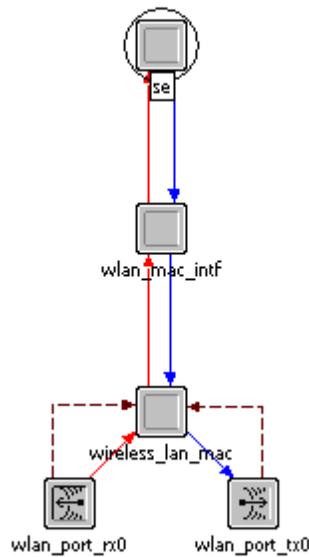


Figure 4-36: Radio SE model—Node Model

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.

4.10.3 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.

4.11 WIRELESS DEVICE EXAMPLE 2

4.11.1 Problem Statement

The following discussion provides implementation-level guidelines for developing a radio end-device JCSS 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 a radio end-device that is capable of generating both voice and data IERs.

4.11.2 High-Level Design

4.11.2.1 Node Model Development

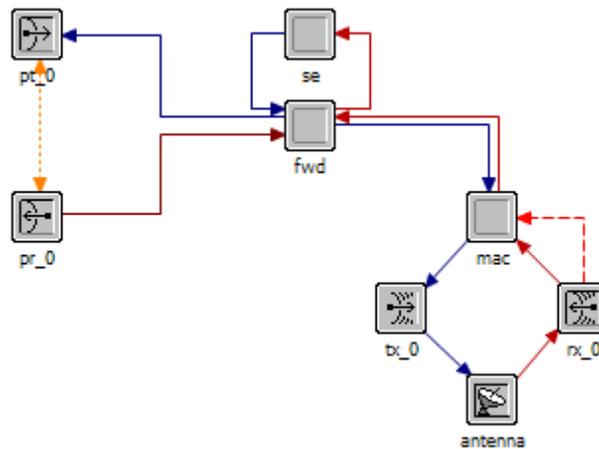


Figure 4-37. Radio End Device Node Model

The node model in Figure 4-37 above is a JCSS *pr_radio* node model and it shows a device with two interfaces—a wired interface and a radio interface. The node is also capable of generating JCSS 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.

4.11.3 fwd module: Detailed Design

4.11.3.1 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 interface modules. 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.

4.11.3.2 Events

There are three different events that can happen at this module. They are:

- Receive packet from SE
- Receive packet from pr_0 (wired interface)
- Receive packet from the MAC (radio mac).

4.11.3.3 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.

4.11.3.4 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	Receives begin_sim interrupt	Perform initialization steps, initialize state variables.	Wait
	Simulation Does not initialize	Radio is in Satellite Mode	No action	Halt
Wait	Packet arrival	Packet arrived from <i>se</i> or <i>wired_mac</i>	Forward packet to the <i>mac</i> module ³⁸	Wait
		Packet arrived from <i>mac</i>	If packet is designated to <i>se</i> , send the packet to <i>se</i> . Otherwise, forward the packet to the <i>wired_mac</i> interface.	Wait
Halt	Default	All conditions	No action	Halt

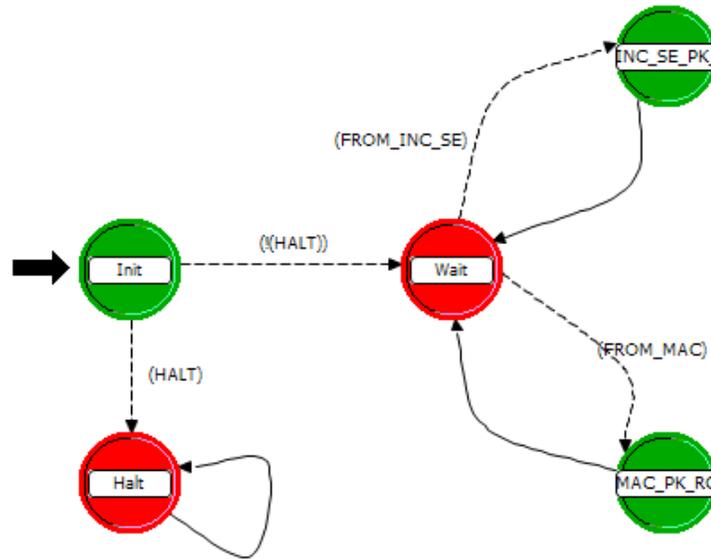


Figure 4-38: fwd Module Process Model

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.

4.11.3.5 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_format (pkt, format);
if(!strcmp (format,"ip_dgram_v4"))
{
    op_pk_send (pkt, strm_to_ptp_tx);
}
else if(!strcmp (format,"voice_packet"))
{
    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_ptp_rx || in_strm == strm_from_se))
```

In this state, packets are received from either the *pr_0* or the *se* module. These packets are sent directly to the *mac* if the radio is available.

```
if (radio_availability == 1)
{
    op_pk_send_delayed (pkt, strm_to_mac, fec_comsec);
}
else
{
    /* destroy the packet if radio is not part of broadcast network */
    op_pk_destroy(pkt);
}
```

4.11.4 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 JCSS model suite has radio models with specific MAC implementations; please refer to the PRC and EPLRS models 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.

4.11.5 se Module

4.11.5.1 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 4-39.)

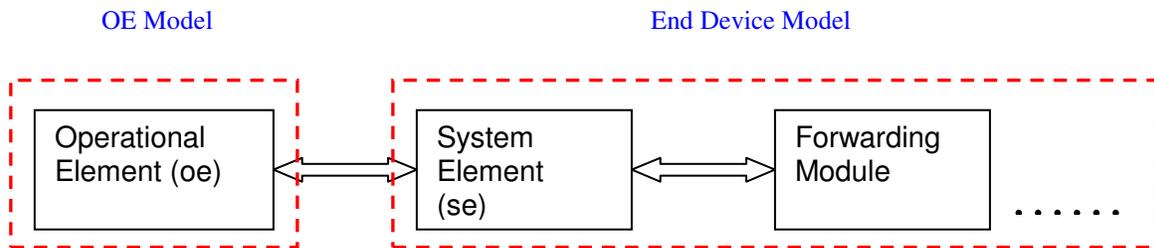


Figure 4-39: SE Module Interfaces

4.11.5.2 Events

Two events can occur at this module:

1. Packet arrival from the forwarding module. This packet signifies the reception of the IER for which this device is destined.
2. Reception of information from the *OE* to start a new IER.

4.11.5.3 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.

4.11.5.4 Event Response Table

Table 4-11: Event Response Table for the 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

4.11.5.5 Radio SE Process Model

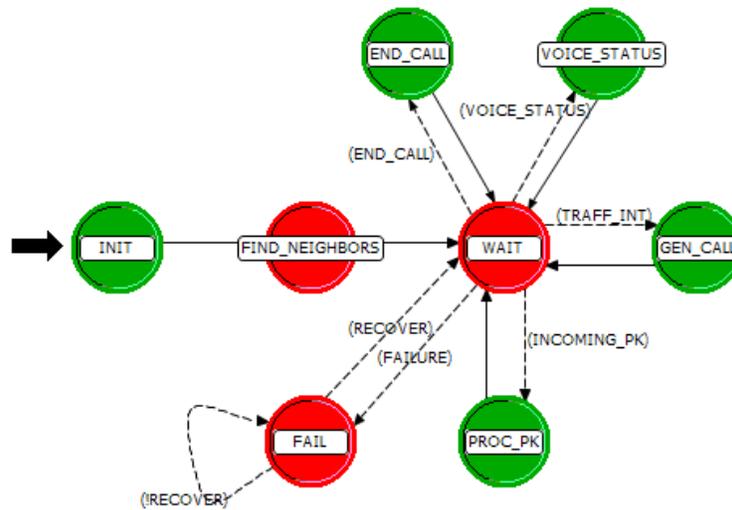


Figure 4-40: Radio SE Process Model

4.11.5.6 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 */
my_module_id = op_id_self();
my_node_id = op_topo_parent(my_module_id);

/* initialize counters for statistics */
received_voice_calls = 0;
attempted_voice_calls = 0;
failed_voice_calls = 0;

/* Register in the global registry so TPAL can find our SE */
nw_se_transport_register ();

/* get attribute values */
tx_module_id = op_id_from_name(my_node_id, OPC_OBJTYPE_RATX, "tx_0");
op_ima_obj_attr_get (tx_module_id, "net_id", &net_id);

channel_obj = op_topo_child(tx_module_id, OPC_OBJTYPE_COMP, 0);
txch_obj = op_topo_child(channel_obj, OPC_OBJTYPE_RATXCH, 0);
op_ima_obj_attr_get (txch_obj, "data rate", &data_rate);

/* obtain streams to the fwd */
fwd_module_id = op_id_from_name(my_node_id, OPC_OBJTYPE_PROC, "fwd");
temp_strm_id = op_topo_connect(my_module_id, fwd_module_id, OPC_OBJTYPE_STRM, 0);
op_ima_obj_attr_get(temp_strm_id, "src stream", &strm_to_fwd);

temp_strm_id = op_topo_connect(fwd_module_id, my_module_id, OPC_OBJTYPE_STRM, 0);
op_ima_obj_attr_get(temp_strm_id, "dest stream", &strm_from_fwd);

```

Gen_Call State Implementation:

The control reaches this state if the se receives a remote interrupt from the OE:

```
#define TRAFF_INT (intrpt_type == OPC_INTRPT_REMOTE && \
    intrpt_code == NwC_Ier_Key_Remote_Intrpt_Code_Fire_Ier_Instance)
```

First retrieve the IER parameters from the ICI associated with the remote interrupt. Make sure that the interrupt code used by the OE is “TRAFF_INT”.

```
if (oe_ici)
{
    if (NwC_Ier_Key_Remote_Intrpt_Code_Fire_Ier_Instance == intrpt_code)
    {
        /* Get IER keys and consumer information */
        op_ici_attr_get (oe_ici, NwC_Ier_Key_Field_Name, &ier_key);
        op_ici_attr_get (oe_ici, NwC_Ier_Instance_Key_Field_Name, &ier_instance_key);
        op_ici_attr_get (oe_ici, NwC_Consumer_Objid_Field_Name, &dest_nd_id);
    }
}
```

To generate the IER:

- 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.
- 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
- Send the packet out to the “fwd” module.

```
num_consumers = nw_ier_support_get_num_consumers (ier_key);
call_duration = nw_ier_support_get_size (ier_key, ier_instance_key);

attempted_voice_calls += num_consumers;

op_stat_write(global_stats.attempted_calls, attempted_voice_calls);
op_stat_write(local_stats.attempted_calls, attempted_voice_calls);

if(voice_busy)
{
    failed_voice_calls += num_consumers;
    op_stat_write(global_stats.failed_calls, failed_voice_calls);
    op_stat_write(local_stats.failed_calls, failed_voice_calls);
    FRET(OPC_TRUE);
}

/* place the IER in a new voice packet */
new_packet = op_pk_create_fmt(NWC_VOICE_PACKET);

op_pk_nfd_set(new_packet, "packet type", PACKET_FROM_SE);

/* Put the keys in the packet */
op_pk_nfd_set (new_packet, NwC_Ier_Key_Field_Name, ier_key);
op_pk_nfd_set (new_packet, NwC_Ier_Instance_Key_Field_Name, ier_instance_key);

op_pk_total_size_set(new_packet, data_rate * call_duration);

prc_voice_busy_alert(OPC_TRUE);

/* Aptracking */
nw_ier_support_tag_aptracking_info (ier_key, ier_instance_key, new_packet);

/* send out the voice packet */
op_pk_send(new_packet, strm_to_fwd);
```

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.”

Note here that if the IER is to be “multicast” to more than one destination, then the destination node should check whether it is one of the consumers. This destination list should be checked when processing the IER at the reception end.

Proc_Pk State Implementation:

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 use the IER API to inform to the OE in the OPFAC about the received IER.

```
pkt = op_pk_get (op_intrpt_strm ());
/* Get the IER keys from the packet */
op_pk_nfd_get (pkt, NwC_Ier_Key_Field_Name, &ier_key);
op_pk_nfd_get (pkt, NwC_Ier_Instance_Key_Field_Name, &ier_instance_key);
if (nw_ier_support_check_if_consumer_device (ier_key, ier_instance_key, my_node_id))
{
    /* Apptracking */
    apptrack_pk_reception_record (pkt);

    op_stat_write(local_stats.received_calls, ++received_voice_calls);
    op_stat_write(global_stats.received_calls, received_voice_calls);

    /* Inform about successful reception of the IER */
    nw_ier_support_inform_ier_received (ier_key, ier_instance_key, my_node_id);
}
op_pk_destroy (pkt);
```

4.11.5.7 ICI and Packet Formats

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 PRC radio creates a packet of format “voice_packet.”

4.11.6 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.

4.11.7 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.

4.12 SATELLITE TERMINAL GENERIC EXAMPLE

4.12.1 Node Model Contents

A generic satellite terminal, as it is termed in the context of JCSS, 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.

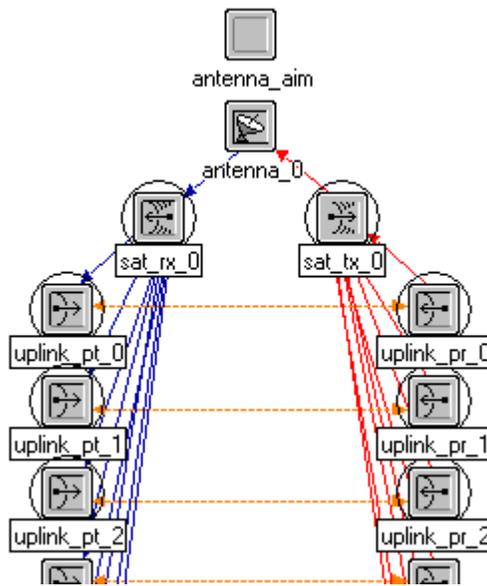


Figure 4-41: Generic Satellite Terminal

4.12.2 Core Self-Description Attributes

Nodal Mode should have the value “Generic.”

Supported Bands should have the value “Ku,X,C,Ka.”

4.12.3 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.

4.12.4 Antenna Aim Process

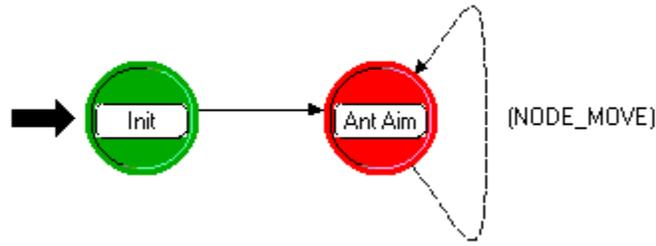


Figure 4-42: Antenna Aim Process

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.

4.12.5 Description of Antenna Aim Process

Ant Aim Enter Executives 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 ()`.

Init Enter Executives 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.

4.13 SATELLITE TERMINAL WITH TSSP EXAMPLE

4.13.1 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 regarding TSSP and nodal versus non-nodal, consult the Chairman of the Joint Chiefs of Staff Manual (CJCSM) 6231 and Military Standard (MIL-STD)-188-168 documentation.

4.13.2 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.

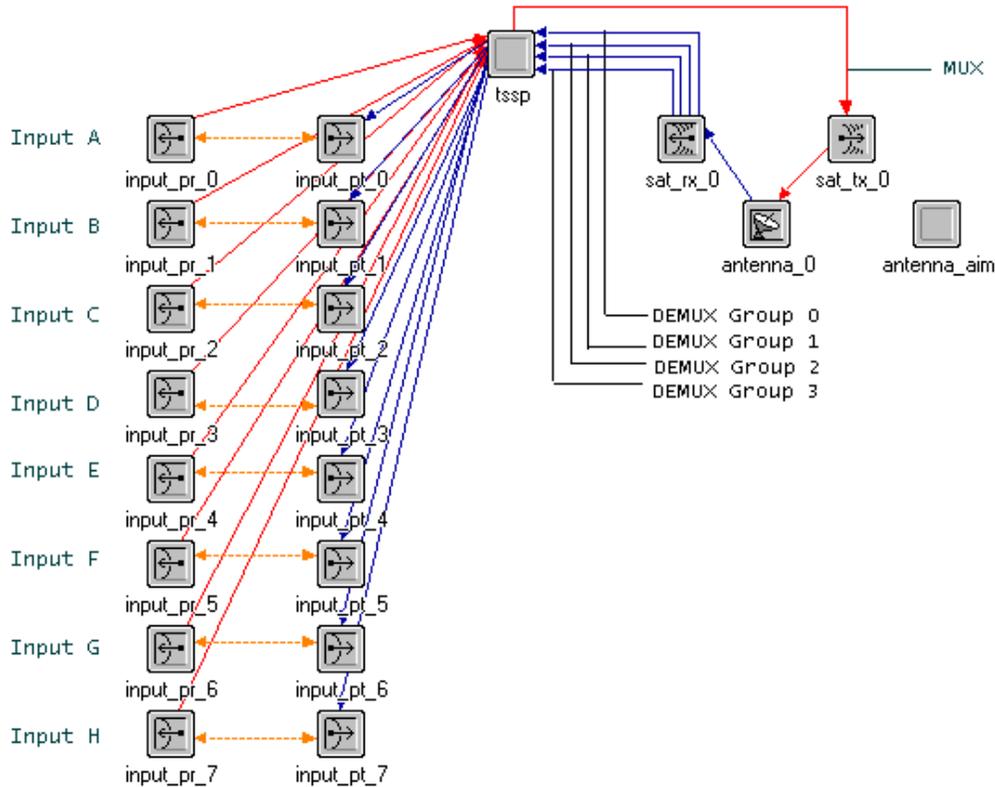


Figure 4-43: TSSP Satellite Terminal

4.13.3 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”.

4.13.4 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.

4.13.5 Node Model Specific Configuration

4.13.5.1 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 *Circuit Configuration*. *Channel Config* has the attributes that characterize a channel, and *Circuit Configuration* has the attributes that define TSSP circuit configurations. The TSSP circuits discussed are using the Generic Circuit API described in section 3.14.1.

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 4-44 below for an example of how to configure these attributes of a nodal TSSP satellite terminal.

Example Configuration: TSSP Nodal Terminals

Channel Config | Downlink (compound)

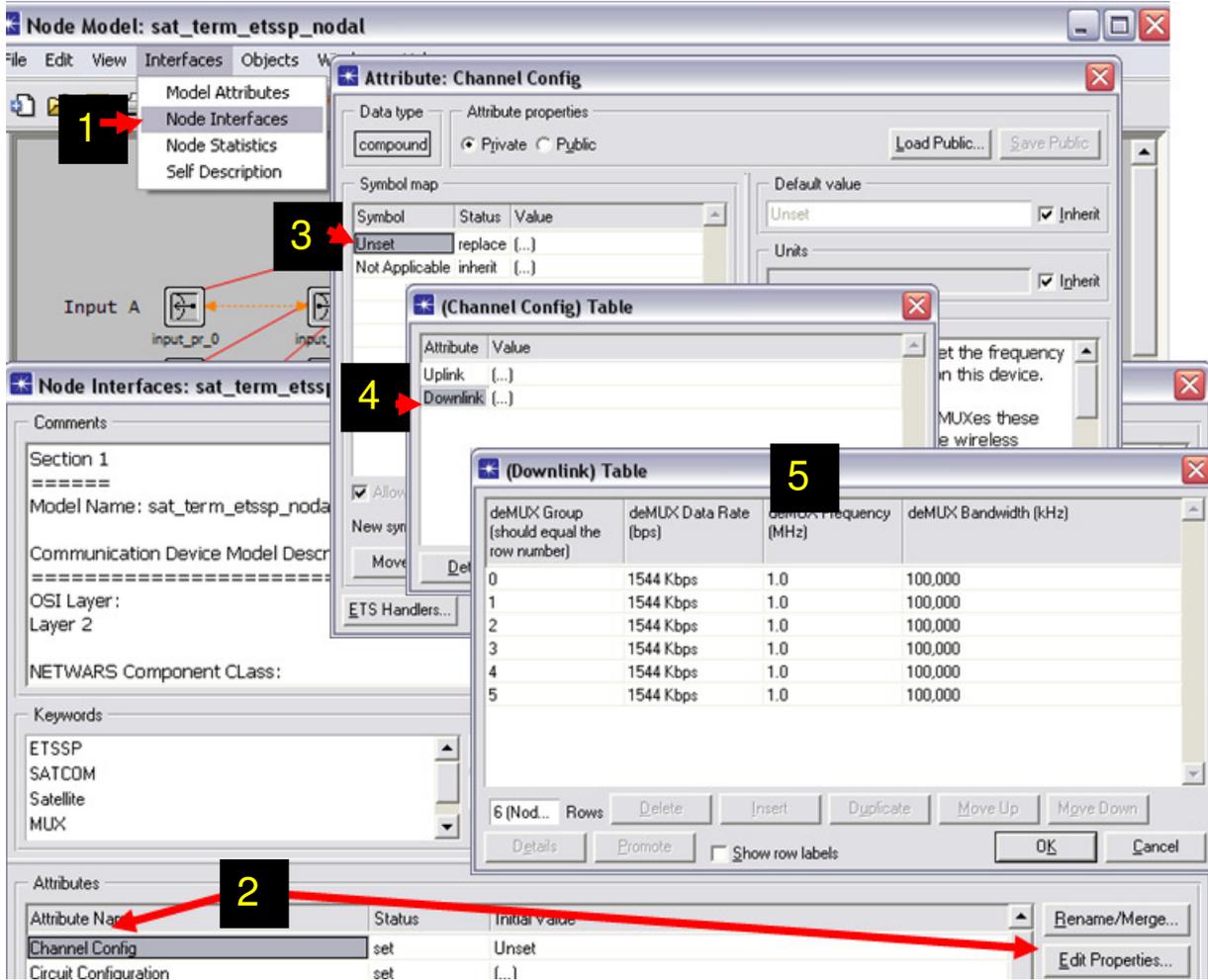


Figure 4-44: Example Configuration—TSSP Nodal Terminals – Channel Config – Downlink Example

The downlink attribute should have exactly six rows. Each row corresponds to a deMUX Group.

Circuit Configuration (compound)

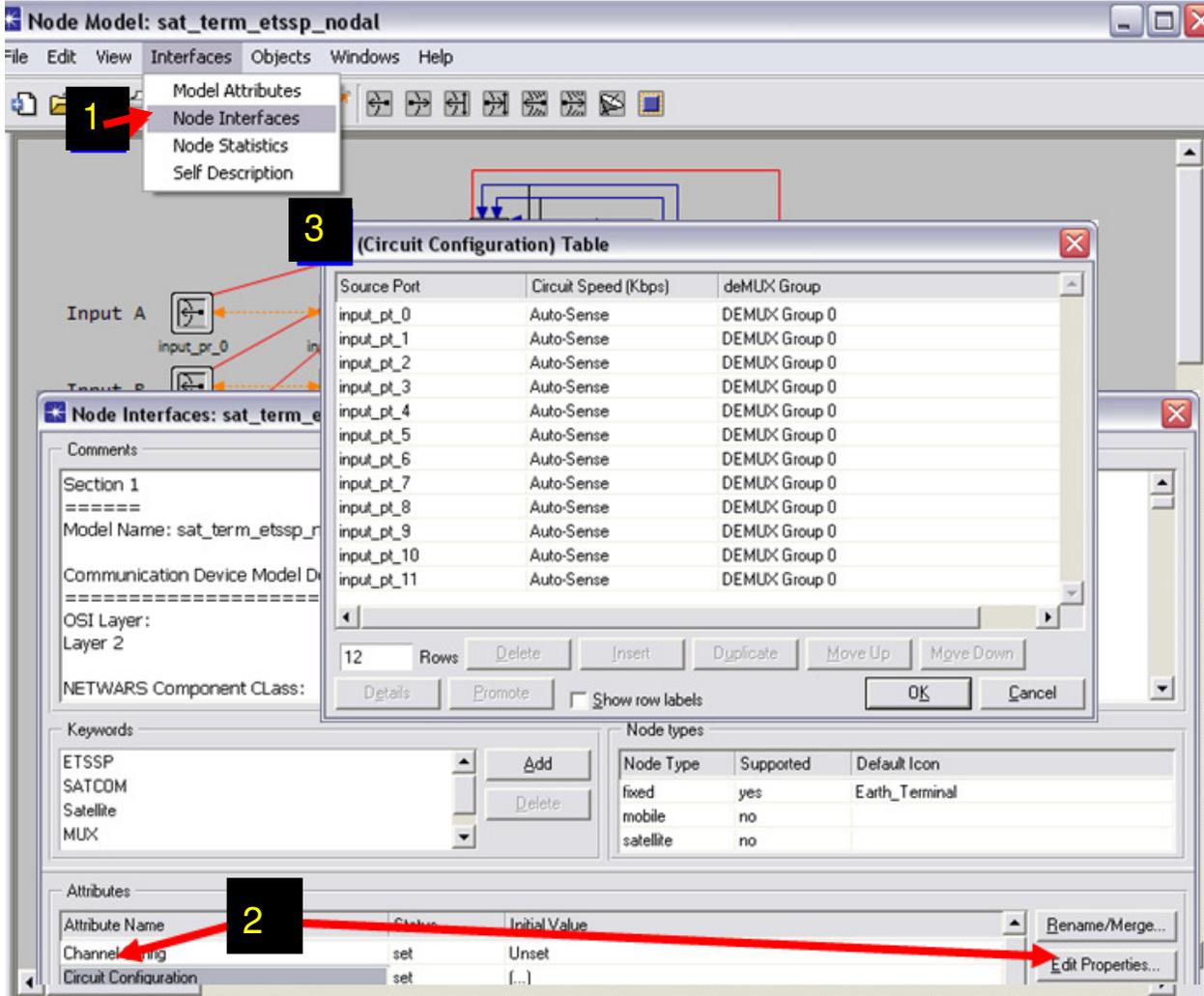


Figure 4-45: Example—Circuit Configuration – Source ports

This should have exactly twelve rows. Each row corresponds to a Source Port.

Table 4-12: Satellite Terminal Settings Table

Configuration	Attribute Settings
TSSP Nodal Terminals	<p><i>Channel Config Downlink (compound)</i> This should have exactly four rows. Each row corresponds to a deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly eight rows. Each row corresponds to an input port group.</p>

Configuration	Attribute Settings
TSSP Non-Nodal Terminals (8 Inputs)	<p><i>Channel Config Downlink (compound)</i> This should have exactly one row. The row corresponds to the single available deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly eight rows. Each row corresponds to an input port group.</p>
ETSSP Nodal Terminals	<p><i>Channel Config Downlink (compound)</i> This should have exactly six rows. Each row corresponds to a deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly twelve rows. Each row corresponds to an input port group.</p>
ETSSP Non-Nodal Terminals w/ 8 Inputs	<p><i>Channel Config Downlink (compound)</i> This should have exactly one row. The row corresponds to the single available deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly 12 rows. Each row corresponds to an input port group.</p>
ETSSP 3G Nodal Terminals	<p><i>Channel Config Downlink (compound)</i> This should have exactly six rows. Each row corresponds to a deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly 12 rows. Each row corresponds to an input port group.</p>
ETSSP 3G Non-Nodal Terminals w/ 8 Inputs	<p><i>Channel Config Downlink (compound)</i> This should have exactly one row. The row corresponds to the single available deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly 12 rows. Each row corresponds to an input port group.</p>
All Non-Nodal Terminals w/ 2 Inputs	<p><i>Channel Config Downlink (compound)</i> This should have exactly one row. The row corresponds to the single available deMUX group.</p> <p><i>Group Configuration (compound)</i> This should have exactly two rows. Each row corresponds to an input port group.</p>

4.13.6 TSSP Process

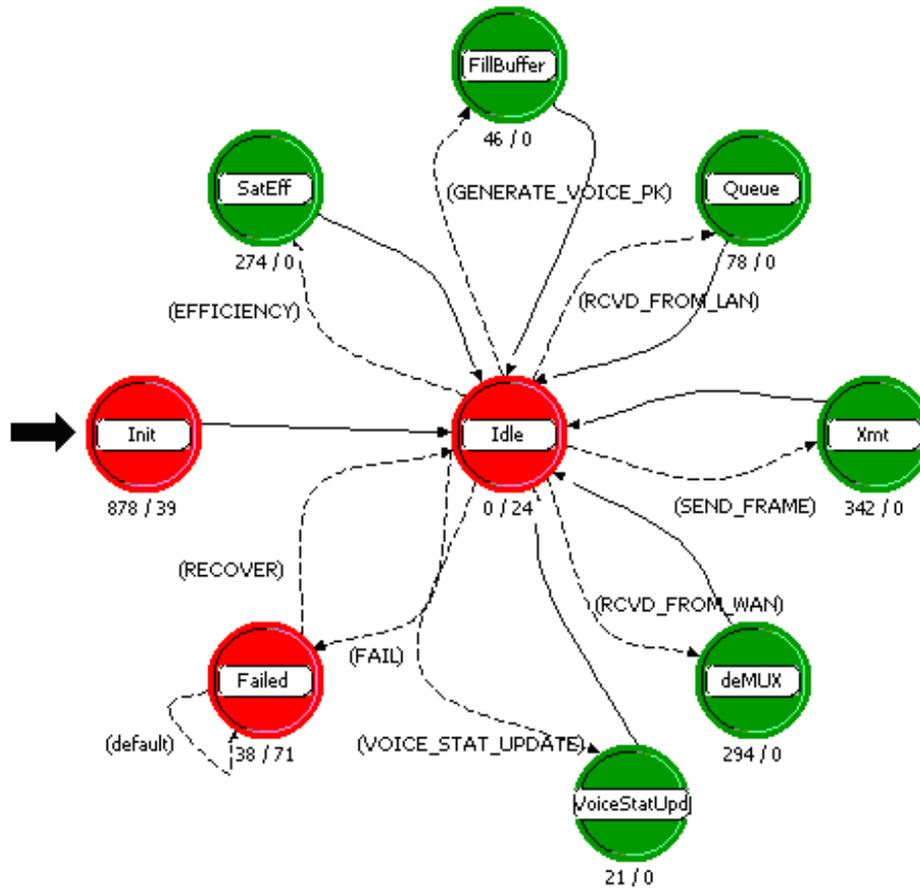


Figure 4-46: TSSP Process Model

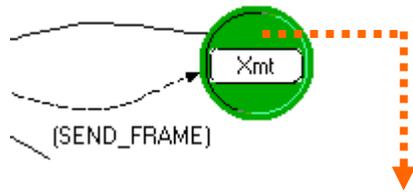
Table 4-13: Events of the TSSP Process Model

Current State	Event	Condition	Action	Next State
Init	Simulation start	None	Perform initialization	Idle
Idle	Self Interrupt	Interrupt code = TsspC_Intrpt_SatEff	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

4.13.7 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 this 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;

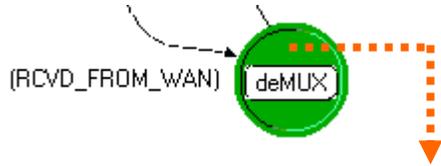
// Get a segment from this Input's SAR buffer.
seg_pkptr = op_sar_srcbuf_seg_remove (sv_input_port [i].segbuf_hndl,
    seg_size);

// 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_hnd1, 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_hnd1)))
            {
                // 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.

4.14 SATELLITE GENERIC EXAMPLE

4.14.1 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 JCSS requires following some basic conventions. Before reading this subsection, be sure to read the subsection “Building Wireless Interfaces” in Section 3, Building JCSS 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 JCSS Standard Model Library.

4.14.2 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.”

4.14.3 Additional Attributes

At its most fundamental level, a satellite model must have some basic attributes that define that model as a satellite node in JCSS. 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.

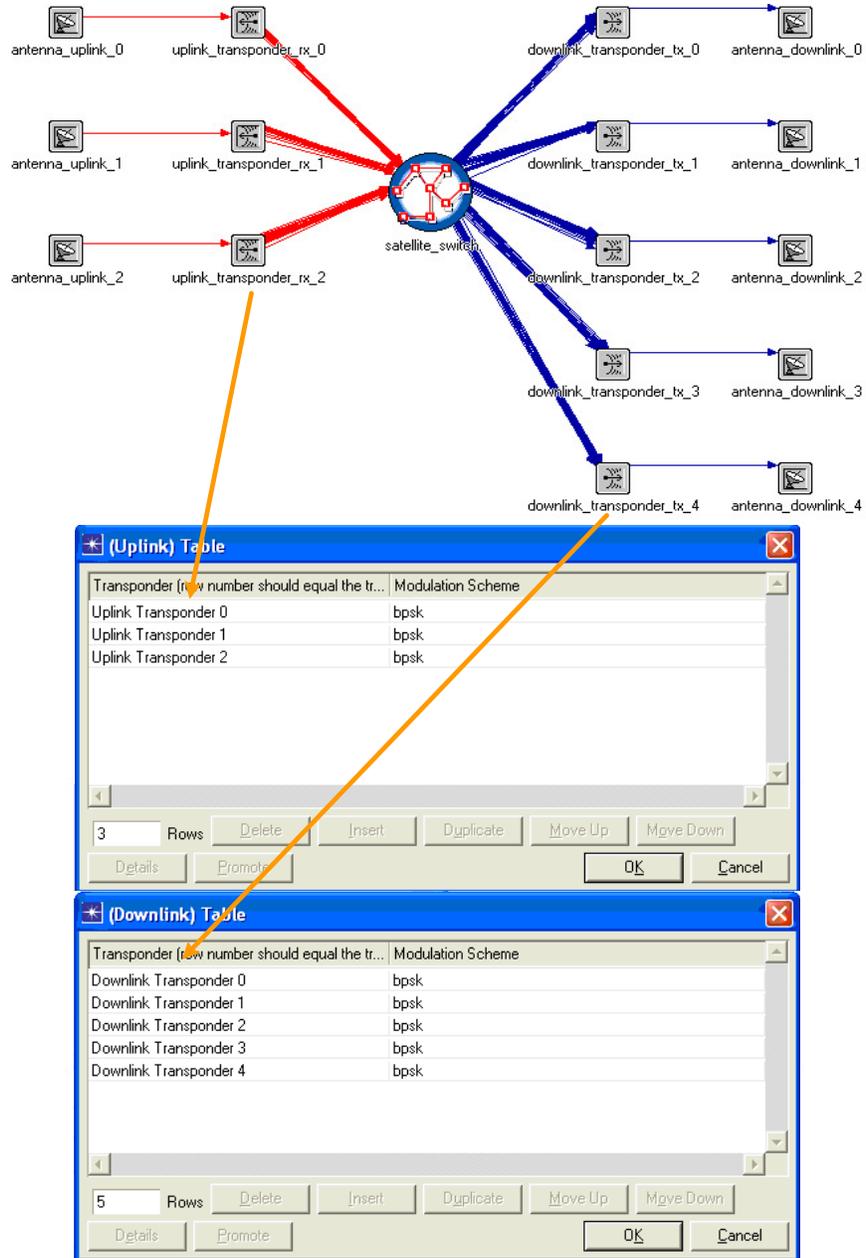


Figure 4-47: Uplink and Downlink Tables

4.14.4 Satellite Switch Process

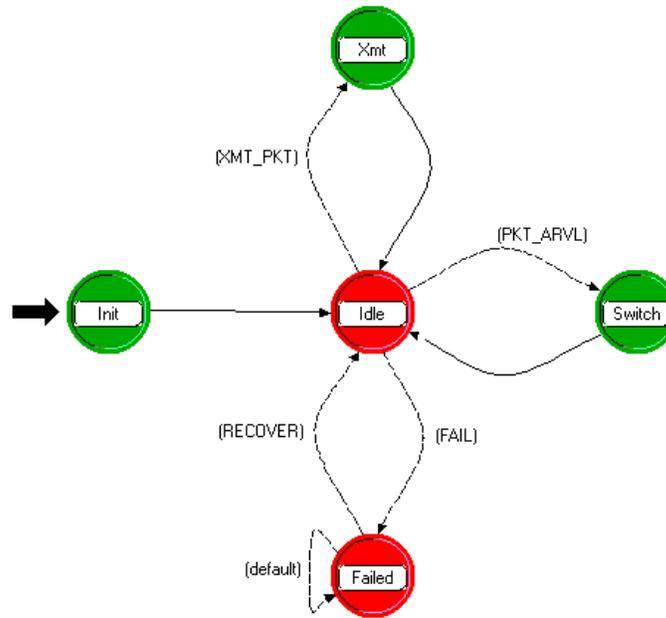


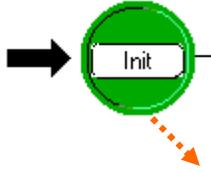
Figure 4-48: Satellite Switch Process Model

Table 4-14: Events of the 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 downlink 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 channel 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);

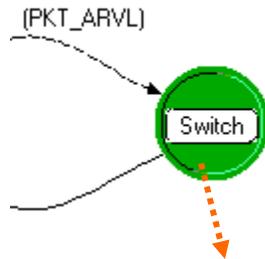
    // Get the uplink channel Objid.
    op_ima_obj_attr_get (ra_mod_id, "channel", &chnl_id);
    chnl_row_id = op_topo_child (chnl_id, OPC_OBJMTYPE_ALL, sw_tbl_entry.dl_chnl);
    op_ima_obj_attr_get (chnl_row_id, "min frequency", &sw_tbl_entry.dl_freq);

    // 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 (%.1fMHz) --> packet %d --> dl trasnp %d ch %d (%.1fMHz)\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.

4.15 LINK MODEL EXAMPLE

4.15.1 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.

4.15.2 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.

4.15.3 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 4-49: Sample Code 3—Adding Signaling Overhead to the 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.

4.16 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 Wireless Configuration Utility Node, which is used for the purpose of specifying failure/ recovery and start/stop times of the broadcast networks and the wireless links. Only a high-level overview is given below. For further details, consult the JCSS Standard model called *Wireless_Configuration.nd.m* and its process model *wireless_config.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 4-50 below.



Figure 4-50: Wireless Configuration Utility Node—Node Model

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 Wireless Configuration utility does all of this using a single *BEGSIM* interrupt.

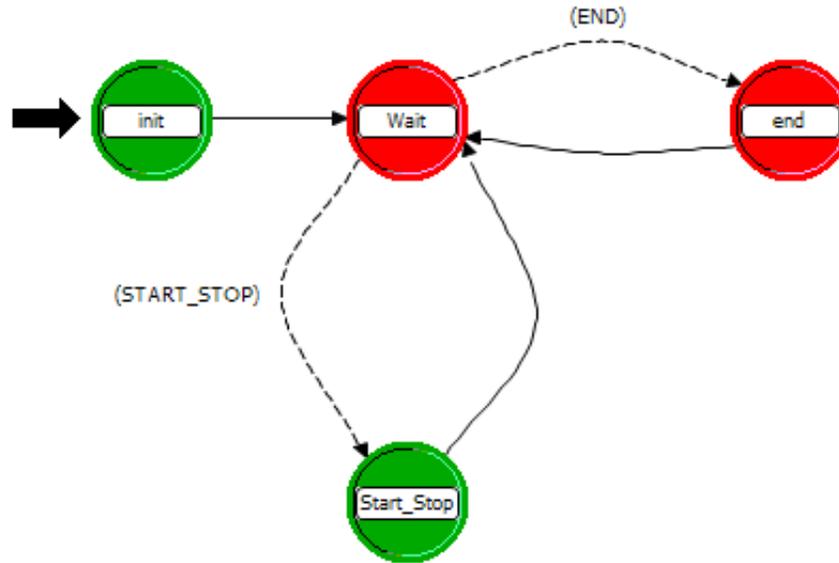


Figure 4-51: Wireless Configuration Object—Process Model

The “end” state is used to ensure errors are not incurred if this device sees an event. The code in the Enter Executives of the “Start_Stop” state performs all of the actions of this object, as seen in the following code sample:

```

// Find if the device channel has to be failed or recovered
if (wireless_entry->status)
{
    // This means the device channel has to be enabled: meaning the min frequency for this
    // channel has to be reset to the original value
    if (rtx_found && (ratx_objid != OPC_OBJID_INVALID))
    {
        op_ima_obj_attr_get (ratx_objid, "channel", &channel_objid);
        row_objid = op_topo_child (channel_objid, OPC_OBJTYPE_RATXCH, (int) wireless_entry->channel_number);

        // Get the frequency value for the device
        fl_count = op_prg_list_size (freq_list);
        for (fl_index = 0; fl_index < fl_count; fl_index++)
        {
            freq_list_entry = op_prg_list_access (freq_list, fl_index);
            if ((freq_list_entry->ch_objid == channel_objid) &&
                (freq_list_entry->ch_num == wireless_entry->channel_number))
            {
                if (freq_list_entry->frequency != 0.0)
                {
                    op_ima_obj_attr_set (row_objid, "min frequency", freq_list_entry->frequency);
                }
            }
            else
            {
                op_sim_end ("Are we failing this broadcast network multiple times without recovering?",
                    OPC_NIL, OPC_NIL, OPC_NIL);
            }
        }
        break;
    }
}
}
}

```

Figure 4-52: Wireless 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.

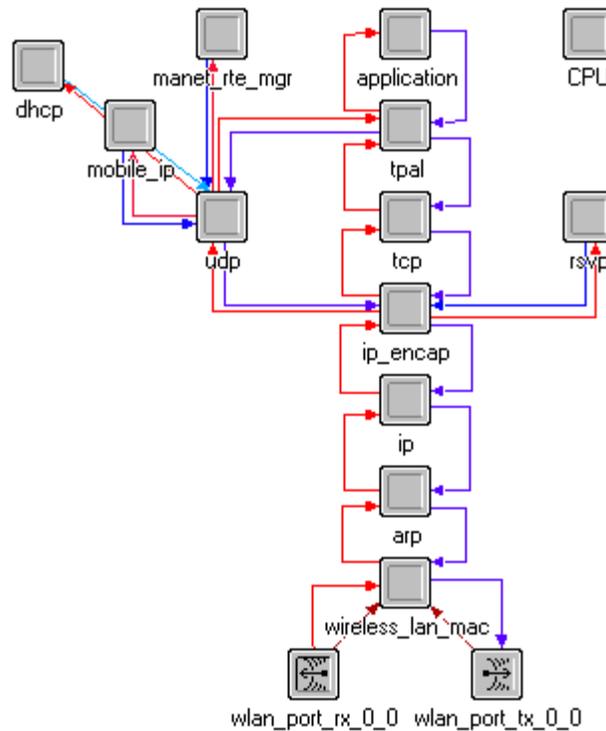


Figure 4-53: Sample Node Model

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 *redibilient_type*.

[End-System Compliance]

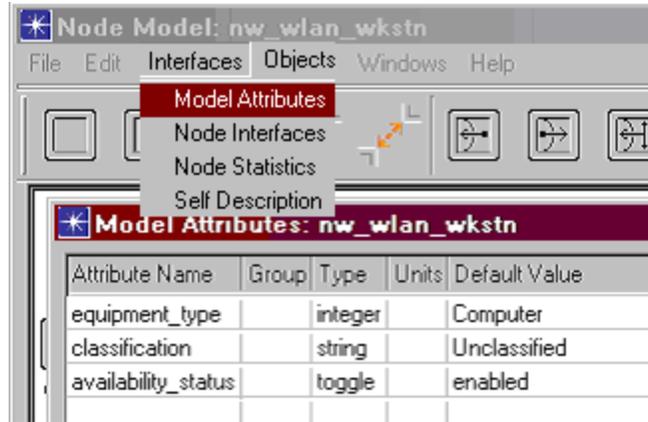


Figure 4-54: Selecting “Computer” for equipment_type

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]

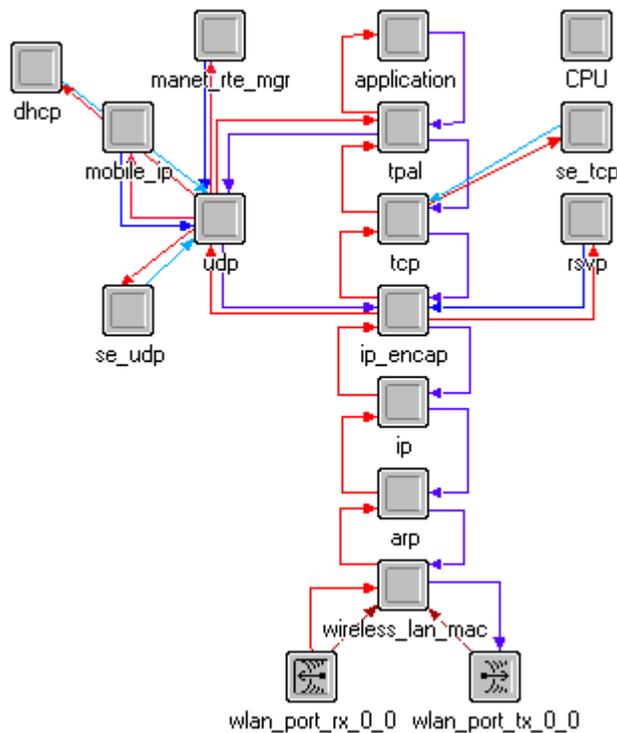


Figure 4-55: Adding *se_tcp* and *se_udp*

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]

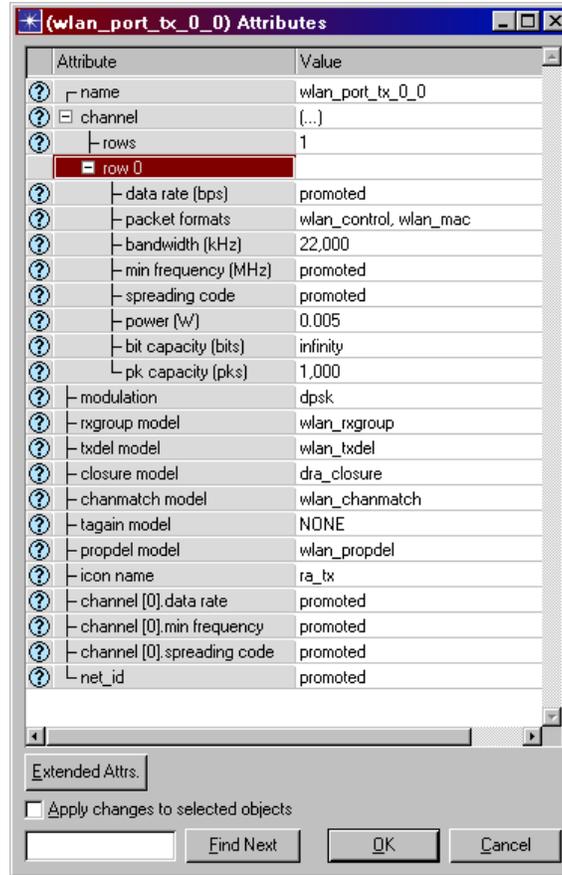


Figure 4-56: Adding the *net_id* Extended Attribute

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, JCSS applies analytical techniques to rapidly determine the bandwidth requirements to support specific traffic profiles and patterns. JCSS 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 JCSS, 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 PRC radio model in JCSS.

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 4-57 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 4-57.

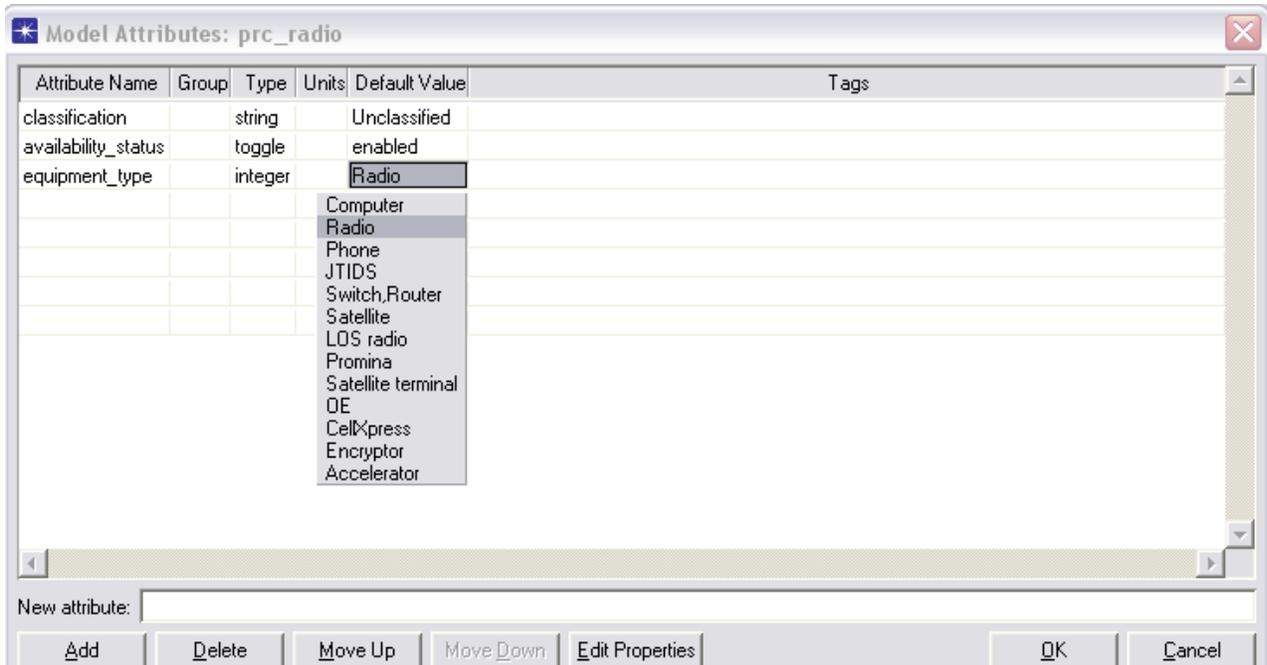


Figure 4-57: Equipment type attribute example

4.18.2.2 Interface Class and Machine Type Attributes

The interface class and machine type attributes are both locating in the self-description section of the device model as shown in the Figure 4-58.

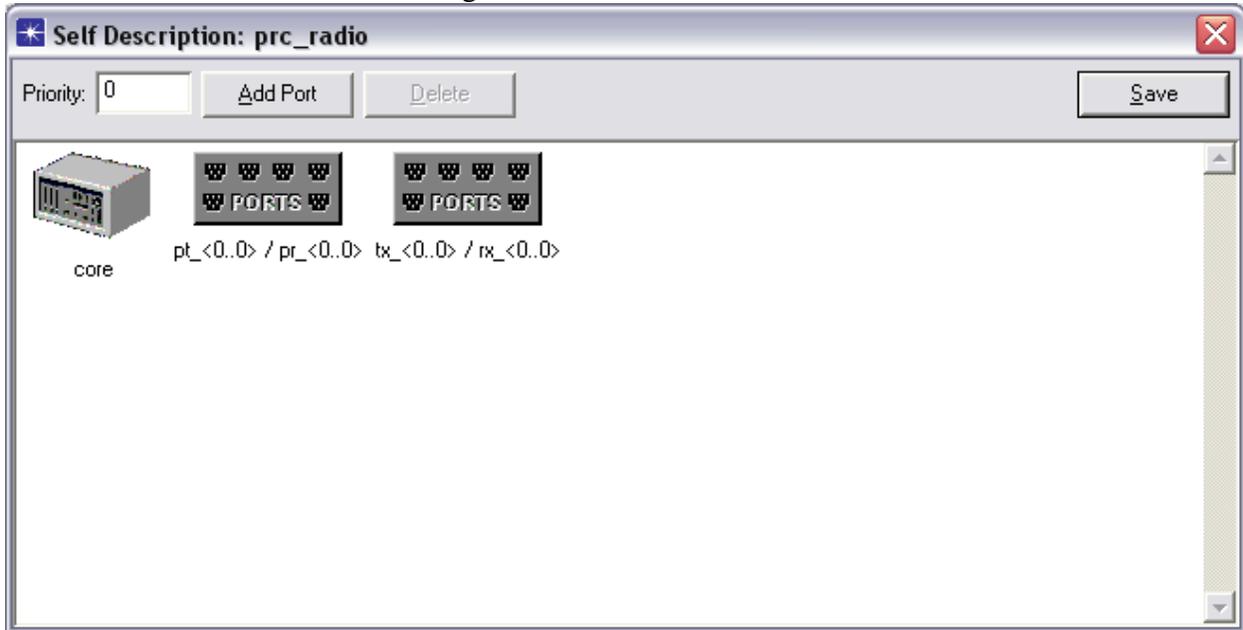


Figure 4-58: Equipment type attribute example

The interface class is defined within the ports description as shown in the following Figure 4-59. In this example, the interface class of the PRC device model is IP.

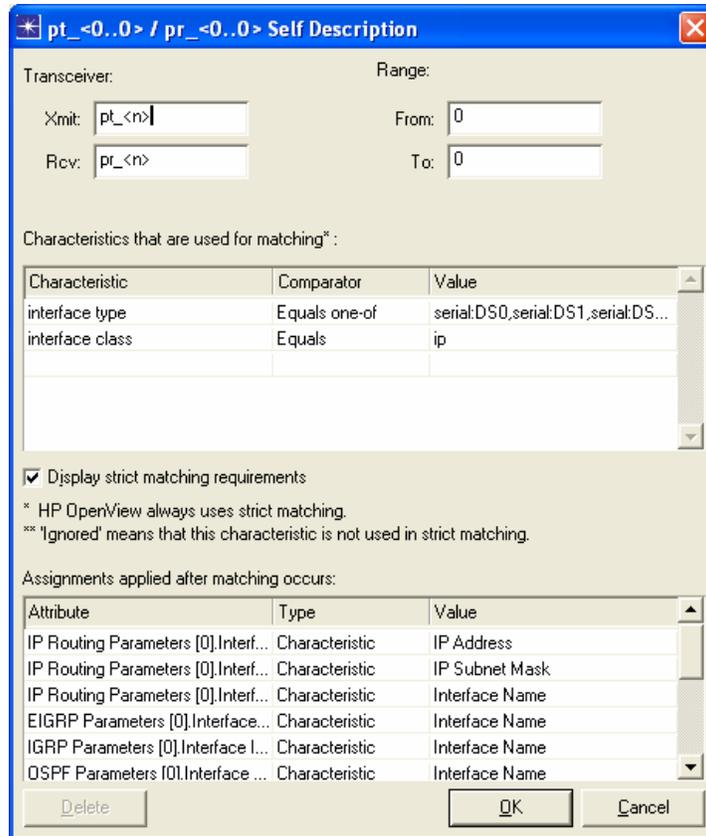


Figure 4-59: Equipment type attribute example

Finally, the Machine type attribute is defined within the core section of the self-description. The following Figure 4-60 shows the example of the machine type that is assigned to the PRC device model, and the value is router.

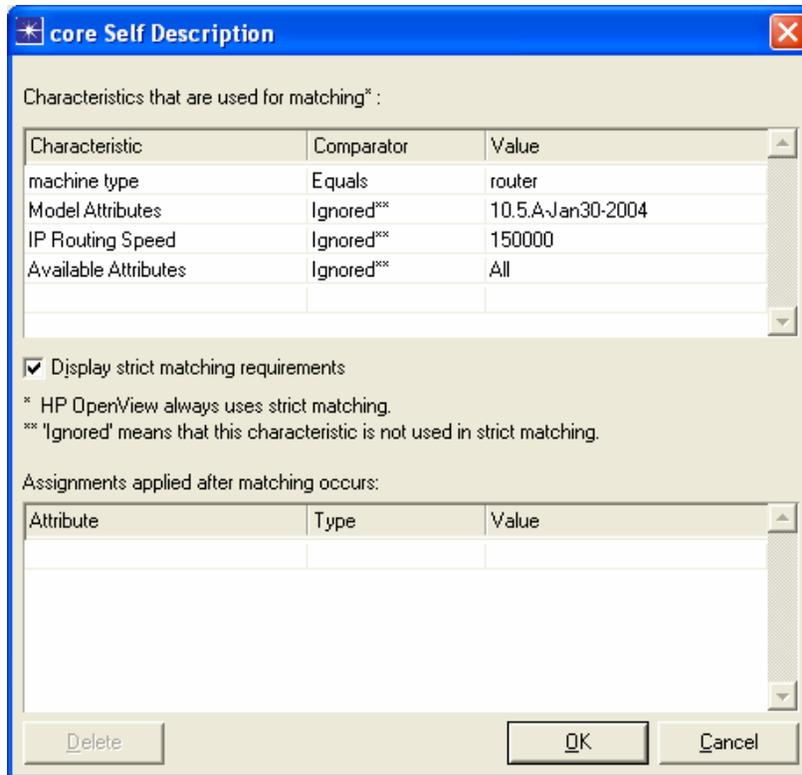


Figure 4-60: Equipment type attribute example

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 the credibility, 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 its VV&A Recommended Practices Guide (RPG), Build 3.1 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
www.dtic.mil/whs/directives/corres/pdf/500059p.pdf
- DoDI 5000.61 – DoD Modeling and Simulation (M&S) Verification, Validation, and Accreditation (VV&A) www.dtic.mil/whs/directives/corres/pdf/500061p.pdf
- VV&A Recommended Practices Guide – Build 3.1 / September 2006
<http://vva.msco.mil/>

The accreditation portion may or may not be significant for the JCSS 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 JCSS-compliant models within the JCSS 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 JCSS. 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 JCSS.

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:

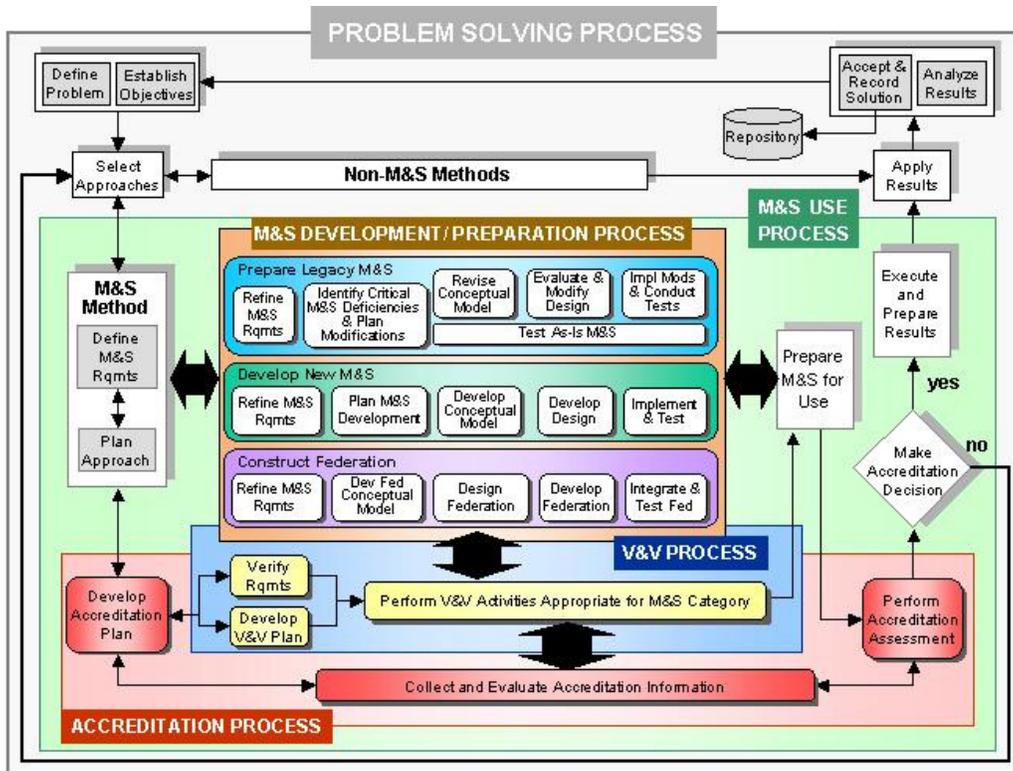
Verificati–n - The process of determining that a model implementation and its associated data accurately represent the develo’er’s conceptual description and specifications.

Validati–n - 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”.



The Overall Problem Solving Process

5/15/01

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 JCSS Compliance V&V are included in Appendix X: JCSS Model Development Guide Checklist. The steps that will be discussed in further detail in the next section are:

- Following the JCSS Model Development Guide Checklist
- Static Testing
- Equipment String
- Capacity Planner

An important reference regarding V&V is the “JCSS Communications Model Verification and Validation Plan.” This document defines the JCSS structured, repeatable process for ensuring that all communications device models included in JCSS 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 JCSS.

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 JCSS COMPLIANCE V&V

The primary objective of this JCSS MDG is to ensure that newly developed models can be integrated to JCSS and shared with the JCSS community. The JCSS 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 JCSS compliance V&V. These resources include the JCSS Model Development Checklist, JCSS Static Testing, JCSS Equipment String, and Capacity Planner Attributes.

5.2.1 JCSS Model Development Checklist

The JCSS Model Development Guide Checklist is the first tool to ensure that newly developed models can be integrated to the JCSS 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 JCSS 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 JCSS 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 JCSS standard palette 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 JCSS Static Testing

The JCSS Static Testing Tool comes with JCSS. 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 JCSS 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 “JCSS Communications Device Model Validation and Verification Plan” for further information.

5.2.3 JCSS Equipment String

In order to ensure that new models are JCSS-compliant, they should be tested using some basic equipment strings that are relevant to the model that was developed. JCSS Program Management Office (DISA GE344) has a “JCSS Equipment Strings document. This living document contains valid equipment strings that involve JCSS 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, prc and eplrs

Invalid Equipment Strings – illogical and unsupported

Another important reference is the “JCSS Equipment Strings Final Test Plan, OPNET 3.4.4. This document provides tests for JCSS 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 JCSS standard library.

5.2.4 Capacity Planner

The JCSS 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 JCSS-specific capacity, therefore models developed for OPNET Modeler and IT-Guru cannot be applied in CP. In order to ensure models are JCSS-compliant with regards to CP and routing, device attributes and properties should be correctly developed. They include:

- Equipment Type
- Interface Type
- Interface Class
- Machine Type
- Nodal mode

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 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 JCSS, 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

Table A-1: Acronyms

Acronym	Definition
ACE	Applications Characterization Environment
ACK	Acknowledgement
ADT	Application Delay Tracking
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
CM	Configuration Management
COI	Community of Interest
COTS	Commercial Off-the-Shelf
CP	Capacity Planner
CPU	Central Processing Unit
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
DTG	Digital Transmission Group
DVDT	DoD VV&A Documentation Tool
ECC	Error Correction Calculation
EIGRP	Extended Interior Gateway Routing Protocol
EPLRS	Enhanced Position Location Reporting System
ETSSP	Enhanced TSSP
FAQ	Frequently Asked Questions
FCC	Federal Communication Commission
FDDI	Fiber Distributed Data Interface
FDMA	Frequency Division Multiple Access
FLAN	Flow Analysis
FR	Frame Relay
FTP	File Transfer Protocol
GBS	Global Broadcast Service

Acronym	Definition
GOTS	Government Off-the-Shelf
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
ICI	Interface Control Information
IEEE	Institute of Electrical and Electronics Engineers
IER	Information Exchange Requirement
IGRP	Interior Gateway Routing Protocol
INC	Internet Controller
INE	Inline Network Encryptor
IP	Internet Protocol
ISDN	Integrated Services Digital Network
JCSS	Joint Communication Simulation System
JTIDS	Joint Tactical Information Distribution System
JVDT	Joint VV&V Documentation Tool
KP	Kernel Process
LAN	Local Area Network
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
MIL-STD	Military Standard
MOP	Measure of Performance
MPLS	Multiprotocol Label Switching
MSE	Mobile Subscriber Equipment
NACK	Negative Acknowledgment
NCES	Net-Centric Enterprise Service
OE	Operational Element
OLSR	Optimized Link State Routing
OMS	OPNET Model Support
OPFAC	Operational Facility
Org	Organization
OSI	Open Systems Interconnect
OSPF	Open Shortest Path Forwarding
OT	Output Table
OV	Output Vector
PPP	Point-to-Point Protocol
QAE	Quality Assurance Engineer
QoS	Quality of Service
RF	Radio Frequency
RIP	Routing Information Protocol
RP	Resource Planner
RPG	Recommended Practices Guide
RSVP	Resource Reservation Protocol

Acronym	Definition
SATCOM	Satellite Communications
SB	Scenario Builder
SE	System Element
SHF	Super High Frequency
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
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TIREM	Terrain Integrated Rough Earth Model
TORA	Temporally Oriented Routing Algorithm
TPAL	Transport Protocol Adaptation Layer
TSSP	Tactical Satellite Signal Processing
UDP	User Datagram Protocol
V&V	Validation and Verification
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

APPENDIX B: GLOSSARY

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_.

Model directories (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 .html or 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.

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. Some of the capabilities are creating libraries of OPFACs and organizations, importing from these libraries, and defining IERs.

Simulation domain: This consists of the Simulation Engine and the models.

Simulation Engine: There are two simulation engines in JCSS: Capacity Planner and DES.

APPENDIX C: ENUMERATED VALUES

The enumerated data types in Table C-1 are provided in JCSS 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 1 Radio Layer 1 Satellite Accelerator Generic Device VTC Terminal (IER only) Media Gateway (IER only)
Classification	Unclassified Classified Confidential Secret Top Secret Edit ... (for user defined – Nopde only)

Traffic Type (IER only)	Data Voice VTC
Protocol (IER only)	TCP UDP N/A (used for Voice and VTC IERs)
Priority (IER only)	ROUTINE PRIORITY IMMEDIATE FLASH FLASH OVERRIDE

APPENDIX D: PACKET FORMATS

Table D-1 lists the packet formats used by the JCSS Standard models. These packet formats may be required for interoperability with the JCSS Standard models and protocols.

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	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\misc\hla
absolute_move	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\misc\hla
ale_word_data	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\falcon
ale_word_lqa	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\falcon
ale_word_std	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\falcon
ckswpkt	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\circuit_switch_voice
data	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\nwstd
dummy_muxer_pk	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\circuits\promina
dummy_voice_pk	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\circuit_switch_voice
eplrs_eot_packet	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_hello	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_mac	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_mcast_ip	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_0	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_1	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_2	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_3	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_4	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_5	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_6	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_packet_7	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_pdu	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_routing_pk	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
eplrs_xmt_unit	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\eplrs
gen_sim_info	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\misc\hla
ier_description	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\misc\hla
IER_Fire	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\misc\hla
ier_info	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\misc\hla
ip_dgram_v4	JCSS\Sim_Domain\op_models\Jcss_std_models\modified_opnet_std_models\ip
JREAP_application_free_text_encoded	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\link16
JREAP_application_free_text_uncoded	JCSS\Sim_Domain\op_models\Jcss_std_models\netwars_std_models\radio\link16

JREAP_application_header	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
JREAP_application_J_series	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
JREAP_full_stack_message_group	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
JREAP_full_stack_transmission_block	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
JREAP_mgmt_message	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
JTIDS_packed_frame	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
jtids_pk	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\jtids
KG194_19	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\crypto
KG84_7	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\crypto
layer_1_circuit_data	JCSS\Sim_Domain\op_models\Jess_std_models\contributed_models\navy_spawar_models
Link_16_free_text_message	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
Link_16_J_series_message	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
link_info	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
link11_data	JCSS\Sim_Domain\op_models\Jess_std_models\contributed_models\navy_spawar_models
MIL_STD_1553_packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\link16
mop_data	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
mop_info	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
move_opfac_by	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
move_opfac_by_bearing	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
move_opfac_to	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
mse_data_packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuit_switch_voice
mse_hello_packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuit_switch_voice
new_ier_description	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
NIMA	JCSS\Sim_Domain\op_models\Jess_std_models\contributed_models\navy_spawar_models
nw_ip_voice_hello	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\nwstd
nw_rtp_pkt	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\nwstd
opfac_damage	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
opfac_init	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
opfac_repair	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
phone_switch	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuit_switch_voice
positional_move	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
pre_data_packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\pre
pro_cx_pk	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuits\promina
pro_hello_pk	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuits\promina
pro_wan_pk	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuits\promina
radio_packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\pre
satellite_pk	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\satellite\tssp
SRAP_application	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\satellite\dama
SRAP_application_v2	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\satellite\dama
tdm_data_pkt	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuits\circuit_emulation_devices
tdm_hello_pkt	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuits\circuit_emulation_devices
tdm_setup_pkt	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\circuits\circuit_emulation_devices
trigger_ier	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
trigger_new_ier	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
tssp_frame	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\satellite\tssp
UHF_SATCOM_Sat_Packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\satellite\dama
USMTF	JCSS\Sim_Domain\op_models\Jess_std_models\contributed_models\navy_spawar_models
vector_move	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\misc\hla
voice_packet	JCSS\Sim_Domain\op_models\Jess_std_models\netwars_std_models\radio\pre

APPENDIX E: STANDARD OPNET 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. The following table is not an exhaustive list.

Table E-1: Interfaces and Packet Formats

Interface Technology	Supported Packet Formats	Link Type
ATM	ams_atm_cell	ATM_generic ATM_OC3 ATM_OC12 ATM_SONET_OC48 ATM_SONET_OC24 ATM_SONET_OC12 ATM_SONET_OC3 ATM_SONET_OC1
Circuit Switch	ckswpkt phone_switch	wire_ptp phone_switch
Ethernet	ethernet_v2	1000BaseX 100BaseT 10BaseT
FDDI	fdi_mac_fr fdi_mac_tk	FDDI
Frame Relay	frms_admin_frame frms_frame_fmt frms_tpal_setup_frame	FR_link_generic FR_DS0 FR_E3 FR_E1 FR_T3 FR_T1
Multiplexer	ckswpkt ip_dgram_v4 KG194_19 KG84_7 layer_1_circuit_data mse_hello_packet tssp_frame	wire_ptp mux_aggregate T3 T1 PPP_E3 PPP_E1 PPP_SONET_OC12 PPP_SONET_OC3 PPP_SONET_OC1 PPP_DS3 PPP_DS1 PPP_DS0 KG194 KG95-2 KG75 KY57 KG94 KIV19 KG84 KIV7

Interface Technology	Supported Packet Formats	Link Type
Promina	ethernet_v2 ams_atm_cell ckswpkt ip_dgram_v4 KG194_19 KG84_7 layer_1_circuit_data mse_hello_packet tssp_frame pro_hello_pk pro_wan_pk	wire_ptp mux_aggregate T3 T1 PPP_E3 PPP_E1 PPP_SONET_OC12 PPP_SONET_OC3 PPP_SONET_OC1 PPP_DS3 PPP_DS1 PPP_DS0 ATM_generic ATM_OC3 ATM_OC12 ATM_SONET_OC48 ATM_SONET_OC24 ATM_SONET_OC12 ATM_SONET_OC3 ATM_SONET_OC1all 1000BaseX 100BaseT 10BaseT KG194 KG95-2 KG75 KY57 KG94 KIV19 KG84 KIV7 promina_wan_link
SLIP (DSL, ISDN)	ip_dgram_v4	wire_ptp T3 T1 PPP_E3 PPP_E1 PPP_SONET_OC12 PPP_SONET_OC3 PPP_SONET_OC1 PPP_DS3 PPP_DS1 PPP_DS0
Token Ring	tk_llc_fr tk_mac_fr tk_mac_tk	TR4 TR16

Interface Technology	Supported Packet Formats	Link Type
Wireless LAN	wlan_control, wlan_mac	N/A

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 JCSS 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.

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 JCSS folder of “nwstd.”

Select “ier_info.ic.m” file, to display a dialog box with the ICI format Attribute Names, Type, Default Value, and Description (if any).

Table F-1: Interfaces and Packet Formats

ICI Format	Location
bbs_atm_intf	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\circuit_emulation_devices
call_established	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
earth_tdm_bgutil	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\gbs
eplrs_graph	JCSS\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_hdr	JCSS\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_hdr_graph_update	JCSS\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_pipeline_ici	JCSS\Sim_Domain\op_models\netwars_std_models\radio\eplrs
eplrs_ptc_ici	JCSS\Sim_Domain\op_models\netwars_std_models\radio\eplrs
epuu_to_eplrs	JCSS\Sim_Domain\op_models\netwars_std_models\radio\eplrs
fail_rec	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
from_data_switch	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
fsr_initiate	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
gss_inport_ici	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\gbs
gss_packet_ici	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\gbs
ier_ack	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
ier_info	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
ier_pkt_info	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
inform_data_switch	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
inform_mux	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
JRE_mgr	JCSS\Sim_Domain\op_models\netwars_std_models\radio\link16
link_stat_ici	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
mse_ici	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
nw_gateway_call_end_ici	JCSS\Sim_Domain\op_models\netwars_std_models\voip\gateway
NW_HLA_ABSOLUTE	JCSS\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_IER_FIRE	JCSS\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_NEW_IER_FIRE	JCSS\Sim_Domain\op_models\netwars_std_models\misc\hla
NW_HLA_POSITIONAL	JCSS\Sim_Domain\op_models\netwars_std_models\misc\hla

NW_HLA_VECTOR	JCSS\Sim_Domain\op_models\netwars_std_models\misc\hla
nw_voice_ici	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
nw_voice_mgr	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
pro_perm_bgutil	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
release_bandwidth	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
reserve_bandwidth_failure	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
tdm_bgutil	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\circuit_emulation_devices
thread_info	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
tpal_req	JCSS\Sim_Domain\op_models\modified_opnet_std_models\tpal
tpal_se	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
UHF_SATCOM_DAMA_Info	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Entity_Config	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Entity_Registration	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Hello	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Terminal_Rev_Info	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM-Token_Passing	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama

APPENDIX G: MODELING FILE FORMATS

The *typed file* attribute is used to specify file names for intrinsic file types recognized inside the OPNET environment. Table G-1 lists typed file functions.

Table G-1: 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.obj	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
.ets.c	External Tool Support C code file
.ets.cpp	External Tool Support CPP code file

File Suffix	File Function
.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 JCSS

Table I-1 lists the measures of performance reported by OE in a JCSS 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

Statistic Name	Method of Calculation	Scope
Completion Rate	Shows the percentage of completed IERs vs. failed IERs. This statistic specifies the probability that an IER is completed on the network.	Local Global
Connection Latency (seconds)	This statistic specifies the average amount of time that an IER requires to initially setup in the network.	Local Global
End to End Delay (seconds)	Shows the total end-to-end delay of the IER. This statistic specifies the average amount of delay for an IER to reach the destination in the network. This is based off the sent time of the IER (i.e., when the IER was sent to the device).	Local Global
Failed Count	Specifies the number of IERs that failed during the simulation.	Local Global
Grade of Service	This statistic specifies the percentage that an IER was able to be sent onto the network.	Local Global
Perished Count	Specifies the number of IERs that perished during the simulation. Perished IERs mean the IER was not received by the time specified in the IER definition.	Local Global
Received Count	Specifies the number of IERs that completed successfully during the simulation.	Local Global
Retry Attempts	Specifies the number of retry attempts needed to send the IER on the network (i.e., it was not blocked). A blocked IER means that no device was available to accept the particular IER traffic at the start time and it must be retried at a later time.	Local Global
Sent Count	Specifies the number of IERs sent during the simulation.	Local Global
Speed of Service (seconds)	This statistic specifies the average amount of delay for an IER to reach the destination in the network. This is based off the start time of the IER (i.e., when the IER signaling was started on the device).	Local 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

Statistics Groups	Scope
Individual IERs	Local
Total Data IERs	Global
Total Flash IERs	Global
Total Flash Override IERs	Global
Total Immediate IERs	Global
Total Priority IERs	Global
Total Routing IERs	Global
Total Voice IERs	Global
Total VTC IERs	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.

DES OT Reports

The first type of report is the Discrete Event Simulation OT Reports. These reports are utilized by many of the JCSS device models but have been expanded to include IER information. The information seen in these reports would include producer and consumer device names, actual start and stop times, whether the IER was successful or failed, what the reason for the failure was, and who failed the IER, among other things.

1. The user needs to enable the IER reporting capabilities. This functionality can be enabled either globally through the Discrete Event Simulation or per-IER by using the Edit Attributes dialog box on a specific IER definition.
 - a. To configure IER reporting globally for DES, navigate to the **DES > Configure/Run Discrete Event Simulation...** menu. Set the **IER Reports** attribute in the **Global attributes** tab to *Export Reports*. By default, this attribute is set to *Do Not Export Reports* for simulation efficiency reasons.

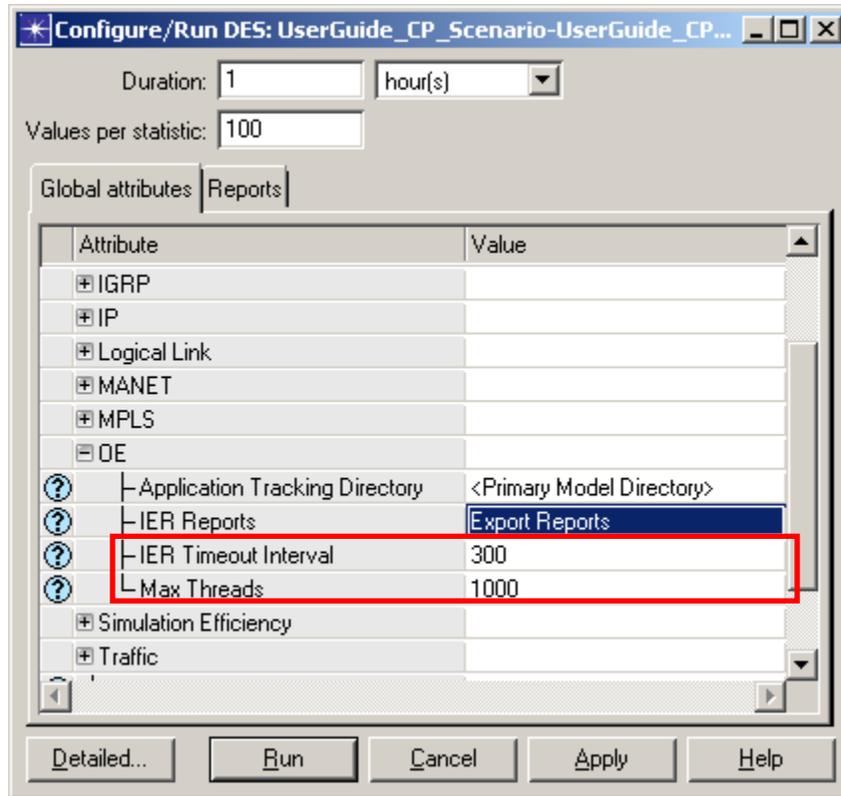


Figure I-1: Enabling IER DES Reporting Capabilities Globally

- b. To configure IER reporting per-IER, right click on an IER Traffic Flow object and select the **Edit Attributes** menu. Enable the **Export Reports** checkbox and select **OK** to save the changes.

The screenshot shows the 'IER' dialog box with the following configuration:

- Name: computer --> computer
- ID: USER105
- Type: DATA*
- Equipment: Computer*
- Protocol: TCP
- Classification: Unclassified
- Priority: ROUTINE
- Perishability (s): 15
- Message: Not Configured

Generation Parameters:

Size (s for VOICE/WTC, bytes for other ...)	constant(100000)
Interarrival (s)	exponential(30)
Start Time	100
Stop Time	END

Application Delay Tracking
 Export Reports
 Validate IER Values

Buttons: OK, Cancel

Figure I-2: Enabling IER DES Reporting Capabilities Per-IER

2. The user must run a Discrete Event Simulation to completion on a particular IER scenario. This can be accomplished using the **DES > Configure/Run Discrete Event Simulation...** menu.
3. Once a simulation is finished, users can either right-click on the scenario and select **View Results** or navigate to the **DES > Results > View DES Reports** menu. This will launch the **Results Browser** dialog box which shows the normal DES statistics, as well as, the additional IER reports. To navigate to the reports, select the **DES Run (1) Tables** tab in the **Results Browser** dialog box. All IER reports will be located under the **IERs** section in the treeview for this tab.

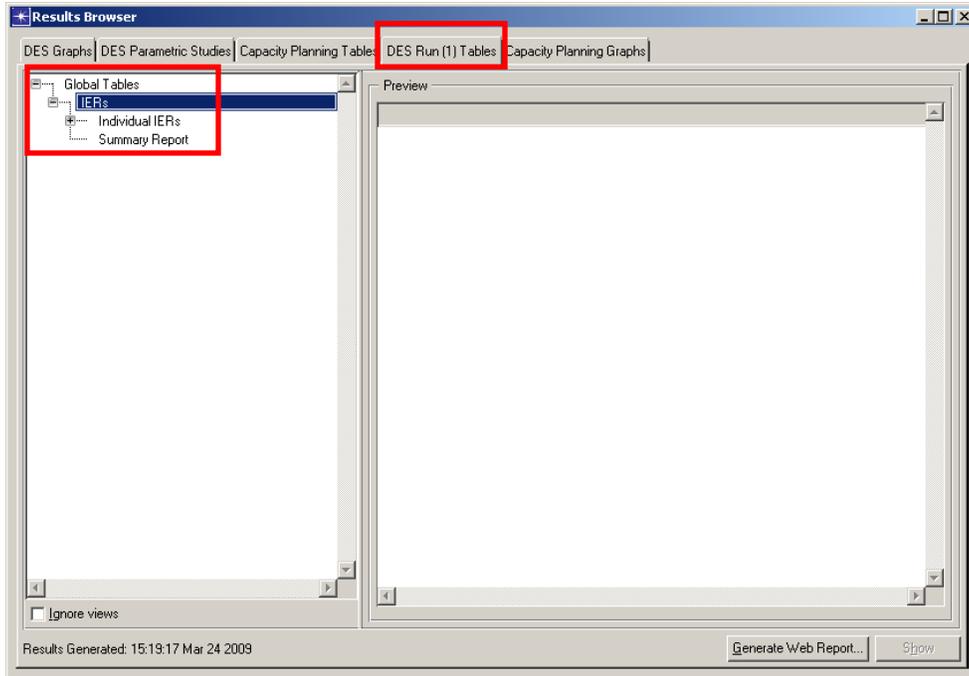


Figure I-3: Viewing the IER Reports

4. Select an IER report inside the treeview. Once the report is selected, a preview will be shown with a portion of the information found in the report. Click the **Show** button to view the entire report. Note that when viewing these reports, hyperlinks and export capabilities allow the user to easily understand and translate the report information.

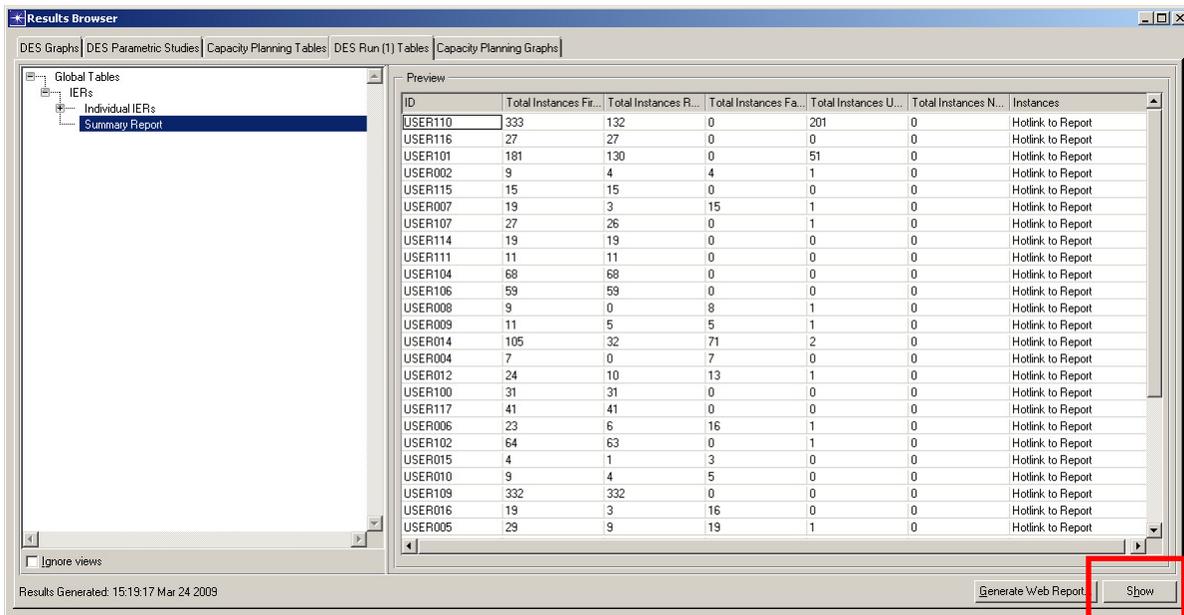


Figure I-4: Selecting an IER Report

There are currently several DES Reports available to the user. The **Summary Report** allows the user to see the overall IER information per IER (number of instances fired,

number of instances received, etc.). The **Individual IERs Report** allows the user to see the information per IER on two levels: **Details** and **Instances**. The **Details** section shows the standard IER information (type, Interarrival, size, producer device, consumer device, etc.). The **Instances** section shows each time the IER specified in the Details section was fired in the scenario (i.e., an IER instance). Included with each instance is whether the traffic was successful, a setup time, status (failed or successful), and the reason for failure (if any), among other things.

Also, similar reports are given for Threaded IERs in the scenario.

Application Delay Tracking

Application Delay Tracking (ADT) is a standard OPNET feature that was integrated into the IER traffic. ADT helps identify the sources of application delay in DES. With ADT, a user can follow each packet of an application message throughout the simulated network. For more information, on this feature, refer to the OPNET Standard documentation. To enable this tracking with IERs, use the following steps:

1. The user needs to enable the ADT reporting capabilities. This functionality can only be enabled per-IER by using the Edit Attributes dialog box on a specific IER definition. To configure IER reporting per-IER, right click on an IER Traffic Flow object and select the **Edit Attributes** menu. Enable the **Application Delay Tracking** checkbox and select **OK** to save the changes.

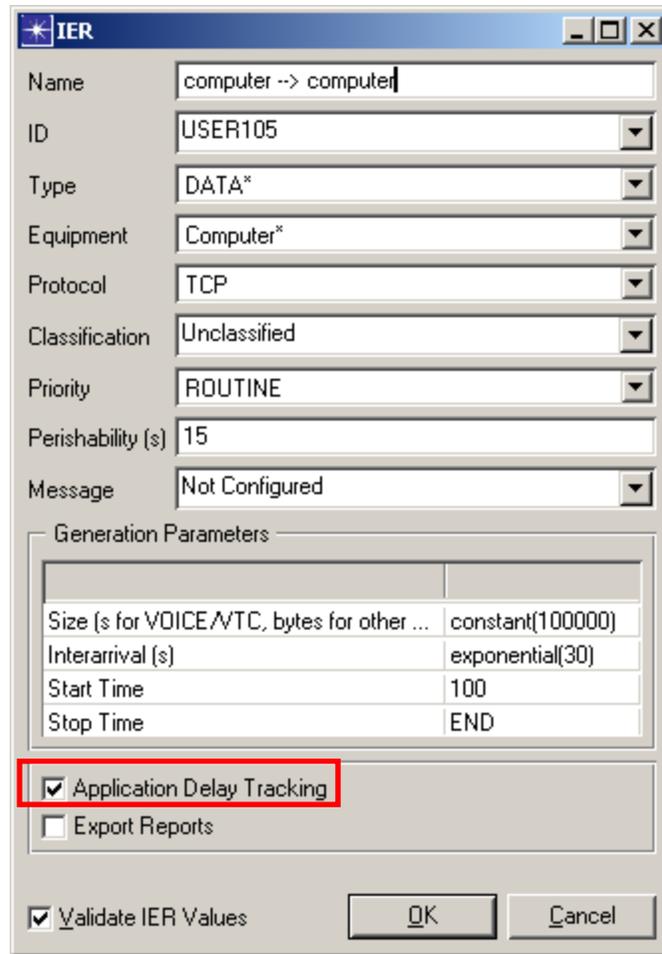


Figure I-5: Enabling Application Delay Tracking Per-IER

2. Next, the user can specify the directory of the output ADT file. To do this, navigate to the **DES > Configure/Run Discrete Event Simulation...** menu. Set the **Application Tracking Directory** attribute in the **Global attributes** tab to any valid Windows directory. By default, this attribute is set to *<Primary Model Directory>* which means the file will be output to the first directory set inside your *Model Directories* preference (see the **Edit > Preferences > Advanced** menu).
3. The user must run a Discrete Event Simulation to completion on a particular IER scenario. This can be accomplished using the **DES > Configure/Run Discrete Event Simulation...** menu.
4. Once a simulation is finished, users can navigate to the **DES > Results > View Application Delay Tracking** menu for further review of the data. The user will then specify the location of the ADT file (which will be located in the set directory above) and will be named according to the following format: *<project name>-<scenario name>.adt*. This will launch the **Application Segment Tracking Viewer** dialog box which shows the collected ADT information from the simulation.

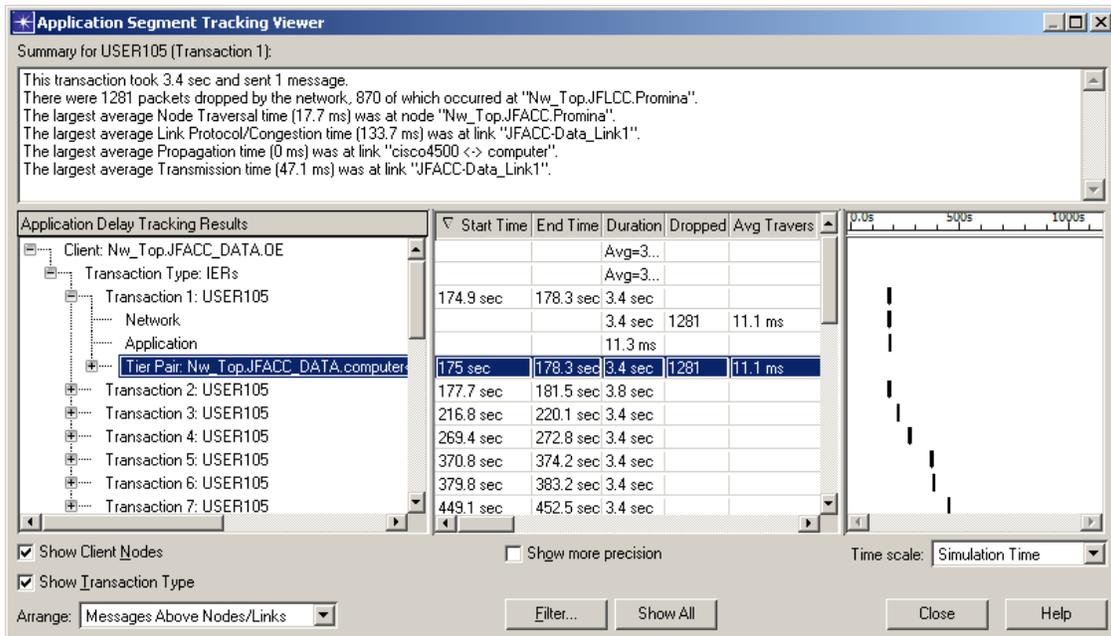


Figure I-6: Viewing Application Delay Tracking Files

Device-Level MOPs

The ability to collect device-level MOPs in JCSS allows the model developer and user to collect any OPNET node-level statistic in a JCSS simulation. Any statistic promoted to the node level will appear when the user chooses statistics in the Scenario Builder editor. 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 JCSS:

- Queuing Delay (recorded in both forward and reverse directions separately for wireline links)
- Throughput (recorded in both forward and reverse directions separately for wireline links)
- Utilization (recorded in both forward and reverse directions separately for wireline links)
- Channel Utilization (only for links sending Voice traffic such as Circuit Switch and Promina)

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. Shown below is a basic template of information that should be included:

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

Below describes some of the above in more detail:

- 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
trc_170	trc170	This is the line of sight radio transmission device.

- 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
ip_dispatch	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 JCSS 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 JCSS 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: *JCSS_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: *JCSS_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.

APPENDIX L: JCSS SIMULATION API AND HELPER FUNCTIONS

APIs are 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 JCSS model files looking for those files that end in “.ex.c*”, which would include both C and C++ APIs.

The following table lists the External Files along with the features supported by each file.

Table L-1: JCSS APIs and their Locations

Support Area	API List	Location
BGP Models	bgp_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\bgp
Encryptor Models	crypto_support	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
EPLRS Model	eplrs_support	JCSS\Sim_Domain\op_models\netwars_std_models\radio\epIrs
Circuit Switch Models	flood_search_routing	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
HAiPE Models	gdc_notif_log_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\gdc
	gdc_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\gdc
Standard Application Models	gna_sup_conn_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\applications
	gna_sup_lib	JCSS\Sim_Domain\op_models\modified_opnet_std_models\applications
IP Auto-Addressing	ip_auto_addr_sup_v4	JCSS\Sim_Domain\op_models\modified_opnet_std_models\ip
IP Modeling	ip_rte_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\ip
Wireless Configuration Node Support	jcss_wireless_config_support	JCSS\Sim_Domain\op_models\netwars_std_models\misc\utility
Link16 Modeling	Link_16	JCSS\Sim_Domain\op_models\netwars_std_models\radio\link16
Circuit Switch Voice and Channelization Modeling	netwars_logical_link_support	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
JCSS Satellite (TSSP, Bentpipe) Models	netwars_satellite_support	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\gbs
General JCSS Modeling Support	netwars_support	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
Accelerator 4000 Model	nw_accelerator_sup	JCSS\Sim_Domain\op_models\netwars_std_models\router
Generic Circuit APIs	nw_circuit_api	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
	nw_circuit_stat	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
IP Auto-Addressing	nw_custom_ip_auto_addr	JCSS\Sim_Domain\op_models\netwars_std_models\misc\cots_support
IER Modeling	nw_ier_ot_support	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
	nw_ier_support	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
IP Auto-Addressing	nw_ip_modification_support	JCSS\Sim_Domain\op_models\netwars_std_models\misc\cots_support
Tracer Packet APIs	nw_oms_basetraf_src	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
	nw_tracer_pkt_support	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
VoIP Phone	nw_voice_gateway	JCSS\Sim_Domain\op_models\netwars_std_models\voip\gateway

and Gateway Models	nw_voice_mgr	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
Tracer Packet APIs	oms_buffer_bgutil	JCSS\Sim_Domain\op_models\modified_opnet_std_models\oms
PEP Model	pep_app_support	JCSS\Sim_Domain\op_models\netwars_std_models\pep
Promina Models	promina_rte	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
	promina_support	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
	promina_support_alt	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
	promina_topo	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
	promina_voice_support	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
RTP Models	rtp_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\rtsp
TCP Models	tcp_api	JCSS\Sim_Domain\op_models\modified_opnet_std_models\tcp
TIREM Propagation Model	tirem_support	JCSS\Sim_Domain\op_models\netwars_std_models\misc\tirem
TPAL Model (Interface to UDP/TCP)	tpal_api	JCSS\Sim_Domain\op_models\modified_opnet_std_models\tpal
	tpal_app_support	JCSS\Sim_Domain\op_models\modified_opnet_std_models\applications
DES Debugger Trace Messaging Support	trace_support	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
UHF DAMA Models	UHF_SATCOM_CPS_Entity	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
	UHF_SATCOM_CPS_ServicePlan_Parser	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
	UHF_SATCOM_CPS_TextManipulation	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
	UHF_SATCOM_Noise_Area	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
	UHF_SATCOM_Orderwires	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
	UHF_SATCOM_Platform	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
	UHF_SATCOM_Platform_Utilization	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Port_Map	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama	
USN Circuit Modeling	USN_ckt_supp	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models

APPENDIX M: ATTRIBUTE TYPE DEFINITIONS

This appendix describes the various attribute types used in JCSS. 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 JCSS 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 JCSS MODELS

This appendix will go through an example of locating a JCSS Model and the information relevant to the model. Table N-1 provides a list of all the JCSS Models in alphabetic order to use as a reference in locating a specific model.

The following table is a complete list of the JCSS Models in alphabetical order.

Table N-1: List of JCSS Models (Alphabetic)

JCSS Models	Location
Accelerator4000	JCSS\Sim_Domain\op_models\netwars_std_models\router
Alcatel7270_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
Alcatel7470_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Alcatel7750SR1_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Alcatel7750SR12_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Alcatel7750SR7_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
ale_config	JCSS\Sim_Domain\op_models\netwars_std_models\radio\facon
AN_FCC_100_V	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_131_V_BB_Transmitter	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_131_V_NB_Transmitter	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_131_V_Receiver	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_URC_139_V	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_USC_38_MDR	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V11	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V14	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V15	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V17	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V18	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V2	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V3	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V6	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V7	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_3_V9	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_5_V	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V2	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V4	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V5	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V7	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V9_C	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_6_V9_X	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_8_V1	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
AN_WSC_8_V2	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
CA_Satellite	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
CB_SS_2200	JCSS\Sim_Domain\op_models\netwars_std_models\router
CB_SS_6000	JCSS\Sim_Domain\op_models\netwars_std_models\router
CB_SS_9000_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router

CellXpress_PVC_Config	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
CISCO 2505	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2507	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2509	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2511	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2512	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2516	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2524	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2621	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2916	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2924	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2950G 24 EI	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 2950G 24 EI_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3000	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3620	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3640	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3660	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3725_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3745	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 3745_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Cisco 3750_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 4006	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 4500-M	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 4700-M	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 7010	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 7206	JCSS\Sim_Domain\op_models\netwars_std_models\router
CISCO 7507	JCSS\Sim_Domain\op_models\netwars_std_models\router
Cisco_LS_1010	JCSS\Sim_Domain\op_models\netwars_std_models\switches
cisco2514_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
cisco4500_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
cisco7505_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Cisco7513_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
cs_end_device_base	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
CTP_1012	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\circuit_emulation_devices
CTP_2024	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\circuit_emulation_devices
Definity Prologic	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
dnvt	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
DPA	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
DPM	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
DSCS_III_Satellite	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
DSS-1	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
DSS-2	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
DSS-3	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
DTA	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
EPLRS_ENM	JCSS\Sim_Domain\op_models\netwars_std_models\radio\ephrs
EPLRS_RS	JCSS\Sim_Domain\op_models\netwars_std_models\radio\ephrs
EPLRS_Simple	JCSS\Sim_Domain\op_models\netwars_std_models\radio\ephrs
ethernet_gdc_server_bgp_adv	JCSS\Sim_Domain\op_models\modified_opnet_std_models\gdc
ethernet4_slip8_gdc_gtwy_adv	JCSS\Sim_Domain\op_models\modified_opnet_std_models\gdc
Falcon_II_ale	JCSS\Sim_Domain\op_models\netwars_std_models\radio\falcon
Firewall_2NIC	JCSS\Sim_Domain\op_models\netwars_std_models\router
Firewall_3NIC	JCSS\Sim_Domain\op_models\netwars_std_models\router
Firewall_4Slot	JCSS\Sim_Domain\op_models\netwars_std_models\router
FLBCST	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
FoundryFastIron1500Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron2402Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron400Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron4802Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron800Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryFastIron9604Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryNetIron1500Router_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
FoundryNetIron400Router_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models

FoundryNetIron800Router_adv	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
Generic ATM Switch	JCSS\Sim_Domain\op_models\netwars_std_models\router
Generic Ckt Switch	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
Generic Hub	JCSS\Sim_Domain\op_models\netwars_std_models\switches
Generic IDS	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
Generic IP Data Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Generic Layer 2 Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\switches
Generic Layer 3 Switch_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Generic MW LOS	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
Generic Router	JCSS\Sim_Domain\op_models\netwars_std_models\router
Generic Server	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
Generic Telephone	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
Generic UFO	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
generic_broadcast_satellite	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\gbs
Harris_6010_adv	JCSS\Sim_Domain\op_models\netwars_std_models\radio\facon
Harris_Megastar_155	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
IDNX-20	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
IDNX-90	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
IER_Firing_Rules_Config	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
ier_loader	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
INMARSAT_B_HSD	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
INMARSAT_B_Satellite	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
IP_ATM_TACLANE	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
JRE_Gateway	JCSS\Sim_Domain\op_models\netwars_std_models\radio\link16
jtids	JCSS\Sim_Domain\op_models\netwars_std_models\radio\jtids
JTIDS_Terminal	JCSS\Sim_Domain\op_models\netwars_std_models\radio\link16
KG_194	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
KG_84	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
KG-175 ATM	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG-175 IP	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG175-E_10	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG175-E100	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG194_crypto_base	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG-235	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG-250	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KG84_crypto_base	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KIV7_crypto_base	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
KY68	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
LAN WAN IP network_adv	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
layer_1_crypto_base	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
len	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
Link_11	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
Link_16_Config	JCSS\Sim_Domain\op_models\netwars_std_models\radio\link16
Link_16_Host_Processor	JCSS\Sim_Domain\op_models\netwars_std_models\radio\link16
Marconi_ASX1000	JCSS\Sim_Domain\op_models\netwars_std_models\router
Marconi_ASX1200	JCSS\Sim_Domain\op_models\netwars_std_models\router
Marconi_ASX200BX	JCSS\Sim_Domain\op_models\netwars_std_models\router
Marconi_PH6000_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Marconi_PH7000_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Marconi_PH8000_adv	JCSS\Sim_Domain\op_models\netwars_std_models\router
Marconi_TNX1100	JCSS\Sim_Domain\op_models\netwars_std_models\router
Media_Gateway	JCSS\Sim_Domain\op_models\netwars_std_models\voip\gateway
MilStar_2_Satellite	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
MMT	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
Motorola NES	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
MRC-142	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
mux_12inputs	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\mux
mux_16inputs	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\mux
mux_2inputs	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\mux
mux_4inputs	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\mux
mux_8inputs	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\mux
NAVMACS	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models

ncs	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
NES	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
NIMA	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
nw_eth_switched_lan_adv	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
nw_ethernet_server	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
nw_ethernet_wkstn	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
nw_generic_device	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
nw_hla_interaction	JCSS\Sim_Domain\op_models\netwars_std_models\misc\hla
nw_jam_pulsed	JCSS\Sim_Domain\op_models\netwars_std_models\radio\jammers
nw_jam_sb	JCSS\Sim_Domain\op_models\netwars_std_models\radio\jammers
nw_jam_swept	JCSS\Sim_Domain\op_models\netwars_std_models\radio\jammers
nw_mitre_manet_router_no_mcast	JCSS\Sim_Domain\op_models\contributed_models\jcas_contributed_models
nw_modeler_support	JCSS\Sim_Domain\op_models\netwars_std_models\misc\utility
nw_multihomed_server	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
nw_multihomed_wkstn	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
nw_ppp_server	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
nw_ppp_wkstn	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
Nw_QoS_Attribute_Config	JCSS\Sim_Domain\op_models\netwars_std_models\misc\utility
nw_radio_wkstn	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
Nw_Sink	JCSS\Sim_Domain\op_models\netwars_std_models\misc\utility
oe	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
Omni Switch 3WX	JCSS\Sim_Domain\op_models\netwars_std_models\switches
Omni Switch 5WX	JCSS\Sim_Domain\op_models\netwars_std_models\switches
Omni Switch 9WX	JCSS\Sim_Domain\op_models\netwars_std_models\switches
Patch_Panel_48	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
Patch_Panel_96	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
prc_inc	JCSS\Sim_Domain\op_models\netwars_std_models\radio\prc
prc_radio	JCSS\Sim_Domain\op_models\netwars_std_models\radio\prc
pro_cell_express_adv	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
pro_portmap_utility	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina_10l_5w_2eth_2sclx_2cx_adv	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina-100	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina-200	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina-400	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina400_e180_sl180	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Promina-800	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\promina
Proteon CNX 500	JCSS\Sim_Domain\op_models\netwars_std_models\router
Proteon CNX 600	JCSS\Sim_Domain\op_models\netwars_std_models\router
REDCOM HDX	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 1 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 10 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 16 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 2 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 3 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 4 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 5 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 6 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 7 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
REDCOM IGX 8 Shelf	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
RedEagle_INE-100	JCSS\Sim_Domain\op_models\netwars_std_models\crypto
sat_term_etssp_nodal	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
sat_term_etssp_non_nodal	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
sat_term_etsspG3_nodal	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
sat_term_etsspG3_non_nodal	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
sat_term_generic_lport	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\generic
sat_term_generic_8port	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\generic
sat_term_tssp_nodal	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
sat_term_tssp_non_nodal	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
satellite_generic	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\generic
SB-3865 1 Stack	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
SB-3865 2 Stack	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice

SB-3865 3 Stack	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
SCREAM_100	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\circuit_emulation_devices
SCREAM_50	JCSS\Sim_Domain\op_models\netwars_std_models\circuits\circuit_emulation_devices
sen	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
Server_4Slot	JCSS\Sim_Domain\op_models\netwars_std_models\data_applications
SHOUTip	JCSS\Sim_Domain\op_models\netwars_std_models\voip\gateway
SMU	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
SRC-57	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
STU-III	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
TACINTEL	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
TCDL_Radio	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
tcp_pep_adv	JCSS\Sim_Domain\op_models\netwars_std_models\pep
TD1271	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
Thales_SONET_Datacryptor	JCSS\Sim_Domain\op_models\netwars_std_models\usfk_models
Timeplex_CX-1500	JCSS\Sim_Domain\op_models\netwars_std_models\switches
Timeplex_Link_100	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
Timeplex_Link_2	JCSS\Sim_Domain\op_models\contributed_models\navy_spawar_models
TRC-170 V2	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-170 V3	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-170 V5	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
trc-170	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
TRC-173B	JCSS\Sim_Domain\op_models\netwars_std_models\radio\trc170
TSC-100A	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-152 w ETSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-152 w TSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-152 wo TSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-154	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-85B	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-85C w ETSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-85C	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-93B	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-93C w ETSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSC-93C	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
tsc-94	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\gbs
TSC-94A	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
TSQ-190	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
ttc-39	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-39A V3	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-39A V4	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-39D	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-39E	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-42	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-46 LEN	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-48 SEN	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
TTC-56	JCSS\Sim_Domain\op_models\netwars_std_models\circuit_switch_voice
UHF_SATCOM_CPS	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_NCS_Platform	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_PSC_Radio	JCSS\Sim_Domain\op_models\netwars_std_models\radio\psc
UHF_SATCOM_Satellite_FLTSATCOM	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Satellite_UFO	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_SRAP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
UHF_SATCOM_Terminal_Platform	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\dama
USC-59 w ETSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
USC-59 w TSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
USC-59 wo TSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
USC-60A w ETSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
USC-60A w TSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
USC-60A wo TSSP	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
voice_config	JCSS\Sim_Domain\op_models\netwars_std_models\nwstd
VoIP_Phone	JCSS\Sim_Domain\op_models\netwars_std_models\voip
voip_phone_mod2_hellos_enabled	JCSS\Sim_Domain\op_models\contributed_models\jcas_contributed_models
Wireless_Configuration	JCSS\Sim_Domain\op_models\netwars_std_models\misc\utility

WSC-6 V5	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
WSC-6 V6	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
WSC-6 V7	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
WSC-6 V9 C Band	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
WSC-6 V9 X Band	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp
WSC-8	JCSS\Sim_Domain\op_models\netwars_std_models\satellite\tssp

APPENDIX O: JCSS DOCUMENTATION SET

Figure O-1 illustrates a JCSS documentation set.

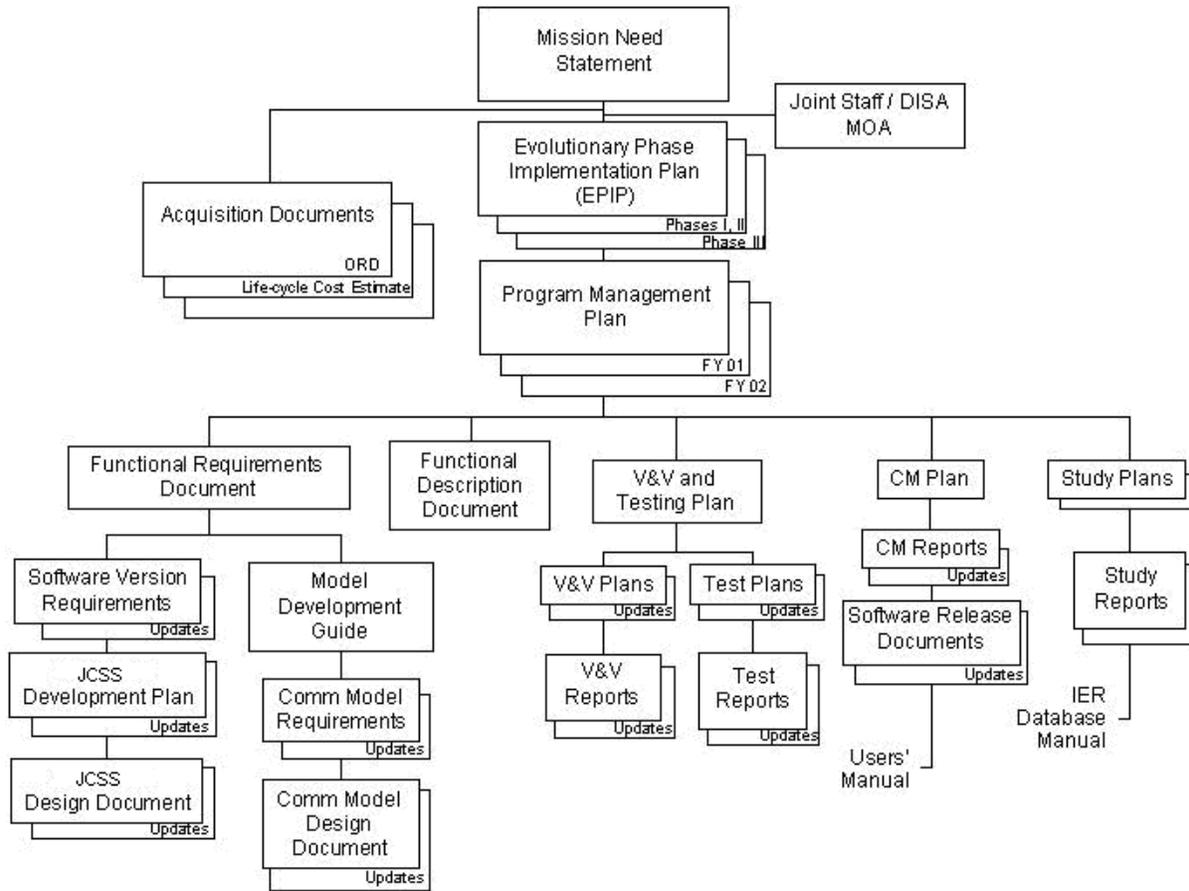


Figure O-1: JCSS Documentation Set

APPENDIX P: CREATING MODEL REPOSITORIES IN JCSS

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 JCSS 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 JCSS 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 *repository*. 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 15.0/OPNET Console), type the following command:

For building a development repository:

```
op_mkso -env_db "<drive_letter>\JCSS..\Sim_Domain\op_admin\env_db15.0"
-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 an optimized repository:

```
op_mkso -env_db "<drive_letter>\JCSS..\Sim_Domain\op_admin\env_db15.0"
-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>\JCSS..\Sim_Domain\op_admin*" folder)

Put this environment variable in the *env_db* file (located in "*<drive_letter>\JCSS..\Sim_Domain\op_admin*" folder)

```
repositories      : NAME_OF_REPOSITORY
```

APPENDIX Q: TROUBLESHOOTING JCSS SIMULATION

Refer to the standard online OPNET documentation for troubleshooting a Discrete Event Simulation.

APPENDIX R: FREQUENTLY ASKED QUESTIONS

Table R-1 outlines the solution to several frequently asked questions (FAQ).

Table R-1: FAQs

Question/Problem	Solution
<p>How should I set the mod_dirs environment variable for the various env_db files for custom model development?</p>	<p>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 JCSS, is located under the Scenario_Builder\op_admin folder of the JCSS 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 JCSS environment the user needs to include this mod_dirs entry in the JCSS env_db files. Please note that if the custom models are modified JCSS or OPNET Standard models, they must be placed before the JCSS and OPNET Standard model directories in both the env_db files.</p>
<p>How do I enable the debug mode in my simulation? How can I enable OPNET debugger in my JCSS simulation?</p>	<p>To enable the OPNET debugger (odb) for JCSS simulation, check the "Use OPNET Simulation Debugger" checkbox under Execution OPNET Debugger in the Configure/Run dialog box before running the simulation.</p>
<p>I want to specify simulation attributes. Where can I do that?</p>	<p>This can be done in the Configure/Run dialog box before running the simulation. The simulation attribute can be defined under Inputs Global Attributes.</p>
<p>I want to see the routing tables generated during the simulation. How can I do that?</p>	<p>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.</p>
<p>I have the TIREM data files on my system but still the TIREM is not enabled. Why?</p>	<p>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.</p>
<p>All my data IERs are failed or reported undelivered. What could be the reason?</p>	<p>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.</p>
<p>I have large data IERs (greater than 20 Mb) and none of the IERs go through? Why?</p>	<p>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</p>

Question/Problem	Solution
	the case of the application layer in JCSS, the size limitation of the transport layer is handled by segmenting the application layer packets.
What are Undelivered IERs? How are they calculated?	The IERs are reported as Undelivered when they do not make it to the destination before the simulation completes. Undelivered IERs = IERs Sent – (IERs Received + IERs failed)
I see some of the IERs reported as Undelivered. Where did they go?	The IERs are reported as Undelivered 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 undelivered. 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: FUNCTIONAL ENHANCEMENTS FROM EARLIER JCSS VERSIONS

Changes Made from JCSS Version 8.0 to JCSS Version 9.0:

- JCSS v9.0 is based on the latest OPNET CORE 15.0 PL1.
- EPLRS Model Enhancement:
 - As a continued effort to improve upon the current JCSS EPLRS model, further enhancements were made to the EPLRS model including adding Dynamic Routing for OSPF and Antenna characteristics. Please refer to the EPLRS Model User Guide for further assistance.
- DoDAF Import/Export:
 - The Department of Defense (DoD) Architecture Framework (DoDAF) provides a standard for description, development, presentation, and integration of systems for the DoD. DoDAF is a large standard consisting of many products some of which will be integrated into JCSS in future releases.
 - JCSS v9.0 provides supports for DoDAF OV-3 and SV-6 products as they closely resemble the JCSS Information Exchange Requirements (IERs) workflow. The interface consists of the ability to take already existing DoDAF OV-3 and SV-6 products and merge them into a JCSS scenario in the form of IERs. The user will be able to import the products, test the configuration using simulations, modify the configuration, and export the modified information to a valid OV-3 or SV-6

product. The advantage of providing this capability allows the user to quickly fill-in missing information and test the validity of the configuration. In the end, this allows the user to have more complete DoDAF products which will allow the military to make better comparisons and decisions. Please refer to the JCSS IER Model Userguide for additional information.

- IER Model Enhancements:
 - A major update was made to the JCSS IERs in v9.0. JCSS now makes use of demand objects to represent IERs and Threads in Scenario Builder as an alternative to the custom data structure that was previously used. The new IERs allows for better visualization, editing and offers several new capabilities including the use of various distribution attributes.
 - With the implementation of IER demands, the user will be able to create IERs by deploying a demand from the Object Palette or through a GUI interface. All IERs created by users in previous software releases will be automatically converted to the new format. For further information please see the IER Model Userguide.
- VNE SERVER Import (VNESI):
 - JCSS v9.0 now offers support for VNE server. VNESI is an OPNET COTS product that allows you to create a network model from information from VNE Server data. Please note that JCSS imposes an OPFAC and Organization hierarchy on all devices. Devices must live within an OPFAC in a JCSS environment. A VNE license is required to operation this capability and Users must contact OPNET Technologies.
- NetMapper Support:
 - JCSS v9.0 offers OPNET CORE NetMapper support. NetMapper provides up-to-date Microsoft Office Visio network diagrams “on demand.” These diagrams extend beyond simple topology to feature detailed logical views of the network including: Layer 2-3, VPNs, OSPF, VLANs, etc.
- Network Layouts:
 - OPNET CORE 14.5 and 15.0 introduces several new Network Layout Wizards that allow users to quickly build network models, setup wireless devices and run network difference reports. JCSS v9.0 provides support for these wizards and include support for the following:

Table S-1: JCSS Wizards

Under Menu	Wizard Name
Topology	Rapid Configuration
Topology	Deploy Wireless Network
Topology	Open Edge Connectivity Wizard
Topology	Clear Trajectory Assignment
Topology	Random Mobility
Topology	Import STK Orbit
Topology	Shared Risk Groups
Scenario	View Associated Output Tables
Scenario	User-Defined Reports
Scenario	Network Difference Report

Scenario	Object/Attribute Difference Report
Scenario	Live Object/Attribute Difference

- New Project Wizard:
 - JCSS v9.0 has a more streamlined Project Wizard will be simplified to make the workflow more intuitive, while increasing functionality. Overall, the dialog boxes will be combined, new import options added, as well as, utilizing the new OPNET 14.0 mapping capabilities. This New Project Wizard will not affect the existing Task Assistant.
- Explicit Setting of BER, ECC Threshold and Antenna Pattern in wireless Models:
 - Currently in JCSS users can model the bit error rate (BER) for wireless communication channels which is calculated from the SINR by looking up the BER curve for the modulation used by the transmitter and receiver pair. This framework provides an accurate modeling of the parameters which can affect the reception of the packets. However, in some R&D applications the users demand more control over the channel properties particularly over the channel BER to study different protocols and settings. Also, some JCSS users have expressed the need to use the BER values measured from actual wireless networks in the simulation model for further studies of their network architectures.
 - In support of this activity, this feature introduces a new node model and the supporting software to allow the JCSS user to modify the BER, ECC Threshold and Antenna Pattern for individual wireless nodes in a scenario and override the default settings of corresponding node models. The new model not only provides the ability to “playback” the previously defined BER and ECC Threshold values over time, but also it allows the JCSS user to set those values plus possibly the Antenna Patterns on *selected* nodes without the need to open and modify the corresponding node models

Changes Made from JCSS Version 7.0 to JCSS Version 8.0:

- OPNET CORE:
 - JCSS 8.0 is based on OPNET CORE 14.5 PL5
- 64-bit Support for the Scenario Builder (Capacity Planner):
 - In previous versions of JCSS, the Discrete Event Simulation (DES) was the only simulation engine which supported 64-bit architectures. Now, the Capacity Planner and other Scenario Builder features, also support 64-bit machines. This enhancement will allow the user to run larger scenarios inside Capacity Planner without memory problems.
 - During testing, baseline scenarios were able to successfully achieve 5000-6000 flows during a Capacity Planner simulation with 64-bit support. Note that this number can vary depending on the type of 64-bit machine running JCSS.
- EPLRS Model Enhancement:
 - MSG Needline Support: The MSG needline provides hosts with a *few- to-many* communication capability. Messages are transmitted by a select group of Source RSs (RSs that are allowed to transmit data from their hosts to other RSs on the

- needline) and are carried on the MSG needline, either directly or through relays, to other RSs on the needline.
 - EPLRS IP Multicast Support: The EPLRS model now supports IP Multicasting. For additional information, please refer to the EPLRS Model Userguide.
 - EPLRS GUI: Active handler attributes were added to the EPLRS GUI to make it more user friendly.
- Equipment List:
 - The JCSS equipment list GUI functionality was replaced with the new OPNET core functionality “Generate Network Inventory Summary”.
- HAIPE Peer Discovery Model: In support of the DISA EWSE task, an enhanced OPNET Border Gateway Protocol model was developed to support the High Assurance Internet Protocol Encryptor (HAIPE) Peer Discovery (HPD). Enhancements include:
 - BGP Encapsulation SAFI and BGP Tunnel Encapsulation Attribute:
 - [Mohapatra-Rosen] Specifies a BGP extended community attribute attached to BGP UPDATE messages that carry payload prefixes to indicate encapsulation protocol type
 - Defines a new SAFI: Encapsulation SAFI
 - Indicates tunnel type: GRE, L2TPv3, or IP in IP
 - Function to build SAFI octet / modify existing function
 - Additional tunnel types :
 - [Berger] Defines support for IPsec tunnels (AH and ESP)
 - Defines support for IP-in-IP and MPLS-in-IP protected by IPsec transport mode
 - Tunnel type is encoded in new sub-TLV: ”IPsec Tunnel Authenticator Sub-TLV”
- Unified GUI for JCSS Circuit Configuration:
 - JCSS 7.0 introduced a unified circuit GUI that allowed users to deploy circuits between various circuit switch models using a single circuit deployment gui. In JCSS 8.0, the TSSP and MUX models now support circuit deployment via the same GUI interface.
- IER Database:
 - The IER database has been removed from JCSS 8.0. Users can still import IERs via text files as an alternative.
- License Updates:
 - As of JCSS 8.0, to proceed with a Discrete Event Simulation (DES), the user must have both a valid IT-Guru or Modeler License and a Simulation Runtime License.
- JCSS JNN Scenario:
 - The JCSS JNN scenario was enhanced and now supports both Capacity Planner and Discrete Event Simulation.

APPENDIX T: 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 JCSS models. If the custom models do not support the same packet format information as the JCSS 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 JCSS models, each port category must have a self-description port object. For example, MRC-142 (JCSS 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 T-1).

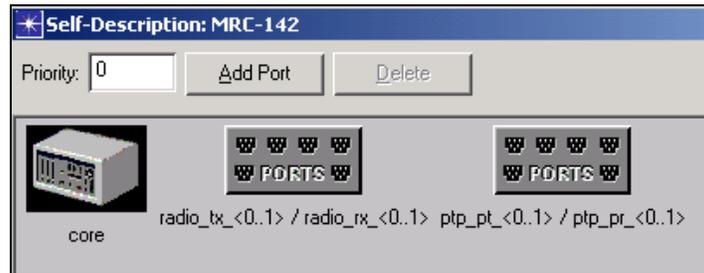


Figure T-1: Self-Description Port Objects

Each port category needs an “*interface type*” characteristic defined for it. This interface type defines the technologies that the set of ports support. These technologies are defined based on the packet formats for each port category. Then, based on this definition, the interface types are defined for each port category depending on what packet formats they support (see for details).

Table T-1: JCSS Port Types and Supported Packet Formats

Port Type	Support Packet Formats	Supported Technologies
ckt	phone_switch	circuit_switched:Voice_LAN
dtg	ckswpkt	circuit_switched:Voice_WAN
eplrs	all packet types	eplrs
lan	ckswpkt ip_dgram_v4 KG194_19 KG84_7 pro_hello_pk pro_wan_pk tssp_frame	serial:DS0 serial:DS1 serial:DS3 serial:T1 serial:T3 serial:OC3 serial:OC12 serial:OC36 serial:OC48

Port Type	Support Packet Formats	Supported Technologies
		serial:OC192 encryptor:KG194_KIV19 encryptor:KG84_KIV7 circuit_switched:Voice_WAN multiplexer:mux_aggregate promina:WAN
prc	prc_data_packet voice_packet	radio_rf:single_channel
sat terminals	tssp_frame	radio:SatelliteTSSP
satellite	all packet types	N/A
wan	pro_cx_pk pro_hello_pk pro_wan_pk	promina:WAN promina:CellXpress

APPENDIX U: IP AUTO ADDRESSING IN CUSTOM MODELS

Overview

For some cases, the makeup of a node model may require that the developer 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. Also refer to the file *nw_ip_modification_support.h* for information on how existing JCSS devices are traversed by auto addressing.

Secondly, add the code needed to support the custom device model. To do this, modify the file *nw_custom_ip_auto_addr.ex.c* starting with the function *nw_custom_ip_traverse ()* that primarily serves to call the correct routine that implements custom IP auto addressing. This function takes the following 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.

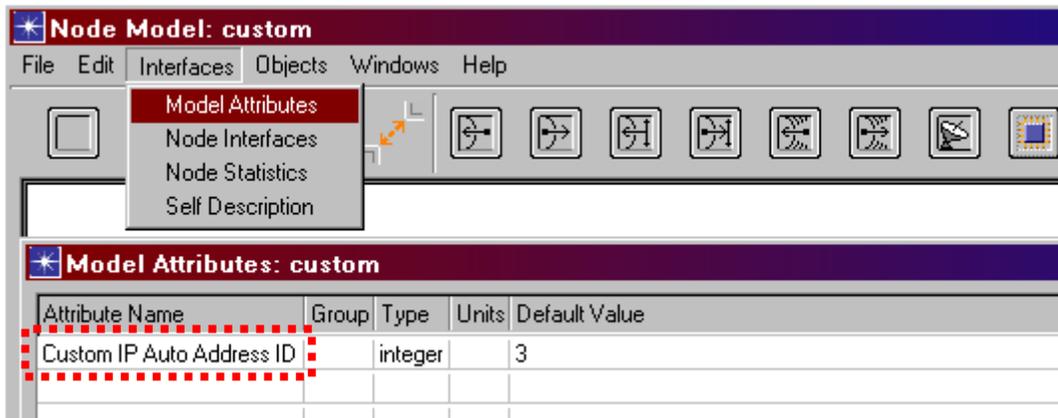


Figure U-1: Custom IP Auto Address ID Attribute

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 V: REFERENCES

- OPNET Modeler Online Documentation
- ACE Whiteboard Tutorial
- JCSS Interface Control Document
- JCSS User Manual
- JCSS Communications Model Verification and Validation Plan
- JCSS Communications Device Model Validation and Verification Plan
- JCSS Equipment Strings

- JCSS Equipment Strings Final Test Plan
- DoD Standard Practice: Documentation of Verification, Validation and Accreditation (VV&A) for Models and Simulations. (MIL-STD-XXX002, Draft of 5 December 2006)
- 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>
- DoD VV&A Documentation Tool

APPENDIX W: JCSS MODEL DEVELOPMENT GUIDE CHECKLIST

The purpose of the checklist in Table W-1 is to help the developer and program managers determine levels of effort to develop new JCSS Standard models or integrate existing models to JCSS.

Table W-1: JCSS Model Development Guide Checklist

JCSS Model Development Guide Checklist		
Model Compliance	Yes/No	Comments
Does the model contain all the JCSS required attributes? Please refer to the <i>JCSS Model Development Guide</i> 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 JCSS version?		
Does the model support Circuits? If so, does it work with the Generic Circuit API?		
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		

JCSS Model Development Guide Checklist		
Model Compliance	Yes/No	Comments
2. Link and circuit utilizations		
3. Bandwidth requirements		
Does the model contain the following model documentations?		
1. Embedded Documentation		
2. User Documentation		
3. Test Plan		
4. Static Testing Results (including parameters used to get the results)		
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 JCSS Standard Palette? 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?		

Filename: JCSS MDG v4.0 - 20090723
Directory: D:\Documents and
Settings\Shery.Salama\Local Settings\Temp\wz89ef
Template: D:\Documents and
Settings\Shery.Salama\Application
Data\Microsoft\Templates\Normal.dot
Title:
Subject:
Author: Smith, Jan D.
Keywords:
Comments:
Creation Date: 7/23/2009 11:17:00 AM
Change Number: 13
Last Saved On: 7/23/2009 12:15:00 PM
Last Saved By: OPNET
Total Editing Time: 30 Minutes
Last Printed On: 9/28/2010 3:37:00 PM
As of Last Complete Printing
Number of Pages: 263
Number of Words: 66,274 (approx.)
Number of Characters: 377,768 (approx.)