A Computer Simulation Model Suite for the Analysis of
All Optical Networks

by
Gregory S. Campbell

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Copyright 1995 Gregory S. Campbell. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute
publicly paper and electronic copies of this thesis document in whole or in
part, and to grant others the right to do so.

Author .................................................................
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by ..........................................................
Steven G. Finn
Principal Research Scientist
Thesis Supervisor

Accepted by ..................................................
F.R. Morgenthaler
Chairman, Departmental Committee on Graduate Students
A Computer Simulation Model Suite for the Analysis of All Optical Networks

by
Gregory S. Campbell

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 1995, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract
In this thesis a suite of models has been developed for the simulation of All Optical Networks (AONs) in the OPNET simulation tool. The models are based on the propagation of pulses through the AON. Pulses are modeled as complex pulse envelopes on a central frequency carrier. As pulses propagate through the network, optical components transform them and delay them appropriately. Probes can be inserted to view the pulses at specific points in the network. This knowledge can help an AON engineer make informed decisions about AON design thereby allowing him to more rapidly test possible network configurations.

Thesis Supervisor: Steven G. Finn

Title: Principal Research Scientist
Acknowledgments

I would first like to thank Dr. Steven Finn for both his technical and personal inspiration, guidance and understanding in helping me produce this work. Without his help this thesis would never have been possible.

I would also like to thank Dr. Roe Hemenway for introducing me to the wonders of non-linear optics, and helping me understand the dynamics of optical components. This thesis is much greater thanks to his inspiration.

My parents and family receive my greatest gratitude for helping me come to M.I.T. and supporting me in all my endeavors. Without them, none of this would have been possible.
# Contents

1 Introduction ........................................................................... 17  
  1.1 Background ..................................................................... 17  
  1.2 AON Model Suite Objectives ......................................... 18  

2 Simulation Concepts ........................................................... 19  
  2.1 OPNET Concepts .......................................................... 19  
    2.1.1 Packets .................................................................. 21  
    2.1.2 Links .................................................................. 21  
    2.1.3 Nodes .................................................................. 21  
    2.1.4 Processes ............................................................ 22  
  2.2 Probing and Analysis ....................................................... 22  
  2.3 A Simple AON Example ............................................... 23  

3 Simulation Structure .......................................................... 25  
  3.1 Simulation Global Variables .......................................... 25  
  3.2 Pulse Structure ............................................................ 27  
  3.3 Noise Structure ............................................................ 29  
  3.4 Ports and Port Structures .............................................. 30  
  3.5 Simulation Flow ............................................................ 34  

4 Component Models ............................................................ 37  
  4.1 Transmitter .................................................................. 39  
  4.2 Optical Fiber .................................................................. 44  
    4.2.1 Fiber Parameters .................................................. 45  
    4.2.2 Propagation Delay ................................................ 47  
    4.2.3 Split-Step Fourier Method ...................................... 47  
    4.2.4 Linear Effects ...................................................... 48  
    4.2.5 Non-Linear effects caused by the Pulse ................. 48  
    4.2.6 Non-Linear effects caused by Pulses at other Frequencies 49  
  4.3 Fused Biconical Coupler ................................................. 51  
  4.4 Star Coupler .................................................................. 54  
  4.5 Optical Amplifier .......................................................... 56  
  4.6 ASE Filter .................................................................... 58  
  4.7 Fiber Fabry-Perot Filter ................................................ 60  
  4.8 Mach-Zehnder Filter ...................................................... 63  
  4.9 Wavelength Division (De)Multiplexer ............................ 66  
  4.10 Wavelength Router ...................................................... 68  
  4.11 Probe .......................................................................... 71  
  4.12 Receiver ...................................................................... 73  

5 Simulation Results .............................................................. 75  
  5.1 Fiber Model ................................................................. 76  
    5.1.1 Dispersion in Linear Regime .................................... 76  
    5.1.2 Non-linearities at the Zero Dispersion Point ........... 80
5.2 Filters ................................................................. 83
   5.2.1 Fabry-Perot Filter ........................................... 83
   5.2.2 Mach-Zehnder Filter ......................................... 86
5.3 Fused Biconical Coupler ........................................... 89

6 Conclusion ............................................................... 93

Appendix A Component Process Model Reports ............................. 95
   A.1 aon_xmt0 ......................................................... 96
   A.2 aon_xmt_seq .................................................... 98
   A.3 aon_xmt0_sech ............................................... 101
   A.4 aon_xmt_sech_seq ............................................. 103
   A.5 aon_fib .......................................................... 106
   A.6 aon_fbc .......................................................... 110
   A.7 aon_stc .......................................................... 113
   A.8 aon_amp .......................................................... 117
   A.9 aon_ase .......................................................... 121
   A.10 aon_fabry ....................................................... 124
   A.11 aon_mzf ........................................................ 127
   A.12 aon_wdm ........................................................ 130
   A.13 aon_rou ........................................................ 134
   A.14 aon_probe ....................................................... 138
   A.15 aon_rcv ........................................................ 141

Appendix B Supporting Code ............................................. 145
   B.1 Transmitter Support Code ...................................... 146
      aon_xmt.ex.h ................................................... 146
      aon_xmt.ex.c ................................................... 146
   B.2 Optical Fiber Support Code .................................... 148
      aon_fib.ex.h ................................................... 148
      aon_fib.ex.c ................................................... 150
   B.3 Fused Biconical Coupler Support Code ......................... 164
      aon_fbc.ex.h ................................................... 164
      aon_fbc.ex.c ................................................... 164
   B.4 Star Coupler Support Code .................................... 167
      aon_stc.ex.h ................................................... 167
      aon_stc.ex.c ................................................... 167
   B.5 Optical Amplifier Support Code ................................ 169
      aon_amp.ex.h ................................................... 169
      aon_amp.ex.c ................................................... 170
   B.6 ASE Filter Support Code ....................................... 174
      aon_ase.ex.h ................................................... 174
      aon_ase.ex.c ................................................... 174
   B.7 Fabry-Perot Filter Support Code ................................ 176
      aon_fab.ex.h ................................................... 176
      aon_fab.ex.c ................................................... 176
B.8 Mach-Zehnder Filter Support Code ........................................... 178
    aon_mzf.ex.h .............................................. 178
    aon_mzf.ex.c .............................................. 178
B.9 Wavelength Division (De)Multiplexer Support Code ...................... 180
    aon_wdm.ex.h .............................................. 180
    aon_wdm.ex.c .............................................. 180
B.10 Wavelength Router Support Code ........................................... 182
    aon_rou.ex.h .............................................. 182
    aon_rou.ex.c .............................................. 182
B.11 Receiver and Probe Support Code .......................................... 184
    aon_rcv.ex.h .............................................. 184
    aon_rcv.ex.c .............................................. 184
B.12 Complex Mathematics Support Code ...................................... 188
    cmath.ex.h .............................................. 188
    cmath.ex.c .............................................. 188
B.13 Linear Transfer Function Support Code .................................. 193
    aon_lin.ex.h .............................................. 193
    aon_lin.ex.c .............................................. 193
B.14 Pipeline Stages ............................................................ 195
    aon_ps.ex.h .............................................. 195
    aon_propdel.ps.c ......................................... 195
    aon_proprcv.ps.c ......................................... 197
    aon_txdel.ps.c ........................................... 198
    aon_txrcv.ps.c ........................................... 199

Appendix C Usage Comments 201
List of Figures

Figure 2-1: Network level model of a metropolitan area All Optical Network. ........ 19

Figure 2-2: Node level model of a FBC node. Packets enter the node through point to point receivers and exit the node through point-to-point transmitters. The components in the node send packets to each other through packet streams. ......................... 20

Figure 2-3: Process level model of a simple FSM containing one forced state (init) and one unforced state (steady). ............................................ 21

Figure 2-4: Simple AON Example to demonstrate how the AON Model Suite and OPNET work together to simulate an All Optical Network. ....................... 24

Figure 3-1: The pulse shape is defined by complex samples over a span of AonI_Duration seconds. Here, AonI_Nu = 5 and AonI_Duration = 300 ps. ............... 28

Figure 3-2: Mach-Zehnder Filter port layout. ........................................ 30

Figure 3-3: Flow diagram for linear component ..................................... 34

Figure 3-4: Flow diagram for non-linear component ............................... 35

Figure 4-1: AON Transmitter icon and port layout. Incoming packets on port 0 are discared. .............................................................. 39

Figure 4-2: Gaussian pulse amplitude (m = 1, t₀ = 100 ps, P₀ = 0.1 W). ........... 40

Figure 4-3: Super-Gaussian pulse amplitude (m = 3, t₀ = 100 ps, P₀ = 0.1 W). .... 40

Figure 4-4: Hyperbolic-secant pulse amplitude (m = 1, t₀ = 100 ps, P₀ = 0.1 W). .. 41

Figure 4-5: Four-bit Finite Sequence Machine: given a non-zero initial state, this machine will generate all four bit sequences for a total sequence length of bits [Pet, 148]. This particular machine has an initial state equal to 1, and a pn connections parameter equal to 3 because connections 0 and 1 are connected. In this machine, bit 3 in state n+1 is equal to the exclusive or of the connected bits. ...................... 43

Figure 4-6: AON Fiber icon and port layout. ........................................... 44

Figure 4-7: AON Fused Biconical Coupler icon and port layout. .................... 51

Figure 4-8: Amplitude of H(f) of Fused Biconical Coupler. .......................... 53

Figure 4-9: AON Star Coupler icon and port layout. .................................. 54

Figure 4-10: AON Amplifier icon and port layout. The amplifier is a unidirectional device. Incoming packets on port 1 are discarded. ......................... 56

Figure 4-11: AON ASE Filter icon .......................................................... 58

Figure 4-12: Amplitude of H(f) of ASE Filter. (FSR = 0.5 THz, a = 1dB, W = 0.25 THz) 59
Figure 4-13: AON Fiber Fabry-Perot icon ........................................ 60

Figure 4-14: Amplitude of $H(f)$ of Fabry-Perot Filter for three different values of finesse. FSR = 0.5 THz, $T(f)_{max} = 0.9$. ........................................ 62

Figure 4-15: Phase of $H(f)$ of Fabry-Perot Filter for three different values of finesse. FSR = 0.5 THz, $T(f)_{max} = 0.9$. ........................................ 62

Figure 4-16: AON Mach-Zehnder Filter icon and port layout. ................... 63

Figure 4-17: Amplitude of $H_{acr}(f)$ and $H_{opp}(f)$ of Mach-Zehnder Filter for FSR = 0.5 THz. ..................................................... 65

Figure 4-18: Phase of $H_{acr}(f)$ and $H_{opp}(f)$ of Mach-Zehnder Filter for FSR = 0.5 THz. ..................................................... 65

Figure 4-19: AON Wavelength Division (De)Multiplexer icon and port layout. ................... 66

Figure 4-20: AON Router icon and port layout ........................................ 68

Figure 4-21: AON Probe icon and port layout. ........................................ 71

Figure 4-22: AON Receiver icon ..................................................... 73

Figure 5-1: Network and Node Level descriptions of test network. The links in this network are simplex. This is because the object of the experiment is to study the effects of dispersion on receivability in the absence of other effects. ..................... 76

Figure 5-2: A pulse is flattened due to dispersion after going through sections of fiber with a positive group velocity dispersion coefficient. The flattened pulse is chirped by. The original pulse is reconstructed by going through a section of fiber that “unchirps” the pulse by inducing an equal and opposite amount of chirp. ..................... 77

Figure 5-3: The bit stream coming out of the transmitter. The eye is fully dilated, with a maximum opening of 1 mWatt. The signal can be received easily. ..................... 78

Figure 5-4: The bit stream after going through 50 km of dispersive fiber. The eye is still quite dilated, with a maximum opening of 0.43 mWatts. The signal can still be received. ..................................................... 78

Figure 5-5: The bit stream after going through 100 km of dispersive fiber. The eye is nearly shut, with a maximum opening of 75 microWatts. The signal can be received only with difficulty. ..................................................... 79

Figure 5-6: The bit stream after reconstruction. The eye is fully dilated, with a maximum opening of 0.95 mWatts. The signal can again be received easily. ..................... 79

Figure 5-7: Node level description of network for testing non-linearities at the zero dispersion point. The links in this model are simplex. This is because the object of the model is to examine the effects of SPM on the complex phase envelope in the absence of other effects. ..................................................... 80
Figure 5-8: Pulse amplitude before and after traveling through the fiber. The pulse amplitude has not changed appreciably. ........................... 81

Figure 5-9: Pulse phase before and after traveling through the fiber. SPM has altered the pulse phase considerably. ........................................... 81

Figure 5-10: Fourier Transform Amplitude of the pulse before and after traveling through the fiber. SPM has broadened the spectrum significantly. ........................... 82

Figure 5-11: Fourier Transform phase of the pulse before and after traveling through the fiber. SPM has had a profound effect. ........................... 82

Figure 5-12: Node level description of network for testing the Fabry-Perot filter. The pulse entering the filter is the same pulse generated in section 5.1.2, a gaussian chirped by SPM. .................................................. 83

Figure 5-13: The pulse amplitude before and after going through the Fabry-Perot filter. Because the carrier frequency lies centered on a passband of the Fabry-Perot filter, more energy is lost in sections of the pulse where the spectral components are further from the carrier frequency. Because this pulse was chirped by SPM, the sections of the pulse where the absolute value of the slope of the complex pulse envelope is high are the sections of the pulse with spectral components far from the carrier frequency. .................................................. 84

Figure 5-14: The pulse phase before and after going through the Fabry-Perot filter. ........................................ 84

Figure 5-15: The amplitude of the Fourier Transform of the pulse before and after going through the Fabry-Perot filter. The large side lobes of the Fourier Transform are attenuated considerably by the filter. .................................................. 85

Figure 5-16: The phase of the Fourier Transform of the pulse before and after going through the Fabry-Perot filter. .................................................. 85

Figure 5-17: Node level description of network for testing the Fabry-Perot filter. The pulse entering the filter is the same pulse generated in section 5.1.2, a gaussian chirped by SPM. .................................................. 86

Figure 5-18: The pulse amplitude coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. Because the carrier frequency lies centered on a FSR of the Mach-Zehnder filter, the pulse is split into two pulses with one pulse getting almost all of the energy. A null of the transfer function for the pulse going to RCV lies directly on the carrier frequency, and this creates a null for components of the pulse with frequencies equal to the carrier frequency. This corresponds to flat sections of the pulse. This is the reason for the null in the center of the pulse. ........................................ 87

Figure 5-19: The pulse phase coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. .................................................. 87
Figure 5-20: The amplitude of the Fourier Transform coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. Because the carrier frequency lies centered on a FSR of the Mach-Zehnder filter, the pulse is split into two pulses with one pulse getting almost all of the energy. A null of the transfer function for the pulse going to RCV lies directly on the carrier frequency, and this creates a null in the amplitude of the Fourier Transform at the carrier frequency. ............ 88

Figure 5-21: The phase of the Fourier Transform coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. .......................... 88

Figure 5-22: Node level description of network for testing the Fused Biconical Coupler. The pulse entering the FBC is the same pulse generated in section 5.1.2, a gaussian chirped by SPM. .................................................... 89

Figure 5-23: The pulse amplitude coming in through port 0 and leaving through ports 2 and 3 of the Fused Biconical Coupler. Because the carrier frequency lies near an area of the FBC transfer functions where the two pulses are split roughly evenly the pulse power is split roughly evenly. Because the slopes of the transfer functions are so great in this area, the one pulse receives most of its energy from the higher frequency spectral components, while the other pulse receives most of its energy from the lower frequency spectral components. ........................................ 90

Figure 5-24: The pulse phase coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. ................................................. 90

Figure 5-25: The amplitude of the Fourier Transform of the pulse coming in through port 0 and leaving through ports 2 and 3 of the Fused Biconical Coupler. The FBC transfer functions send most of the higher frequency energy to RCV, and most of the lower frequency energy to RCVB. ............................................. 91

Figure 5-26: The phase of the Fourier Transform of the pulse coming in through port 0 and leaving through ports 2 and 3 of the Fused Biconical Coupler. ............... 91
List of Tables

| Figure 3-1: Simulation Global Variables | 26 |
| Figure 3-2: Pulse Structure | 29 |
| Figure 4-1: Standard Transmitter Parameters | 39 |
| Figure 4-2: Additional Parameters for Gaussian Transmitter | 41 |
| Figure 4-3: Additional Parameters for Hyperbolic Secant Transmitter | 41 |
| Figure 4-4: Additional Parameters for Single Pulse Transmitter | 42 |
| Figure 4-5: Additional Parameters for Single Pulse Transmitter | 42 |
| Figure 4-6: Optical Fiber Parameters | 45 |
| Figure 4-7: Fused Biconical Coupler Parameters | 51 |
| Figure 4-8: Star Coupler Parameters | 55 |
| Figure 4-9: Optical Amplifier Parameters | 57 |
| Figure 4-10: ASE Filter Parameters | 58 |
| Figure 4-11: Fiber Fabry-Perot Filter | 61 |
| Figure 4-12: Mach-Zehnder Filter | 64 |
| Figure 4-13: Wavelength Division (De) Multiplexer | 67 |
| Figure 4-14: Wavelength Router | 69 |
| Figure 4-15: Probe | 72 |
| Figure 4-16: Receiver | 73 |
Chapter 1: Introduction

1.1 Background

All Optical Networks (AONs) are data networks in which nodes are connected end-to-end optically. Other types of networks use optical links, but AONs are unique in that once an end node transfers the data stream into an optical signal, the optical signal is not converted back into electrical voltages in an electronic circuit until it reaches its destination. Other types of networks (e.g. SONET) which utilize optical components make this transformation at each intermediate node in the network.

While in traditional networks which use optical links an optical signal is electrically “regenerated” at each node, in an AON any transformations that the signal undergoes in transit are propagated through the network. This leads to some interesting problems in AONs. Some of these problems are aggravated analogs to problems seen in traditional optical networks, while some are entirely specific to AONs. For example, in a standard optical network dispersion and non-linearities in the fiber limit the distance-bitrate product by smearing nearby optical signals together [Gre, 39]. A standard network can counteract this by placing intermediate nodes closer together in order to limit the distance-bitrate product for a given link. In an AON, this is not a valid solution -- as an optical signal goes through an optical node in an AON, it is not regenerated. Specific to AONs is the optical routing problem. This problem deals with the networks ability to direct data flow between two end-points. Standard networks using optical links are not concerned with optical routing.

In order to make effective decisions on the design of AONs, the AON engineering team should be able to rapidly prototype and test ideas. Unfortunately, testing on a real AON testbed is time consuming, and resources are expensive. Therefore, simulation can be an important and useful tool in the development of AON technology. Simulation can help the AON engineering team to determine which experiments to actually perform on the testbed, aiding in the efficient design of the AON, and shortening the development cycle. In
this thesis a powerful set of models is developed for the simulation of AONs in order to aid in their development.

1.2 AON Model Suite Objectives

The three most important characteristics of a simulation tool are ease of use for rapid prototyping, simulation accuracy and speed, and ease of use in the display and analysis of simulation results. This thesis attempts to address these critical areas while accurately modeling pulse transmission in AONs.

The AON Model Suite is built on the OPNET simulation platform. OPNET (OPtimized Network Engineering Tools) is a product of MIL3, Inc. designed as a simulation engine geared towards data networks. The AON Model Suite/OPNET combination provides a stable, efficient, easy to use simulation platform which allows:

- Rapid prototyping of an All Optical Network
- Accurate and fast simulation
- Powerful graphical analysis tools

Additionally, OPNET has been designed to provide a high level of modeling flexibility in model development, allowing for efficient further development of complex AON components without sacrificing model accuracy.
Chapter 2: Simulation Concepts

The All Optical Network Model Suite is built on top of the OPNET simulation platform. The OPNET simulation platform yields a stable, efficient simulation environment on which to place the AON Model Suite. OPNET has a number of concepts used by the AON Model Suite. Additionally, OPNET provides powerful probing and analysis capabilities.

2.1 OPNET Concepts

OPNET divides the modeling hierarchy into three logical levels called the Network level, the Node level and the Process level. These levels each deal with a different aspect of a network. The Network level is composed of nodes specified in the Node level. Likewise, the Node level is composed of components, some of which have processes specified in the Process level. Components communicate with each other through the use of packets.

The Network level (See Figure 2-1) deals with the spatial and topological distribution of OPNET nodes and the links between those nodes. Nodes have inputs and outputs and are connected by links. Nodes are designed at the Node level. Links are connections between nodes along which packets travel. As a packet goes through a link a series of procedures operate on the packet. These procedures are defined in the AON Model Suite to model optical fiber.

![Figure 2-1: Network level model of a metropolitan area All Optical Network.](image)

The Node level (See Figure 2-2) deals with the logical connection of components within a node. Components are connected by packet streams. Packet streams are logical connections between components along which packets travel. Packet streams merely deliver
packets with no delay. Some Node level components exhibit properties, such as propagation delay and insertion loss, which can be modeled with a process designed at the Process level. Other Node level components are used only as connections to links at the Network level. These components are called point-to-point transmitters and receivers.

The Process level (See Figure 2-3) allows for the design of Finite State Machine processes found in many components in the Node level. This is where one finds the heart of the AON Model Suite. These FSM based processes alter and delay the packets entering the component in order to model the effects of the component.

This hierarchy lends itself easily to the development of the AON Model Suite. The Network and Node level hierarchy allow for easy organization of an AON, while the Process level allows for precise modeling of the optical components.

**Figure 2-2:** Node level model of a FBC node. Packets enter the node through point to point receivers and exit the node through point-to-point transmitters. The components in the node send packets to each other through packet streams.
2.1.1 Packets

*Packets* are the primary means of communication in OPNET. Packets travel along *links* and *packet streams*. The AON Model Suite uses *packets* to simulate the movement of light in an AON. A *packet* either holds a single pulse or holds data representing a change in the noise level.

2.1.2 Links

*Links* are connections between nodes at the *Network* level. Each *link* represents an optical fiber or a bundle of optical fibers. Optical power travels along *links* in packets. A *link* is defined by a number of procedures called the *Transceiver Pipeline* that, in the AON Model Suite, modify traversing packets and calculate propagation delay in order to simulate light traveling through an optical fiber.

2.1.3 Nodes

*Nodes* are structures which are designed at the *Node* level and instantiated at the *Network* level. *Nodes* are composed of *components*. While OPNET provides a wide variety of *component classes*, the AON Model Suite only uses three -- the *processor class*, the *point-to-point transmitter class* and the *point-to-point receiver class*. The *point-to-point transmitter class* and *point-to-point receiver class* each supports only one type of *component* in the AON Model Suite. The *point-to-point transmitter class* supports the *point-to-point transmitter component*. The *point-to-point receiver class* supports the *point-to-point receiver component*. The *processor class*, on the other hand, supports a large number of *component* types, such as star couplers, optical fibers and optical amplifiers. These *component* types are differentiated by the *process* specified for the *processor class* based *component*.
The point-to-point transmitter component sends packets along links at the Network level. The point-to-point receiver component receives packets from links at the Network level. The processor class based components manipulate packets according to a process designed in the Process level.

2.1.4 Processes
In the AON Model Suite, processes are designed to model the properties of an optical component. These processes are designed as Finite State Machines at the Process level, and are made up of:

- **Unforced states**
- **Forced states**
- **Transitions** between states

Both types of states contain two sequential sections of C program code. When a process enters an unforced state, it executes the C code in the first section of the state and then exits. The unforced state resumes where it left off upon being woken up either by a packet arrival or some other event, such as an event scheduled by the process itself, and executes the C code in the second section of the state and progresses along a transition to the next state. When a process enters a forced state, it executes the C code in both of the sequential sections of the state and progresses along a transition to the next state. The transition taken can depend upon the current state of the process.

2.2 Probing and Analysis

OPNET allows for the collection of statistics through the use of the Probe Editor. One can specify probes in the Probe Editor in order to log statistics written out by components in the simulation. Each processor class component in an OPNET simulation has an array of outstats. An outstat is a variable that changes with time. Each probe records an outstat from a single component in a single node in the network. In the AON Model Suite there are two component types used to probe outstats. These are the probe component and the receiver component.
2.3 A Simple AON Example

The following is a simple example to show how the AON Model Suite and OPNET work together to simulate an All Optical Network. The example is an amplifier - fiber - filter network (See Figure 2-4). The first event in the simulation is the transmission of a pulse packet by the transmitter component. A pulse packet is represented by the symbol $\triangle$. This pulse travels over a packet stream to an EDF Amplifier component. The amplifier modifies the packet by multiplying the signal by a complex transfer function in the Fourier domain, and sends it on with a specified delay ($\Delta$). Additionally, the amplifier generates a noise packet ($\nabla$). A noise packet is represented by the symbol $\nabla$. These packets travel over a packet stream to the point-to-point transmitter component, the device used to put packets on the optical fiber represented at the Network level by a point-to-point link. The point-to-point transmitter component sends the pulse and noise packets over the link. These packets travel over the link, causing the AON Model Suite defined Pipeline Stages to execute. These procedures simulate the fiber effects by altering the pulse and noise packets. The link then forwards the modified packets ($\nabla$) to the point-to-point receiver component. The point-to-point receiver component forwards these packets ($\nabla$) to the fiber Fabry-Perot filter component, which modifies the packets by passing them through a complex transfer function and sends them with a specified delay to the receiver component. The receiver component collects statistics and destroys the packets.
Figure 2-4: Simple AON Example to demonstrate how the AON Model Suite and OPNET work together to simulate an All Optical Network.
Chapter 3: Simulation Structure

In order to model an All Optical Network, one must have a model of the optical signals traveling through the system, as well as models of each of the optical components. The AON Model Suite is based on the propagation of pulses and noise through optical components. As a pulse travels through the AON, it is passed from component to component and manipulated appropriately depending upon the component type and parameters. Noise also passes from component to component and is handled appropriately according to the component type and parameters.

3.1 Simulation Global Variables

Several variables are maintained in the AON Model Suite that need to be accessed by every component. These global variables describe the standard parameters of the pulse and noise data, and are used by the components in manipulating the pulse and noise data.

The following global variables are maintained by the AON Model Suite:

- *AonI_Nu* describes the number of complex samples per pulse. $2^0$ is the number of complex samples per pulse. The number of samples per pulse is described this way as a result of the use of the radix-2 Fast Fourier Transform algorithm throughout the model suite.

- *AonI_Len* is the cached value of $2^0$, the number of complex samples per pulse.

- *AonI_Duration* is the number of picoseconds sampled for each pulse. While pulses may have a shorter duration than *AonI_Duration*, a longer duration results in aliasing of pulse data.

- *AonI_Low_Freq* is the lowest noise frequency, in THz, tracked by the AON Model Suite.
• *AonI* High*Freq* is the highest noise frequency in THz tracked by the AON Model Suite.

• *AonI* N*Segment* is the number of noise frequency bands tracked by the AON Model Suite. Increasing *AonI* N*Segment* increases the accuracy of the results due to noise in the model suite.

• *AonI* Min*Power* is the minimum significant power in the simulation. Pulse and noise packets with power less than *AonI* Min*Power* are not transmitted.

• *AonI* Min*Change* is the minimum significant percentage change of noise power in a noise band in the simulation. If the noise power changes by a smaller percentage than *AonI* Min*Change*, the change will not be propagated.

• *AonI* Connectors is a flag indicating whether or not connectors are to be modeled in the simulation. If *AonI* Connectors is set, attenuation and reflection will occur at connections between optical components.

• *AonI* Attenuation is the power attenuation factor of a connector. This variable is only significant if *AonI* Connectors is set.

• *AonI* Reflection is the power reflectance factor of a connector. This variable is only significant if *AonI* Connectors is set.

• *AonI* Delay is the delay associated with a connector. This variable is only significant if *AonI* Connectors is set.

• *AonI* Unco*Refl* is the power reflection factor of an unterminated or unconnected port.

Table 3-1: Simulation Global Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AonI_Nu</td>
<td>integer</td>
<td>N/A</td>
<td>Each pulse shape is described by $2^n$ complex samples.</td>
</tr>
</tbody>
</table>
### Table 3-1: Simulation Global Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AonI_Len</td>
<td>integer</td>
<td>N/A</td>
<td>Cached value of $2^v$.</td>
</tr>
<tr>
<td>AonI_Duration</td>
<td>double</td>
<td>ps</td>
<td>The number of picoseconds sampled for each pulse.</td>
</tr>
<tr>
<td>AonI_Low_Freq</td>
<td>double</td>
<td>THz</td>
<td>Lowest noise frequency tracked by models.</td>
</tr>
<tr>
<td>AonI_High_Freq</td>
<td>double</td>
<td>THz</td>
<td>Highest noise frequency tracked by models.</td>
</tr>
<tr>
<td>AonI_N_Segment</td>
<td>integer</td>
<td>N/A</td>
<td>Number of frequency bands into which the noise spectrum is divided.</td>
</tr>
<tr>
<td>AonI_Min_Power</td>
<td>double</td>
<td>W</td>
<td>Minimum power propagated through the system.</td>
</tr>
<tr>
<td>AonI_Min_Change</td>
<td>double</td>
<td>N/A</td>
<td>Minimum percentage change of noise power propagated through the system.</td>
</tr>
<tr>
<td>AonI_Connectors</td>
<td>integer</td>
<td>N/A</td>
<td>Enables modeling of connectors when not equal to 0.</td>
</tr>
<tr>
<td>AonI_Attenuation</td>
<td>double</td>
<td>dB</td>
<td>Attenuation factor of connectors. Only valid when AonI_Connectors is set.</td>
</tr>
<tr>
<td>AonI_Reflection</td>
<td>double</td>
<td>N/A</td>
<td>Reflectance of connectors. Only valid when AonI_Connectors is set.</td>
</tr>
<tr>
<td>AonI_Delay</td>
<td>double</td>
<td>ps</td>
<td>Delay of a connector. This variable is only significant if AonI_Connectors is set.</td>
</tr>
<tr>
<td>AonI_Unco_Refl</td>
<td>double</td>
<td>N/A</td>
<td>Reflectance of unconnected port.</td>
</tr>
</tbody>
</table>

### 3.2 Pulse Structure

Pulses are the core of the AON Model Suite. Pulses are described by a data structure, the most important field of which describes the shape. This field holds a pointer to an array of $2^v$ complex values. Each element in the array corresponds to a sample of the complex
envelope of the pulse. The complex envelope of the pulse describes the pulse shape in terms of amplitude and phase (See Figure 3-1). One advantage to keeping track of the complex envelope is the ability to transform the pulse using both linear and non-linear models of pulse propagation. The choice of 2^m samples is for efficiency in computing the Fast Fourier Transform algorithm.

![Figure 3-1: The pulse shape is defined by 2^m complex samples over a span of AonI_Duration seconds. Here, AonI_Nu = 5 and AonI_Duration = 300 ps.](image)

The pulse data structure consists of seven fields:

- The *source* field holds the component identifier of the transmitter component that generated the pulse.
- The *timestamp* field holds the time at which the pulse was transmitted.
- The *freq* field holds the pulse carrier frequency.
- The *id* field holds an integer that identifies the pulse. Each pulse has a unique id upon transmission, and this number identifies the pulse. When a pulse is split or otherwise copied, this number is also copied.
- The *peak_power* field holds the peak power of the pulse.
• The width field holds the FWHM (full width half-maximum) width of the pulse.

• The shape field holds an array of the complex samples of the pulse. There are $2^v$ complex samples per pulse, where $v$ is equal to $AonI_Nu$. The samples cover $AonI_Duration$ picoseconds.

Table 3-2: Pulse Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>integer</td>
<td>N/A</td>
<td>Transmitter component identifier</td>
</tr>
<tr>
<td>timestamp</td>
<td>double</td>
<td>ps</td>
<td>Simulation time at pulse transmission</td>
</tr>
<tr>
<td>freq</td>
<td>double</td>
<td>THz</td>
<td>Frequency of the pulse carrier</td>
</tr>
<tr>
<td>id</td>
<td>integer</td>
<td>N/A</td>
<td>Pulse identifier</td>
</tr>
<tr>
<td>peak_power</td>
<td>double</td>
<td>W</td>
<td>Peak pulse power</td>
</tr>
<tr>
<td>width</td>
<td>double</td>
<td>ps</td>
<td>FWHM pulse width</td>
</tr>
<tr>
<td>shape</td>
<td>array</td>
<td>N/A</td>
<td>Array of complex samples</td>
</tr>
</tbody>
</table>

3.3 Noise Structure

Noise is tracked in a number of evenly distributed frequency bands, specified by $AonI_N_Segment$, between the low frequency specified by $AonI_Low_Freq$, and the high frequency specified by $AonI_High_Freq$. Noise is treated throughout the models as incoherent and of low power. These assumptions allow for ignoring non-linear effects with respect to noise. Noise data travels through the system in packets. Each noise packet holds a noise data structure. The noise data structure includes the following two fields:
• *freq_bin* is an integer from 0 to \((AonI_N\_Segment - 1)\) that indicates which frequency band this structure describes. The center frequency of the noise band is

\[
f_{bin} = AonI\_Low\_Freq + \left(\frac{freq\_bin + 0.5}{AonI\_N\_Segment}\right)(AonI\_High\_Freq - AonI\_Low\_Freq)
\]

• *power* is the optical power level of the noise in the band.

Noise travels through the simulation absolutely. That is, packets holding noise information hold the current noise in a band. Noise in a noise band \(\delta f\) at a port is equal to the power value in the last noise packet describing that band:

\[
N_{\delta f} = N_{\delta f, \text{last}}
\]

where \(N_{\delta f, \text{last}}\) is the noise information in the last packet describing the noise band.

### 3.4 Ports and Port Structures

Each component in the AON Model Suite communicates with other components using packets which travel over packet streams or links. Each packet stream is associated with a source port and a destination port. Components send packets over packet streams by sending them through ports (See Figure 3-2).

![Figure 3-2: Mach-Zehnder Filter port layout.](image-url)
When packets containing pulse or noise data arrive at a port they are transformed and
delayed by the component.

In a linear component with $N$ ports, let

$$ P_{\text{out}}^o(t) = \begin{bmatrix} P_0^o(t) \\ \vdots \\ P_N^o(t) \end{bmatrix} \quad , \quad P_{\text{in}}^i(t) = \begin{bmatrix} P_0^i(t) \\ \vdots \\ P_N^i(t) \end{bmatrix} \quad (1 \times N) $$

be the port outputs and inputs.

The data coming in at a port can be divided into pulse and noise data, and each type is
dealt with differently. Because pulses travel through the simulator as complex amplitude
envelopes on a central carrier frequency, when a pulse is passed through a port it is manip-
ulated by a transformation matrix as follows:

$$ P_{\text{pulse}}^o(t) = S_{\text{pulse}} P_{\text{pulse}}^i(t - D) \quad \text{where} \quad S_{\text{pulse}} = \begin{bmatrix} T_{1,1} & \cdots & T_{N,1} \\ \vdots & \ddots & \vdots \\ T_{1,N} & \cdots & T_{N,N} \end{bmatrix} $$

where $T_{i,j}$ is a transformation of amplitude, and $D$ is a delay. Depending upon the compo-
nent, the delay $D$ can be either dependent upon the pulse frequency ($D(f)$), or upon the
pulse frequency and component state ($D(f, \text{state})$).

Noise, on the other hand, travels through the simulation as power. Thus, when noise is
passed through a port it is manipulated by a transformation matrix as follows:

$$ P_{\text{noise}}^o(t) = S_{\text{noise}} P_{\text{noise}}^i(t - D) \quad \text{where} \quad S_{\text{noise}} = \begin{bmatrix} |T_{1,1}|^2 & \cdots & |T_{N,1}|^2 \\ \vdots & \ddots & \vdots \\ |T_{1,N}|^2 & \cdots & |T_{N,N}|^2 \end{bmatrix} $$
where $|T_{ij}|^2$ is a transformation of power, and $D$ is a delay. Again, depending upon the component, the delay $D$ can be either dependent upon the pulse frequency ($D(f)$), or upon the pulse frequency and component state ($D(f, state)$).

The pulse and noise transformations in a linear system are described in general by the linear $N \times N$ $S$ matrices

$$S_{L, \text{pulse}} = \begin{bmatrix} H_{1,1}(f) & \cdots & H_{N,1}(f) \\ \vdots & \ddots & \vdots \\ H_{1,N}(f) & \cdots & H_{N,N}(f) \end{bmatrix} \quad \text{and} \quad S_{L, \text{noise}} = \begin{bmatrix} |H(f,1,1)|^2 & \cdots & |H(f,N,1)|^2 \\ \vdots & \ddots & \vdots \\ |H(f,1,N)|^2 & \cdots & |H(f,N,N)|^2 \end{bmatrix}$$

where $H(f)$ is a linear complex transfer function. The delay in a linear system, $D(f)$, imposed by the component is a function of signal or noise frequency.

The transformation in a non-linear system is described in general by the $N \times N$ matrices

$$S_{NL, \text{pulse}} = \begin{bmatrix} H_{1,1}(f, state) & \cdots & H_{N,1}(f, state) \\ \vdots & \ddots & \vdots \\ H_{1,N}(f, state) & \cdots & H_{N,N}(f, state) \end{bmatrix}$$

and

$$S_{NL, \text{noise}} = \begin{bmatrix} |H(f, state,1,1)|^2 & \cdots & |H(f, state,N,1)|^2 \\ \vdots & \ddots & \vdots \\ |H(f, state,1,N)|^2 & \cdots & |H(f, state,N,N)|^2 \end{bmatrix}$$

where $state$ describes the state of the component. $H(f, state)$ is a non-linear complex transfer function. The delay in a non-linear system, $D(f, state)$, imposed by the component is a function of signal or noise frequency and the component state.

In order to maintain the state of each source and destination port, the AON Model Suite instantiates the following port structures:
• *Port Pulse:* This structure holds a list of pulses associated with the times they arrived. This type of structure is necessary in non-linear models in order to maintain an accurate representation of the state of the pulses coming into a port.

• *Port Noise In:* This structure holds an array of $AonI_N_Segment$ noise power values. This type of structure is necessary in order to maintain an accurate representation of the state of the noise coming into a port.

• *Port Noise Out:* This structure holds an array of $AonI_N_Segment$ noise power values that represent the power leaving a port in addition to an array of $AonI_N_Segment$ noise power values that track the noise power values that the component has sent through that port to the adjacent component. Essentially, when a change in the noise power value in a specific frequency band

$$\Delta P_{change} = \frac{|N_{8f, current} - N_{8f, last transmitted}|}{N_{8f, current} + N_{8f, last transmitted}}$$

is less than $AonI_Min_Change$

$$\Delta P < AonI_Min_Change$$

the change is deemed insignificant and is not sent. No changes are sent until the current state is significantly different from the transmitted state. This structure, while not strictly necessary, can improve the performance of the simulation significantly.
3.5 Simulation Flow

Simulation flow is determined by the flow of pulses and noise through the AON components. Pulses and noise travel in OPNET packets along OPNET packet streams and point to point links. When a pulse or noise packet arrives at a component input port, the component identifies the packet type and handles the data appropriately. If the component is linear (See Figure 3-3), such as a filter or star, the pulse or noise data is transformed, the ports are updated, and the pulse is sent along to the next component. If the component is
non-linear the pulse or noise data is stored in a port structure, and an event is scheduled for a time in the future when the pulse or noise data is to be passed on to the next component (See Figure 3-4).
Chapter 4: Component Models

The All Optical Network Model Suite includes a number of component models. There are essentially six fundamental component types:

- The transmitter components generate pulses. They are the only components that can initiate a pulse travelling through the network.
- The receiver component destroys pulses. It is the only component that causes pulse or noise data to stop propagating in the network.
- The probe component probes pulse and noise data.
- The point-to-point transmitter and point-to-point receiver send and receive pulse and noise data over links representing optical fiber.
- Linear components receive pulse and noise data, transform it, and then send it on with a delay to the appropriate component in the network.
- Non-linear components receive pulse and noise data, remember it, and set an interrupt for the time when they are supposed to transmit the pulse or noise data. At this point, the non-linear effects have been determined, and the component transforms the pulse or noise data appropriately.

These six component types are divided into three classes:

- **Essential**: These components are essential to every simulation. The essential components are the transmitter components, the probe component and the receiver component.
- **Fully Specified**: These components have well defined complex transfer functions. The fully specified components are the fiber component, the fused biconical coupler component, the Fabry-Perot filter component and the Mach-Zehnder component.
• *Partially Specified:* These components are not well defined in terms of having an accurate or complete complex transfer function. The partially specified components are the star coupler, the ASE filter, the amplifier, the wavelength division multiplexer and the wavelength router.
4.1 Transmitter

Transmitters are the only AON component that can spontaneously generate optical signals. Transmitters generate a pulse with a given shape, and transmit that pulse to another component in the AON. All transmitters share the following standard parameters:

- **source ID** is the source identification number of the transmitter. Each pulse generated holds the source identification number in its source field.
- **frequency** is the pulse carrier frequency in THz.
- **peak power** is the maximum intensity of the pulse.
- **$t_0$** is a parameter related to the width of the pulse in picoseconds. For a gaussian pulse, the Full Width at Half Maximum (FWHM) pulse width is equal to $1.763 \times t_0$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source ID</td>
<td>integer</td>
<td>0 (N/A)</td>
<td>Identification number of transmitter</td>
</tr>
<tr>
<td>frequency</td>
<td>double</td>
<td>192.0 (THz)</td>
<td>Carrier frequency of transmitted pulse</td>
</tr>
<tr>
<td>peak power ($P_0$)</td>
<td>double</td>
<td>0.1 (W)</td>
<td>Peak power of transmitted pulse</td>
</tr>
<tr>
<td>$t_0$ ($t_0$)</td>
<td>double</td>
<td>100 (ps)</td>
<td>Parameter of pulse width</td>
</tr>
</tbody>
</table>

There are currently two classes of transmitters, classified by the pulse shape generated. There is the gaussian transmitter class, and the hyperbolic-secant transmitter class.
The gaussian transmitter class generates gaussian (See Figure 4-2) and super-gaussian (See Figure 4-3) pulse shapes defined by the following equation [Agr, 61]:

$$A(t) = \sqrt{P_0} e^{-\frac{1 + jC(t/t_0)^{2m}}{2}}$$

- $m$ controls the degree of pulse sharpness. Higher values of $m$ sharpen the pulse edges, and cause the pulse to have a squarer shape. $m$ is one for a gaussian pulse.
- $C$ controls the linear chirp of the pulse. $C$ is zero for an unchirped pulse.

![Figure 4-2: Gaussian pulse amplitude ($m = 1$, $t_0 = 100$ ps, $P_0 = 0.1$ W).](image)

![Figure 4-3: Super-Gaussian pulse amplitude ($m = 3$, $t_0 = 100$ ps, $P_0 = 0.1$ W).](image)
Table 4-2: Additional Parameters for Gaussian Transmitter

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m (m)</td>
<td>integer</td>
<td>1 (N/A)</td>
<td>Degree of gaussian.</td>
</tr>
<tr>
<td>C (C)</td>
<td>double</td>
<td>0 (N/A)</td>
<td>Initial chirp of pulse.</td>
</tr>
</tbody>
</table>

The hyperbolic-secant transmitter class generates pulses with the hyperbolic-secant shape (See Figure 4-4). This shape is important because it is the shape of a soliton. The hyperbolic-secant shape is defined by the following equation [Agr, 59]:

$$A(t) = \sqrt{P_0} \text{sech} \left( \frac{t}{t_0} \right) e^{-\frac{JCt^2}{2t_0^2}}$$

- C controls the linear chirp of the pulse. C is zero for an unchirped pulse.

Figure 4-4: Hyperbolic-secant pulse amplitude ($m = 1$, $t_0 = 100$ ps, $P_0 = 0.1$ W).

Table 4-3: Additional Parameters for Hyperbolic Secant Transmitter

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (C)</td>
<td>double</td>
<td>0 (N/A)</td>
<td>Initial chirp of pulse.</td>
</tr>
</tbody>
</table>
Each transmitter class includes two transmitter models. For each class, there is a model that transmits a single pulse, and a model that transmits a finite pulse stream. The single pulse model for each class has the following additional attribute:

- *time* is the transmission time of the leading edge of the single pulse.

**Table 4-4: Additional Parameters for Single Pulse Transmitter**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>double</td>
<td>0 (ps)</td>
<td>Transmission time of the leading edge of the pulse.</td>
</tr>
</tbody>
</table>

The pulse stream model for each class has the following additional attributes which describe a finite sequence machine:

- *start time* is the time in picoseconds of the first transmission.
- *spacing* is the amount of time in picoseconds between pulse transmissions.
- *repeat* is a flag that when set indicates that the finite sequence should be repeated until the end of the simulation.
- *initial state* is the initial state of the finite sequence machine that generates the pulse stream.
- *pn connections* describes the connections in the machine.
- *state bits* is the number of state bits in the machine.

**Table 4-5: Additional Parameters for Single Pulse Transmitter**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start time</td>
<td>double</td>
<td>0 (ps)</td>
<td>Transmission time of the leading edge of the first bit.</td>
</tr>
<tr>
<td>spacing</td>
<td>double</td>
<td>400 (ps)</td>
<td>Time between transmission of bits.</td>
</tr>
</tbody>
</table>
Table 4-5: Additional Parameters for Single Pulse Transmitter

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
<td>integer</td>
<td>0 (N/A)</td>
<td>Flag indicating whether or not to repeat the finite sequence until the end of the simulation.</td>
</tr>
<tr>
<td>initial state</td>
<td>integer</td>
<td>1 (N/A)</td>
<td>Initial state of the Finite Sequence Machine.</td>
</tr>
<tr>
<td>pn connections</td>
<td>integer</td>
<td>3 (N/A)</td>
<td>Connections in the Finite Sequence Machine.</td>
</tr>
<tr>
<td>state bits</td>
<td>integer</td>
<td>4 (N/A)</td>
<td>Number of state bits in the Finite Sequence Machine.</td>
</tr>
</tbody>
</table>

A finite sequence machine generates a pseudo-random stream of bits. For example, the four-bit finite sequence machine shown below (See Figure 4-5) generates a $2^4 - 1 = 15$ bit long stream before repeating.

![Finite Sequence Machine Diagram]

Figure 4-5: Four-bit Finite Sequence Machine: given a non-zero initial state, this machine will generate all four bit sequences for a total sequence length of $2^4 - 1$ bits [Pet, 148]. This particular machine has an initial state equal to 1, and a pn connections parameter equal to 3 ($= 2^3 c_3 + 2^2 c_2 + 2^1 c_1 + 2^0 c_0$) because connections 0 and 1 are connected. In this machine, bit 3 in state n+1 is equal to the exclusive or of the connected bits.
4.2 Optical Fiber

Optical fibers transmit pulses over distances in the AON Model Suite. Optical fibers receive pulse and noise data at an input port, transform that data, and after a delay, send the data out to an output port.

By default, the optical fiber model in the AON Model Suite models a single mode optical fiber with a core area of approximately 65 $\mu m^2$, and a zero dispersion wavelength of 1.33 $\mu m$.

The optical fiber model takes into account both linear and non-linear optical phenomena. The following effects are modeled:

- *Attenuation* is an effect that results in diminished pulse and noise power as a pulse or noise travels along a fiber.

- *propagation delay* is the delay a pulse or noise experiences traveling along the fiber.

- *dispersion* is an effect resulting from the varying value of the index of refraction of the fiber experienced by different wavelengths of light. This effect can alter a pulse's width and peak power.

- *Self Phase Modulation* (SPM) is a non-linear effect that results from a pulse's intensity modulating the phase of the pulse. SPM results in generation of new spectral components to the pulse.
• **Cross Phase Modulation** (XPM) is a non-linear effect that results from the intensity of a different pulse modulating the phase of the pulse. XPM results in the generation of new spectral components to the pulse.

• **Stimulated Raman Scattering** (SRS) is a non-linear effect that results in the transfer- ence of power from a high frequency pulse to a low frequency pulse.

### 4.2.1 Fiber Parameters

Optical fibers in the AON Model Suite are defined by a number of parameters (See Table 4-6).

#### Table 4-6: Optical Fiber Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length ((L))</td>
<td>double</td>
<td>100 ((km))</td>
<td>Length of optical fiber.</td>
</tr>
<tr>
<td>freq1 ((f_1))</td>
<td>double</td>
<td>192.0 ((THz))</td>
<td>First reference frequency.</td>
</tr>
<tr>
<td>freq2 ((f_2))</td>
<td>double</td>
<td>225.0 ((THz))</td>
<td>Second reference frequency.</td>
</tr>
<tr>
<td>B1 at freq1 ((\beta_{1,1}))</td>
<td>double</td>
<td>(4875 \frac{ps}{km})</td>
<td>First term of dispersion relationship at (f_1) THz.</td>
</tr>
<tr>
<td>B1 at freq2 ((\beta_{1,2}))</td>
<td>double</td>
<td>(4872 \frac{ps}{km})</td>
<td>First term of dispersion relationship at (f_2) THz.</td>
</tr>
<tr>
<td>B2 at freq1 ((\beta_{2,1}))</td>
<td>double</td>
<td>(-20 \frac{ps^2}{km})</td>
<td>Second term of dispersion relationship at (f_1).</td>
</tr>
<tr>
<td>B2 at freq2 ((\beta_{2,2}))</td>
<td>double</td>
<td>(0 \frac{ps^2}{km})</td>
<td>Second term of dispersion relationship at (f_2).</td>
</tr>
<tr>
<td>B3 ((\beta_3))</td>
<td>double</td>
<td>(0 \frac{ps^3}{km})</td>
<td>Third term of dispersion relationship.</td>
</tr>
<tr>
<td>alpha ((\alpha))</td>
<td>double</td>
<td>0.2 ((dB/km))</td>
<td>Attenuation per km.</td>
</tr>
<tr>
<td>A eff ((A_{eff}))</td>
<td>double</td>
<td>65 ((\mu m^2))</td>
<td>Effective area of fiber core.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Default (Units)</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>-----------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>n2 ((n_2))</td>
<td>double</td>
<td>(3.2 \times 10^{-16}) cm(^2)</td>
<td>The non-linear index coefficient.</td>
</tr>
<tr>
<td>T Raman ((T_R))</td>
<td>double</td>
<td>0.005 (ps)</td>
<td>The Raman gain time coefficient.</td>
</tr>
<tr>
<td>granularity</td>
<td>double</td>
<td>1 ((N/A))</td>
<td>Iterations of the split step Fourier method per length scale.</td>
</tr>
<tr>
<td>Grmax ((g_{Rmax}))</td>
<td>double</td>
<td>(10^{-16}) km(^{-1})</td>
<td>Maximum Raman gain.</td>
</tr>
<tr>
<td>Frmax ((\Delta f_{Rmax}))</td>
<td>double</td>
<td>12 (THz)</td>
<td>Width of linear section of Raman gain spectrum.</td>
</tr>
</tbody>
</table>

The following parameters are derived from these parameters:

- \(\beta_1(f_{carrier})\) or \(\beta_{1,f}\) is the value of the first term of the dispersion relationship such that:

\[
\beta_{1,f} = \beta_1(f_{carrier}) = \frac{\beta_{1,2} - \beta_{1,1}}{f_2 - f_1} (f_{carrier} - f_1)
\]

- \(v_g(f_{carrier})\) is the group velocity of signal or noise power as a function of frequency.

\[
v_g = v_g(f_{carrier}) = \frac{1}{\beta_1(f_{carrier})}
\]

- \(\gamma\) is the non-linearity coefficient [Agr, 40]:

\[
\gamma(f_{carrier}) = \frac{n_2 2\pi f_{carrier}}{cA_{eff}}
\]
4.2.2 Propagation Delay
Propagation delay of a pulse or of noise power is a function of the carrier frequency of the pulse or the frequency of the noise band. The propagation delay as a function of frequency is:

\[ D(f_{\text{carrier}}) = \frac{L}{v_g} = L\beta_1 \text{ where } \beta_1 = \frac{1}{v_g} \]

4.2.3 Split-Step Fourier Method
The fiber model utilizes a method called the split step fourier method [Agr, 44] to propagate pulses. The split step Fourier method is a method used to numerically approximates the simultaneous effects of both the linear and non-linear effects of the fiber. The split step Fourier method essentially approximates the simultaneous effects of the linear and non-linear effects of the fiber by assuming that over a short distance, the linear and non-linear effects can be assumed to act independently of each other [Agr, 44]. It is named the split step Fourier method because it performs the linear effects in the Fourier domain, and the non-linear effects in the time domain.

The length scales over which the fiber model propagates the pulse depend on the peak power and width of the pulse. There is a length scale associated with dispersion, and a length scale associated with the non-linear effects. Essentially, the fiber is chopped up into sections according to these length scales. The dispersion length is given by [Agr, 52]:

\[ L_D = \frac{T_0^2}{|\beta_2|} \]

where \( T_0 \) is the current FWHM width of the pulse. The non-linear length is given by [Agr, 52]:

\[ L_{NL} = \frac{1}{\gamma P_0} \]
where $P_0$ is the current peak power of the pulse.

### 4.2.4 Linear Effects
The linear effects are modeled by the following equation [Agr, 45]:

$$A(z+h, T) = e^{h\hat{D}} A(z, T)$$

where

$$\hat{D} = -\frac{i}{2} \beta_2 \frac{\delta^2}{\delta T^2} + \frac{1}{6} \beta_3 \frac{\delta^3}{\delta T^3} - \frac{\alpha}{2}$$

where [Agr, 42]

$$T = t - \frac{z}{v_g} = t - \beta_1 z$$

describes a frame of reference moving with the pulse at the group velocity of the pulse.

This equation is easily solved in the Fourier domain using the following relationship [Agr, 45]:

$$e^{h\hat{D}} B(z, T) = \{F^{-1} e^{h\hat{D}(i\omega)} F\} B(z, \omega)$$

where $F$ indicates the Fourier transform operation.

### 4.2.5 Non-Linear effects caused by the Pulse
The non-linear effects that are modeled within the pulse are SPM, SRS and the self-steepening effect that results from the slowly varying non-linear polarization [Agr, 42]. The non-linear effects are modeled by the following equation [Agr, 45]:

$$A(z+h, T) = e^{h\hat{N}} A(z, T)$$

where

$$\hat{N} = j\gamma \left( |A|^2 + \frac{2j}{\omega_0 A} \frac{\delta}{\delta T} \left( |A|^2 A \right) - T_R \frac{\delta |A|^2}{\delta T} \right)$$

Again, $T$ represents a frame of reference moving at the group velocity of the pulse.
4.2.6 Non-Linear effects caused by Pulses at other Frequencies

SRS and XPM effects are modeled between pulses. A walkoff length scale is required [Agr, 225]:

\[ L_w = \frac{T_0}{|\beta_{1,i} - \beta_{1,j}|} \]

where \( \beta_{1,i} \) is equal to \( \beta_1 \) at the first pulse carrier frequency, and \( \beta_{1,j} \) is equal to \( \beta_1 \) at the second pulse carrier frequency.

This equation can be used to determine the interaction length between overlapping samples in different pulses:

\[ h = L_{interaction} = \frac{AonI\_Duration/2^\gamma}{|\beta_{1,i} - \beta_{1,j}|} \]

where \( AonI\_Duration/2^\gamma \) is the time between pulse samples. As pulse samples pass through each other they interact due to XPM and SRS according to the following equation:

\[ S_i(z + h, T) = \prod_{j \neq i} \left( e^{h |\gamma| \gamma^2 + g_R(\Delta f_{i,j}) |S_j(z, T)|^2} \right) S_i(z, T) \]

where \( S_1 \) is the amplitude of the sample of pulse 1 and \( S_2 \) is the amplitude of the sample of pulse 2. The SRS gain as a function of carrier frequency difference, \( g_R(\Delta f_{i,j}) \), is described by:

\[ g_R(\Delta f_{i,j}) = \begin{cases} \frac{\Delta f_{i,j}}{\Delta f_{Rmax}} & \text{for } \Delta f_{i,j} > 0 \\ 0 & \text{for } \Delta f_{i,j} \leq 0 \end{cases} \]

where \( \Delta f_{i,j} = f_j - f_i \).
where $g_{R_{max}}$ is the maximum SRS gain, $\Delta f_{i,j}$ is the difference in the frequency of the carrier signal, and $\Delta f_{R_{max}}$ is the width of the linear part of the Raman gain spectrum.
### 4.3 Fused Biconical Coupler

![Diagram of Fused Biconical Coupler](image)

**Figure 4-7: AON Fused Biconical Coupler icon and port layout.**

The fused biconical coupler is a linear device in which two fibers are fused in such a way as to produce coupling between them such that pulse energy from one fiber can be transferred to the other. A pulse coming into the fused biconical coupler is split into two pulses, one on each fiber. Each pulse is the product in the Fourier domain of the original pulse and a transfer function. This transfer function is determined by three parameters [Gre, 70]:

- $r$ The core radius of the coupling region.
- $Z$ The length of the coupling region
- $\Delta r$ The difference in core radii in the coupling region

#### Table 4-7: Fused Biconical Coupler Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core radius ($r$)</td>
<td>double</td>
<td>8 (\mu m)</td>
<td>The core radius of the coupling region.</td>
</tr>
<tr>
<td>Length ($Z$)</td>
<td>double</td>
<td>10000 (\mu m)</td>
<td>The length of the coupling region.</td>
</tr>
<tr>
<td>delta r ($\Delta r$)</td>
<td>double</td>
<td>0 (\mu m)</td>
<td>The difference in core radii in the coupling region.</td>
</tr>
<tr>
<td>Power Loss ($a$)</td>
<td>double</td>
<td>0 (dB)</td>
<td>Insertion power loss in dB.</td>
</tr>
<tr>
<td>Delay ($D$)</td>
<td>double</td>
<td>10 (ps)</td>
<td>Delay of FBC.</td>
</tr>
</tbody>
</table>

These three parameters in turn define [Gre, 70]:

51
• \( F^2 = \left[1 + \left(\frac{234r^3}{\lambda^3}\right)^2 (\frac{\Delta r}{r})^2 \right]^{-1} \) where \( \lambda = \frac{c}{f} \) The core diameter difference effect

• \( C = \frac{21\lambda^{5/2}}{r^{7/2}} \) The coupling effect

The operation of the component can be described by the following matrix operation for pulses [Gre, 70]:

\[
P_{\text{pulse}}^{\text{out}}(t) = \sqrt{10^{\frac{a(dB)}{10}}} \begin{bmatrix}
0 & 0 & \sqrt{1-\alpha} & j\sqrt{\alpha} \\
0 & 0 & j\sqrt{\alpha} & \sqrt{1-\alpha} \\
\sqrt{1-\alpha} & j\sqrt{\alpha} & 0 & 0 \\
j\sqrt{\alpha} & \sqrt{1-\alpha} & 0 & 0 \\
\end{bmatrix} P_{\text{pulse}}^{\text{in}}(t-D)
\]

where

\[
\alpha^2 = F^2 \sin^2 \left( \frac{CZ}{F} \right)
\]

The operation of the component can be described by the following matrix operation for noise:

\[
P_{\text{noise}}^{\text{out}}(t) = 10^{\frac{a(dB)}{10}} \begin{bmatrix}
0 & 0 & 1-\alpha & \alpha \\
0 & 0 & \alpha & 1-\alpha \\
1-\alpha & \alpha & 0 & 0 \\
\alpha & 1-\alpha & 0 & 0 \\
\end{bmatrix} P_{\text{noise}}^{\text{in}}(t-D)
\]
The amplitude of the complex transfer function of the FBC is shown below (See Figure 4-8).

**Figure 4-8: Amplitude of H(f) of Fused Biconical Coupler. (Δr = 0, r = 8, Z=8716.0)**
4.4 Star Coupler

![Star Coupler Diagram]

Figure 4-9: AON Star Coupler icon and port layout.

The star coupler is modeled as a linear device that consists of a number of coupling devices that yield even power distribution over the $N$ outputs. The star coupler model is a partially specified model in that its transfer function over-simplifies the effects of a star coupler. A pulse or noise packet entering a port of the star coupler is essentially attenuated due to insertion power loss, copied $N-1$ times, and $N$ copies, each with power $1/N$ are sent out to each of the $N$ output ports of the device. The transfer function is:

$$H(f) = \sqrt{\frac{a \text{ (dB)}}{10}}$$

The operation of the component is modeled by the following matrix operation for pulses:

$$P_{\text{pulse}}^{\text{out}} = \begin{bmatrix}
0 & \cdots & 0 & H(f) & \cdots & H(f) \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & H(f) & \cdots & H(f) \\
H(f) & \cdots & H(f) & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
H(f) & \cdots & H(f) & 0 & \cdots & 0
\end{bmatrix} P_{\text{pulse}}^{\text{in}}$$

54
The operation of the component is modeled by the following matrix operation for noise:

\[
P_{\text{noise}}^{\text{out}} = \begin{bmatrix}
0 & \cdots & 0 & |H(f)|^2 & \cdots & |H(f)|^2 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & 0 & |H(f)|^2 & \cdots & |H(f)|^2 \\
|H(f)|^2 & \cdots & |H(f)|^2 & 0 & \cdots & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
|H(f)|^2 & \cdots & |H(f)|^2 & 0 & \cdots & 0
\end{bmatrix} \cdot \begin{bmatrix}
P_{\text{noise}}^{\text{in}}
\end{bmatrix}
\]

Table 4-8: Star Coupler Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Loss (a)</td>
<td>double</td>
<td>0 (dB)</td>
<td>Power loss</td>
</tr>
<tr>
<td>N</td>
<td>integer</td>
<td>2 (N/A)</td>
<td>Number of inputs and outputs. The device has a total of 2N ports.</td>
</tr>
</tbody>
</table>
4.5 Optical Amplifier

![Diagram of optical amplifier](image)

Figure 4-10: AON Amplifier icon and port layout. The amplifier is a unidirectional device. Incoming packets on port 1 are discarded. The amplifier model does not use the port 0 output.

The optical amplifier is modeled after an Erbium doped fiber amplifier (EDF Amplifier) with an optical isolator on one end. The optical isolator effectively makes the amplifier a unidirectional device. An EDF amplifier is a non-linear device. The gain characteristics and noise output vary with the average incident power over a time period. Gain is calculated as such [Cha, 64]:

\[
G(W_{in}) = \frac{G_0}{W_{in}} \left(1 + \frac{E_{total}}{P_{sat}}\right)^{-\frac{t-t_{arrival}}{\tau}}
\]

where \(G_0 = 10^{\frac{G_0(dB)}{10}}\) is the gain when \(W_{in}\) is zero, and \(P_{sat}\) is the saturation point. \(E_{total}\) is the energy of the pulse in picojoules, \(t_{arrival}\) is the arrival time of the pulse, and \(\tau\) is an EDF amplifier specific time constant that describes the relaxation period of the amplifier.

Pulses and noise going through the EDF amplifier experience a gain equal to \(G(W_{in})\). Additionally, the EDF amplifier generates spontaneous noise power. This generated noise power is given by the following relation [Cha, 64]:

\[
P_{ASE}(G(W_{in}), f) = G(W_{in}) hf \Delta f
\]
where \( \delta f = \frac{A_{onI\_High\_Freq} - A_{onI\_Low\_Freq}}{A_{onI\_N\_Segment}} \) is the optical bandwidth of the noise band, \( h = 6.626 \times 10^{-34} \) (Joules \( \cdot \) seconds) is Planck's constant, \( f \) is the center frequency of the noise band, \( N = 10^{10} \) is the EDF amplifier noise figure and \( P_{ASE} \) is the ASE noise power.

Table 4-9: Optical Amplifier Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain ((G_0))</td>
<td>double</td>
<td>10 (dB)</td>
<td>Base power gain</td>
</tr>
<tr>
<td>Saturation Power ((P_{sat}))</td>
<td>double</td>
<td>(1 \times 10^{-6}) (W)</td>
<td>Input Saturation Power</td>
</tr>
<tr>
<td>Noise Coef. ((N))</td>
<td>double</td>
<td>5 (dB)</td>
<td>Noise Coefficient of amplifier</td>
</tr>
<tr>
<td>Relax Time ((\tau))</td>
<td>double</td>
<td>(10^9) (ps)</td>
<td>Relaxation time of amplifier</td>
</tr>
<tr>
<td>Delay</td>
<td>double</td>
<td>(1.5 \times 10^5) (ps)</td>
<td>Delay of amplifier component</td>
</tr>
<tr>
<td>delta noise percent</td>
<td>double</td>
<td>1 (percent)</td>
<td>The noise level is updated every time it changes at least by delta noise percent.</td>
</tr>
</tbody>
</table>
4.6 ASE Filter

The ASE filter component is modeled by an ideal multiple-passband filter. The ASE filter model is a partially specified model in that its transfer function over-simplifies the effects of a filter. The ASE filter is defined by its free spectral range (FSR), its insertion loss attenuation (a), its passband bandwidth (W) and its delay (D). In the passbands, the signal is attenuated by $a$ dB. Outside of the passband, the signal is not passed at all.

Table 4-10: ASE Filter Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attenuation (a)</td>
<td>double</td>
<td>1 (dB)</td>
<td>Insertion Power Loss.</td>
</tr>
<tr>
<td>FSR</td>
<td>double</td>
<td>0.05 (THz)</td>
<td>Free Spectral Range.</td>
</tr>
<tr>
<td>Bandwidth (W)</td>
<td>double</td>
<td>0.01 (THz)</td>
<td>3 dB bandwidth of filter.</td>
</tr>
<tr>
<td>Delay (D)</td>
<td>double</td>
<td>10 (ps)</td>
<td>Delay of filter.</td>
</tr>
</tbody>
</table>

As a pulse traverses the ASE filter, it is multiplied in the Fourier domain by the following complex transfer function:

$$ H(f) = \begin{cases} \sqrt[10]{\frac{a\,(dB)}{10}}, & \text{if } f \text{ in passband} \\ 0, & \text{if } f \text{ not in passband} \end{cases} $$

where

$$ f \text{ in passband} = -\frac{W}{2} + kFSR < f < \frac{W}{2} + kFSR \text{ for any } k $$

Figure 4-11: AON ASE Filter icon
The operation of the component can be described by the following matrix operation for pulses:

\[
P_{\text{pulse}}^{\text{out}}(t) = \begin{bmatrix} 0 & H(f) \\ H(f) & 0 \end{bmatrix} P_{\text{pulse}}^{\text{in}}(t-D)
\]

The operation of the component can be described by the following matrix operation for noise:

\[
P_{\text{noise}}^{\text{out}}(t) = \begin{bmatrix} 0 & |H(f)|^2 \\ |H(f)|^2 & 0 \end{bmatrix} P_{\text{noise}}^{\text{in}}(t-D)
\]

The complex transfer function of the ASE filter is periodic over the frequency spectrum with period \( FSR() \).

Figure 4-12: Amplitude of \( H(f) \) of ASE Filter. (\( FSR = 0.5 \) THz, \( a = 1 \) dB, \( W = 0.25 \) THz)
4.7 Fiber Fabry-Perot Filter

The Fiber Fabry-Perot filter is a tunable filter used in All Optical Networks. The primary attribute of a Fiber Fabry-Perot filter is its finesse:

\[ F = \frac{\pi \sqrt{R}}{1 - R} \]

where \( R \) is the power reflectivity of the mirrors of the Fiber Fabry-Perot filter cavity. Finesse is a measure of the sharpness of the Fiber Fabry-Perot’s etalons.

The free spectral range of a Fiber Fabry-Perot filter is:

\[ FSR = \frac{1}{2\tau} \text{ where } \tau = \frac{nx}{c} \]

where \( \tau \) is the one-way propagation time through the filter.

The maximum transference of a Fiber Fabry-Perot filter is:

\[ T(f)_{\text{max}} = \left[ 1 - \frac{A}{1 - R} \right]^2 \]

where \( A \) is the power attenuation of the mirrors of the Fiber Fabry-Perot filter cavity.
From $F$, $FSR$ and $T(f)_{max}$ the three basic Fiber Fabry-Perot parameters that describe the transfer function, $R$, $\tau$ and $A$, can be found.

As a pulse traverses the Fiber Fabry-Perot filter, it is multiplied in the Fourier domain by the following complex transfer function:

$$H(f) = \frac{1 - A - R}{1 - Re^{-j4\pi f_0}} e^{-j2\pi \tau}$$

This corresponds to the following matrix operation for pulses:

$$\begin{bmatrix}
0 & H(f) \\
H(f) & 0
\end{bmatrix} \begin{bmatrix}
\text{pulse}(t) \\
\text{pulse}(t-D)
\end{bmatrix}$$

The operation of the component can be described by the following matrix operation for noise:

$$\begin{bmatrix}
0 & |H(f)|^2 \\
|H(f)|^2 & 0
\end{bmatrix} \begin{bmatrix}
\text{noise}(t) \\
\text{noise}(t-D)
\end{bmatrix}$$

Table 4-11: Fiber Fabry-Perot Filter

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finesse ($F$)</td>
<td>double</td>
<td>20 (N/A)</td>
<td>Finesse</td>
</tr>
<tr>
<td>FSR ($FSR$)</td>
<td>double</td>
<td>0.05 (THz)</td>
<td>Free Spectral Range</td>
</tr>
<tr>
<td>$T_{max}$ ($T(f)_{max}$)</td>
<td>double</td>
<td>0.9 (N/A)</td>
<td>Maximum Transference</td>
</tr>
<tr>
<td>Delay ($D$)</td>
<td>double</td>
<td>10 (ps)</td>
<td>Delay of filter</td>
</tr>
</tbody>
</table>
The complex transfer function of the Fabry-Perot filter is periodic over the frequency spectrum with period $FSR$ (See Table 4-14, See Figure 4-15).

**Figure 4-14:** Amplitude of $H(f)$ of Fabry-Perot Filter for three different values of finesse. $FSR = 0.5 \, THz$, $T(f)_{max} = 0.9$.

**Figure 4-15:** Phase of $H(f)$ of Fabry-Perot Filter for three different values of finesse. $FSR = 0.5 \, THz$, $T(f)_{max} = 0.9$. 
4.8 Mach-Zehnder Filter

The Mach-Zehnder filter is a four-port device that filters light by taking advantage of interference effects due to interfering light with itself after splitting the light and having it travel over two slightly different path lengths.

As a pulse or noise packet passes through the Mach-Zehnder filter, it is copied. The original copy of a pulse is multiplied in the Fourier domain by the following complex transfer function:

$$H_{acr}(f) = \frac{1}{2} \left(e^{-j2\pi f\tau} - 1\right)$$

and sent off to the port across from the one that it came in through. That is, if a pulse comes in through port zero, it will leave through port two. The pulse copy is multiplied in the Fourier domain by the following complex transfer function:

$$H_{opp}(f) = \frac{1}{2j} \left(e^{-j2\pi f\tau} + 1\right)$$

and sent off to the port opposite from the one that it came in through. That is, if a packet comes in through port zero, it will leave through port three.
Noise packets are dealt with in nearly the same way. The noise slices in the original packet are multiplied by:

$$|H_{acr}(f)|^2$$

and sent to the port across from the one that it came in through. The noise slices in the noise copy are multiplied by:

$$|H_{opp}(f)|^2$$

and send to the port opposite from the one that it came in through.

**Table 4-12: Mach-Zehnder Filter**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>double</td>
<td>0.5 (THz)</td>
<td>Free Spectral Range.</td>
</tr>
<tr>
<td>Delay (D)</td>
<td>double</td>
<td>10 (ps)</td>
<td>Delay of filter component.</td>
</tr>
</tbody>
</table>

The matrix describing the operation of the MZF on pulses is:

$$
P_{pulse}^{out}(t) = \begin{bmatrix}
  0 & 0 & H_{acr}(f) & H_{opp}(f) \\
  0 & 0 & H_{opp}(f) & H_{acr}(f) \\
  H_{acr}(f) & H_{opp}(f) & 0 & 0 \\
  H_{opp}(f) & H_{acr}(f) & 0 & 0
\end{bmatrix}
P_{pulse}^{in}(t-D)$$
The matrix describing the operation of the MZF on noise is:

\[
P_{\text{noise}}^{out}(t) = \begin{bmatrix}
0 & 0 & |H_{acr}(f)|^2 & |H_{opp}(f)|^2 \\
0 & 0 & |H_{opp}(f)|^2 & |H_{acr}(f)|^2 \\
|H_{acr}(f)|^2 & |H_{opp}(f)|^2 & 0 & 0 \\
|H_{opp}(f)|^2 & |H_{acr}(f)|^2 & 0 & 0
\end{bmatrix} P_{\text{noise}}^{in}(t-D)
\]

The complex transfer function of the Fabry-Perot filter is periodic over the frequency spectrum with period \( FSR \) (See Figure 4-8, See Figure 4-15).

**Figure 4-17: Amplitude of \( H_{acr}(f) \) and \( H_{opp}(f) \) of Mach-Zehnder Filter for FSR = 0.5 THz.**

**Figure 4-18: Phase of \( H_{acr}(f) \) and \( H_{opp}(f) \) of Mach-Zehnder Filter for FSR = 0.5 THz.**
4.9 Wavelength Division (De)Multiplexer

The wavelength division (de)multiplexer is a device that can be used to multiplex or demultiplex an optical signal based on its frequency. The wavelength division multiplexer is a partially specified model. Phase shift is not taken into account, and the physical processes behind the device are modeled only qualitatively. Let

\[ H_1(f) = \sqrt{\frac{a(dB)}{10}} \sin\left(\frac{2\pi f}{FSR}\right) \]

and let

\[ H_2(f) = \sqrt{\frac{a(dB)}{10}} \cos\left(\frac{2\pi f}{FSR}\right) \]

The operation of the component is modeled by the following matrix operation for pulses:

\[
P_{\text{out}}^{\text{pulse}}(t) = \begin{bmatrix} 0 & 0 & H_1(f) \\ 0 & 0 & H_2(f) \\ H_1(f) & H_2(f) & 0 \end{bmatrix} P_{\text{in}}^{\text{pulse}}(t-D)
\]

The operation of the component is modeled by the following matrix operation for noise:

\[
P_{\text{out}}^{\text{noise}}(t) = \begin{bmatrix} 0 & 0 & |H_1(f)|^2 \\ 0 & 0 & |H_2(f)|^2 \\ |H_1(f)|^2 & |H_2(f)|^2 & 0 \end{bmatrix} P_{\text{in}}^{\text{noise}}(t-D)
\]
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>double</td>
<td>32 (THz)</td>
<td>Free Spectral Range</td>
</tr>
<tr>
<td>Delay (D)</td>
<td>double</td>
<td>10 (ps)</td>
<td>Delay of WDM.</td>
</tr>
<tr>
<td>Attenuation (a)</td>
<td>double</td>
<td>0 (dB)</td>
<td>Insertion power loss</td>
</tr>
</tbody>
</table>
4.10 Wavelength Router

The wavelength router is a device that routes different wavelengths to different output ports using a diffraction grating. The wavelength router is a partially specified model. Phase shift is not taken into account, and the physical processes are not modeled directly. Essentially, the diffraction grating directs certain wavelengths towards certain ports. The following parameters define a wavelength router in the AON Model Suite.

The operation of the wavelength router is based on the following transfer function:

\[ H_i(f) = \sqrt{a} \left[ \frac{1}{N} \left( \frac{1}{\sin \left( \frac{f - i \delta f}{FSR} \right)} \right)^2 + k \right]^{-1} \]

where

\[ \delta f = \frac{FSR}{N} \]

is the frequency range between adjacent channels,

\[ a = 10^{\frac{a(dB)}{10}} \]

is the attenuation, and

\[ k = 10^{\frac{k(dB)}{10}} \]

is the extinction ratio.

Figure 4-20: AON Router icon and port layout
The operation of the component is modeled by the following matrix operation for pulses:

\[
\begin{bmatrix}
0 & 0 & 0 & \ldots & 0 & H_0(f) & H_{N-1}(f) & H_{N-2}(f) & \ldots & H_1(f) \\
0 & 0 & 0 & \ldots & 0 & H_1(f) & H_0(f) & H_{N-1}(f) & \ldots & H_2(f) \\
0 & 0 & 0 & \ldots & 0 & H_2(f) & H_1(f) & H_0(f) & \ldots & H_3(f) \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & 0 & 0 & \ldots & 0 & H_{N-1}(f) & H_{N-2}(f) & H_{N-3}(f) & \ldots & H_0(f) \\
\end{bmatrix}
\]

\[
\mathbf{p}_{\text{pulse}}^{\text{out}}(t) = \begin{bmatrix}
0 & 0 & 0 & \ldots & 0 & H_0(f) & H_{N-1}(f) & H_{N-2}(f) & \ldots & H_1(f) \\
0 & 0 & 0 & \ldots & 0 & H_1(f) & H_0(f) & H_{N-1}(f) & \ldots & H_2(f) \\
0 & 0 & 0 & \ldots & 0 & H_2(f) & H_1(f) & H_0(f) & \ldots & H_3(f) \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & 0 & 0 & \ldots & 0 & H_{N-1}(f) & H_{N-2}(f) & H_{N-3}(f) & \ldots & H_0(f) \\
\end{bmatrix}
\]

The operation of the component is modeled by the following matrix operation for noise:

\[
\begin{bmatrix}
0 & 0 & 0 & \ldots & 0 & |H_1(f)|^2 & |H_N(f)|^2 & |H_{N-1}(f)|^2 & \ldots & |H_2(f)|^2 \\
0 & 0 & 0 & \ldots & 0 & |H_2(f)|^2 & |H_1(f)|^2 & |H_N(f)|^2 & \ldots & |H_3(f)|^2 \\
0 & 0 & 0 & \ldots & 0 & |H_3(f)|^2 & |H_2(f)|^2 & |H_1(f)|^2 & \ldots & |H_4(f)|^2 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & 0 & 0 & \ldots & 0 & |H_N(f)|^2 & |H_{N-1}(f)|^2 & |H_{N-2}(f)|^2 & \ldots & |H_1(f)|^2 \\
\end{bmatrix}
\]

\[
\mathbf{p}_{\text{noise}}^{\text{out}} = \begin{bmatrix}
|H_0(f)|^2 & |H_{N-1}(f)|^2 & |H_{N-2}(f)|^2 & \ldots & |H_1(f)|^2 \\
|H_1(f)|^2 & |H_0(f)|^2 & |H_{N-1}(f)|^2 & \ldots & |H_2(f)|^2 \\
|H_2(f)|^2 & |H_1(f)|^2 & |H_0(f)|^2 & \ldots & |H_3(f)|^2 \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
|H_{N-1}(f)|^2 & |H_{N-2}(f)|^2 & |H_{N-3}(f)|^2 & \ldots & |H_0(f)|^2 \\
\end{bmatrix}
\]
The transfer function of the wavelength router is periodic over the frequency spectrum with period $FSR$ (See Figure 4-21).

Figure 4-21: Amplitude of $H(f)$ of the wavelength router for two values of $i$. ($FSR = 1.0 \text{ THz}, N=4, k=14.7 \text{ dB}, a=0 \text{ dB}$)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attenuation ($a$)</td>
<td>double</td>
<td>0 (dB)</td>
<td>Insertion power loss.</td>
</tr>
<tr>
<td>$N$</td>
<td>integer</td>
<td>2 (N/A)</td>
<td>Number of input ports and number of output ports. Total number of ports is $2N$.</td>
</tr>
<tr>
<td>$FSR$</td>
<td>double</td>
<td>0.5 (THz)</td>
<td>Free spectral range.</td>
</tr>
<tr>
<td>Extinction Ratio ($k$)</td>
<td>double</td>
<td>16 (dB)</td>
<td>Extinction ratio of router.</td>
</tr>
<tr>
<td>Delay ($D$)</td>
<td>double</td>
<td>10 (ps)</td>
<td>Delay of router.</td>
</tr>
</tbody>
</table>
4.11 Probe

\[ P_0^{\text{in}} \xrightarrow{\text{P}} P_1^{\text{out}} \]

**Figure 4-22:** *AON Probe icon and port layout.*

The AON Probe component is a passive component that allows for the collection of information about the pulses and noise that go through it. The AON Probe writes these values to OPNET local statistics which can be probed using the OPNET Probe Editor and analyzed using the OPNET Analysis Tool. The AON Probe writes out the following information:

- *Received power* is a measure of the instantaneous power going through the probe. This power can either be calculated assuming all pulses are *coherent*, or that the pulses are incoherent. If the pulses are considered coherent, the received power is equal to the square of the sum of the complex amplitudes. Otherwise, the received power is equal to the sum of the squares of the complex amplitudes. Received power is written out to the AON Probe's local `outstat[0]`.

- *Noise power* is a measure of the instantaneous noise power going through the probe. Noise power is written out to the AON Probe's local `outstat[1]`.

- *Eye power* is a statistic that generates an "eye chart" based on the instantaneous power going through the probe. An eye chart is an analysis tool that shows the ability of a receiver to convert optical signals into digital signals. The instantaneous power at time \( t \) is mapped onto the eye time, \( t_{\text{eye}} \), such that

\[
    t_{\text{eye}} = (t - t_0) \mod \text{eye width}
\]
Essentially, the instantaneous power is "wrapped" around the eye chart. Eye power is written out to the AON Probe's local `outstat[2]`. For best results, the `eye width` parameter should be set equal to the `spacing` parameter of the transmitter whose signal is of interest.

- **Pulse amplitude and phase** are two statistics that show the amplitude and phase of a single pulse. Each pulse has its amplitude and phase written out to a unique set of `outstats`. The amplitude and phase of pulse $n$ are written out to `outstat[4n+3]` and `outstat[4n+4]` respectively.

- **Pulse Fourier Transform amplitude and phase** are two statistics that show the amplitude and phase of the Fourier transform of a single pulse. Each pulse has the amplitude and phase of its Fourier transform written out to a unique set of `outstats`. The amplitude and phase of the Fourier transform of pulse $n$ are written out to `outstat[4n+5]` and `outstat[4n+6]` respectively.

<table>
<thead>
<tr>
<th>Table 4-15: Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>eye width</td>
</tr>
<tr>
<td>coherent</td>
</tr>
<tr>
<td>Signal ID</td>
</tr>
</tbody>
</table>
4.12 Receiver

\[
\begin{array}{c}
\text{Figure 4-23: AON Receiver icon. The receiver component model does not use the port 0 output.}
\end{array}
\]

The receiver acts exactly like a probe, except that it destroys any packets that enter it, instead of forwarding them.

Table 4-16: Receiver

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default (Units)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eye width</td>
<td>double</td>
<td>400 (ps)</td>
<td>Width of eye chart in picoseconds</td>
</tr>
<tr>
<td>coherent</td>
<td>integer</td>
<td>0 (N/A)</td>
<td>If set, the receiver performs interference on complex envelopes, otherwise, the separate powers are summed.</td>
</tr>
<tr>
<td>Signal ID</td>
<td>integer</td>
<td>0 (N/A)</td>
<td>Source ID of signal power; other received power is noise for SNR calculations</td>
</tr>
</tbody>
</table>
Chapter 5: Simulation Results

The component models can be classified into three different groups:

- **Essential**: These models are critical to the operation of the simulation and are well defined. Essential models include the transmitter models, the probe model and the receiver model. These models are well tested, and the validity of all other models depends on the validity of these models. Simulation results specific to these models are not explicitly shown in this chapter. All simulations depend upon the accuracy of these models, and so simulation results for these models are implicit in all simulation results.

- **Fully specified**: These models are fully specified in terms of having a well defined complex transfer function, or other mode of operation. The Fully specified models include the fiber model, the fused biconical coupler model, the Fabry-Perot filter model and the Mach-Zehnder filter model. These models are tested explicitly in this chapter and in-depth results and analysis for each model are presented individually.

- **Partially specified**: These models are not well defined in terms of having accurate complex transfer functions. The Partially specified models include the star coupler model, the ASE filter model, the amplifier model, the wavelength division (de) multiplexer model and the wavelength router model. These models have been tested only qualitatively, and no results or analysis is presented in this chapter.

For all of the results in this chapter the default component parameters are used, except as otherwise indicated.
5.1 Fiber Model

5.1.1 Dispersion in Linear Regime
This test is designed to show the validity of the fiber model in the linear regime of a fiber. In order to stay in the linear regime the pulse traveling over the fiber links must be of very low power. The network being tested (See Figure 5-1) consists of a transmitter, three fiber links and a receiver. Additionally, there are probes for the collection of statistics. Links are not duplex in this model, disabling reflections, as the object of the model is to study the effects of dispersion on pulse receivability in the absence of other effects. This network shows the effects of dispersion in both the normal and anomalous regimes of optical fiber. The transmitter transmits a bit stream of gaussian pulses with an initial $t_0$ of 100 ps, and $P_0$ of 0.001 W on a carrier frequency of 192.0 THz. After 50 kilometers of fiber with

$$\beta_2 (192.0THz) = -10\frac{ps^2}{km}$$

the pulse has been flattened due to dispersion (See Figure 5-2). The pulse is further flattened after another 50 kilometers of fiber with

$$\beta_2 (192.0THz) = -10\frac{ps^2}{km}$$

This flattened pulse then goes through 50 kilometers of fiber with $\beta_2 (192.0THz) = 20\frac{ps^2}{km}$ and regains its shape.

Figure 5-1: Network and Node Level descriptions of test network. The links in this network are simplex. This is because the object of the experiment is to study the effects of dispersion on receivability in the absence of other effects.

Dispersion has a profound effect on the receivability of a bit stream. This can be seen by looking at an “eye diagram.” By looking at the eye diagram one can determine what the received power threshold is for the bit stream. In this simulation, the is fully open coming out of the transmitter (See Figure 5-3). The eye contracts over the first 100 km of fiber.
(See Figure 5-4, See Figure 5-5), and then re-opens upon reconstruction (See Figure 5-6).

Figure 5-2: A pulse is flattened due to dispersion after going through sections of fiber with a positive group velocity dispersion coefficient ($\beta_2$). The flattened pulse is chirped by $100\text{km} \times 10\text{ps}^2/\text{km}$. The original pulse is reconstructed by going through a section of fiber that “unchirps” the pulse by inducing an equal and opposite amount of chirp.
Figure 5-3: The bit stream coming out of the transmitter. The eye is fully dilated, with a maximum opening of 1 mWatt. The signal can be received easily.

Figure 5-4: The bit stream after going through 50 km of dispersive fiber \((\beta_1(f) = -10\frac{ps^2}{km})\). The eye is still quite dilated, with a maximum opening of 0.43 mWatts. The signal can still be received.
Figure 5-5: The bit stream after going through 100 km of dispersive fiber

\( (\beta_1(f) = -10\frac{ps^2}{km} ) \). The eye is nearly shut, with a maximum opening of 75 \( \mu \)Watts. The signal can be received only with difficulty.

Figure 5-6: The bit stream after reconstruction. The eye is fully dilated, with a maximum opening of 0.95 mWatts. The signal can again be received easily.
5.1.2 Non-linearities at the Zero Dispersion Point

This test is designed to show the validity of the fiber model in the zero dispersion regime of a fiber. The network being tested (See Figure 5-7) consists of a transmitter, a fiber and a receiver. Additionally, there is a probe on the output of the transmitter in order to collect baseline data. The links in this model are not duplex as the object of the model is to examine the effects of Self Phase Modulation on the complex pulse envelope. This network show the effect of Self Phase Modulation at the zero dispersion point. Because we are interested in the pulse shape, the transmitter sends a single pulse across the fiber. This pulse is a gaussian \( m = 1 \) with an initial \( t_0 \) of 100 ps, and \( P_0 \) of 0.11 W on a carrier frequency of 192.0 THz. After 50 km of fiber with \( \beta_2 (192.0 \text{THz}) = \frac{0.6 \text{ps}^2}{\text{km}} \), the pulse has been altered by Self Phase Modulation. Self Phase Modulation does not alter the pulse amplitude (See Figure 5-8), but rather manipulates the pulse phase (See Figure 5-9). The change in the pulse phase has a profound effect on the spectral composition of the pulse (See Figure 5-10, See Figure 5-11). By altering the pulse phase, SPM generates new spectral components. While this does not directly alter the receivability of a signal, as the pulse shape doesn’t change, these spectral changes can cause problems by exacerbating dispersive effects and pushing pulse energy beyond filter passband limits (See Figure 5-13).
Figure 5-8: Pulse amplitude before and after traveling through the fiber. The pulse amplitude has not changed appreciably.

Figure 5-9: Pulse phase before and after traveling through the fiber. SPM has altered the pulse phase considerably.
Figure 5-10: Fourier Transform Amplitude of the pulse before and after traveling through the fiber. SPM has broadened the spectrum significantly.

Figure 5-11: Fourier Transform phase of the pulse before and after traveling through the fiber. SPM has had a profound effect.
5.2 Filters

5.2.1 Fabry-Perot Filter
This test is designed to show the validity of the Fabry-Perot filter model. The input to the

![Diagram of network for testing the Fabry-Perot filter]

Figure 5-12: Node level description of network for testing the 
Fabry-Perot filter. The pulse entering the filter is the same pulse 
generated in section 5.1.2, a gaussian chirped by SPM.

Fabry-Perot filter is the gaussian pulse chirped by SPM generated in section 5.1.2. The 
links in this model are not duplex as the object of the model is to examine the effects of the 
Fabry-Perot filter on the complex pulse envelope. The filter in this model has the following parameters:

- $FSR = 0.5 \text{ THz}$
- $Finesse = 10.0$
- $T_r(max) = 1.0$

The filter has a significant effect on the complex pulse envelope, as seen below (See Figure 5-13).
Figure 5-13: The pulse amplitude before and after going through the Fabry-Perot filter. Because the carrier frequency lies centered on a passband of the Fabry-Perot filter, more energy is lost in sections of the pulse where the spectral components are further from the carrier frequency. Because this pulse was chirped by SPM, the sections of the pulse where the absolute value of the slope of the complex pulse envelope is high are the sections of the pulse with spectral components far from the carrier frequency.

Figure 5-14: The pulse phase before and after going through the Fabry-Perot filter.
Figure 5-15: The amplitude of the Fourier Transform of the pulse before and after going through the Fabry-Perot filter. The large side lobes of the Fourier Transform are attenuated considerably by the filter.

Figure 5-16: The phase of the Fourier Transform of the pulse before and after going through the Fabry-Perot filter.
5.2.2 Mach-Zehnder Filter
This test is designed to show the validity of the Mach-Zehnder filter model. The input to the Mach-Zehnder filter is the gaussian pulse chirped by SPM generated in section 5.1.2. The links in this model are not duplex as the object of the model is to examine the effects of the Mach-Zehnder model on the complex pulse envelope. The filter in this model has the following parameters:

- \( FSR = 0.5 \) THz

The filter has a significant effect on the complex pulse envelope, as seen below (See Figure 5-18).
Figure 5-18: The pulse amplitude coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. Because the carrier frequency lies centered on a FSR of the Mach-Zehnder filter, the pulse is split into two pulses with one pulse getting almost all of the energy. A null of the transfer function for the pulse going to RCV lies directly on the carrier frequency, and this creates a null for components of the pulse with frequencies equal to the carrier frequency. This corresponds to flat sections of the pulse. This is the reason for the null in the center of the pulse.

Figure 5-19: The pulse phase coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter.
Figure 5-20: The amplitude of the Fourier Transform coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter. Because the carrier frequency lies centered on a FSR of the Mach-Zehnder filter, the pulse is split into two pulses with one pulse getting almost all of the energy. A null of the transfer function for the pulse going to RCV lies directly on the carrier frequency, and this creates a null in the amplitude of the Fourier Transform at the carrier frequency.

Figure 5-21: The phase of the Fourier Transform coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter.
5.3 Fused Biconical Coupler

This test is designed to show the validity of the Fused Biconical Coupler model. The input to the Fused Biconical Coupler is the gaussian pulse chirped by SPM generated in section 5.1.2. The links in this model are not duplex as the object of the model is to examine the effects of the Fused Biconical Coupler model on the complex pulse envelope. The FBC in this model has the following parameters:

- \( r = 8 \mu m \)
- \( \Delta r = 0 \mu m \)
- \( Z = 8716 \mu m \)

The FBC has a significant effect on the complex pulse envelope, as seen below (See Figure 5-23).
Figure 5-23: The pulse amplitude coming in through port 0 and leaving through ports 2 and 3 of the Fused Biconical Coupler. Because the carrier frequency lies near an area of the FBC transfer functions where the two pulses are split roughly evenly the pulse power is split roughly evenly. Because the slopes of the transfer functions are so great in this area, the one pulse receives most of its energy from the higher frequency spectral components, while the other pulse receives most of its energy from the lower frequency spectral components.

Figure 5-24: The pulse phase coming in through port 0 and leaving through ports 2 and 3 of the Mach-Zehnder filter.
Figure 5-25: The amplitude of the Fourier Transform of the pulse coming in through port 0 and leaving through ports 2 and 3 of the Fused Biconical Coupler. The FBC transfer functions send most of the higher frequency energy to RCV, and most of the lower frequency energy to RCVB.

Figure 5-26: The phase of the Fourier Transform of the pulse coming in through port 0 and leaving through ports 2 and 3 of the Fused Biconical Coupler.
Chapter 6: Conclusion

The AON Model Suite was designed to allow for the rapid prototyping of All Optical Networks. In combination with the OPNET simulation platform, the AON Model Suite effectively addresses the three key issues involved in simulating AONs:

- Ease of use for rapid prototyping
- Simulation accuracy and speed
- Ease of use in the display and analysis of simulation results

The AON Model Suite characterization of pulses allows for accurate modeling of pulse transmission through the optical devices found in an AON. By modeling the complex amplitude of a pulse, both linear and non-linear effects of the optical components can be accurately modeled.

Further work can be done on the Model Suite in order to increase the accuracy of the simulations. Polarization is not currently tracked due to the seemingly random fluctuations of polarization state in optical fibers. The AON Model Suite could be improved by:

- Keeping track of the polarization state. This would require modeling polarization change in the optical components, and modeling the effects of polarization. This would allow for the accurate modeling of Four Wave Mixing and a number non-linear effects.
- Developing a stochastic model that would approximate the effects of polarization.

Additionally, all of the component models could be improved. Non-linearities are currently only modeled in the fiber.

These improvements can be made relatively easily due to the modular design of the AON Model Suite and the flexibility provided by the OPNET simulation platform.
Appendix A: Component Process Model Reports

This appendix contains the process model reports of each of the models included in the AON Model Suite. Detailed information about the concepts behind the models including model attributes can be found in chapter four. The following process model reports are included:

- `aon_xmt0` The AON Model Suite Transmitter model (single gaussian pulse).
- `aon_xmt_seq` The AON Model Suite Transmitter model (sequence of gaussian pulses).
- `aon_xmt0_sech` The AON Model Suite Transmitter model (single hyperbolic secant pulse).
- `aon_xmt_sech_seq` The AON Model Suite Transmitter model (sequence of hyperbolic secant pulses).
- `aon_fib` The AON Model Suite Fiber model.
- `aon_fbc` The AON Model Suite Fused Biconical Coupler model.
- `aon_stc` The AON Model Suite Star Coupler model.
- `aon_amp` The AON Model Suite Amplifier model.
- `aon_ase` The AON Model Suite ASE Filter model.
- `aon_fabry` The AON Model Suite Fabry-Perot Filter model.
- `aon_mzf` The AON Model Suite Mach-Zehnder Filter model.
- `aon_wdm` The AON Model Suite Wavelength Division (De) Multiplexer model.
- `aon_rou` The AON Model Suite Router model.
- `aonprobe` The AON Model Suite Probe model.
- `aon_rcv` The AON Model Suite Receiver model.
A.1 aon_xmt0

**Process Model Attributes**

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (ps)</td>
</tr>
<tr>
<td>peak power</td>
<td>promoted</td>
<td>double</td>
<td>0.1 (W)</td>
</tr>
<tr>
<td>frequency</td>
<td>promoted</td>
<td>double</td>
<td>192 (THz)</td>
</tr>
<tr>
<td>m</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>chirp</td>
<td>promoted</td>
<td>double</td>
<td>0.0</td>
</tr>
<tr>
<td>time</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (sec.)</td>
</tr>
<tr>
<td>source ID</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
</tbody>
</table>

**Header Block**

```c
/* AON Model Suite */
/* Greg Campbell */
#include "mach.h"
#include "aon_base.ex.h"
#include "aon_xmt.ex.h"
```

**State Variable Block**

```c
/* State variable */
Objid my-id:
double frequency;
double xmt_time;
AonXmt_Gaussian gaussian;
int source_id:
```

**Temporary Variable Block**

```c
Packet* pkptr;
CmathT_Complex* shape;
AonT_Pulse* pulse;
```

**forced state init**

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

**enter execs init**

```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_lma_obj_attr_get (my_id, "t0", &gaussian.t0);
op_lma_obj_attr_get (my_id, "peak power", &gaussian.peak_power);
op_lma_obj_attr_get (my_id, "frequency", &gaussian.frequency);
```
op_ima_objattrget (my_id, 'chirp', &gaussian.chirp);
op_ima_objattrget (my_id, 'time', &xmt_time);
op_ima_objattrget (my_id, 'source ID', &source_id);

15 /* Send single pulse */
shape = Aon_Xmt_Gaussian (&gaussian);
pulse = Aon_Pulse_Create (source_id, opstim_time, 0,
frequency, shape, gaussian.peak_power);

pkptr = Aon_Pulse_Packet_Create (pulse);
op_pk_send_delayed (pkptr, 0, xmt_time);

<table>
<thead>
<tr>
<th>transition</th>
<th>init -&gt; rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_2</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unforced state</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>rest</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
</tr>
</tbody>
</table>
Process Model Report: aon_xmt_seq
Tue May 30 14:47:26 1995
Page 1 of 3

All Optical Network Model Suite

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (ps)</td>
</tr>
<tr>
<td>peak power</td>
<td>promoted</td>
<td>double</td>
<td>0.1 (W)</td>
</tr>
<tr>
<td>frequency</td>
<td>promoted</td>
<td>double</td>
<td>192 (THz)</td>
</tr>
<tr>
<td>m</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>chirp</td>
<td>promoted</td>
<td>double</td>
<td>0.0</td>
</tr>
<tr>
<td>start time</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (psec.)</td>
</tr>
<tr>
<td>source ID</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>spacing</td>
<td>promoted</td>
<td>double</td>
<td>100 (psec.)</td>
</tr>
<tr>
<td>initial state</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>pn connections</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>state bits</td>
<td>promoted</td>
<td>integer</td>
<td>5</td>
</tr>
<tr>
<td>repeat</td>
<td>promoted</td>
<td>integer</td>
<td>0</td>
</tr>
</tbody>
</table>

Header Block

```c
/* AON Model Suite */
/* Greg Campbell */
#include "math.h"
#include "aon_base.ex.h"
#include "aon_xmt.ex.h"
```

State Variable Block

```c
/* State variable */
Objid mnid;
double frequency;
double start_time;
5 AonTXmt_Gaussian gaussian;
AonTXmt_Seq seq;
int source_id;
double spacing;
int repeat;
10 AonTPulse* pulse;
int initial_state;
```

Temporary Variable Block

```c
Packet* pkpr;
CmathT_Complex* shape;
AonTPulse* new_pulse;
int out;
```

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_ima_obj_attr_get (my_id, "c0", &gaussian.id);
op_ima_obj_attr_get (my_id, "peak power", &gaussian.peak_power);
op_ima_obj_attr_get (my_id, "frequency", &frequency);
op_ima_obj_attr_get (my_id, "chirp", &gaussian.chirp);
op_ima_obj_attr_get (my_id, "peak power", &gaussian.peak_power);
op_ima_obj_attr_get (my_id, "source ID", &source_id);

/* Set initial state. */
seq.state = initial_state;

/* Set self-interrupts for start time. */
op_intrpt_schedule_self (start_time, 0);

/* Generate pulse template. */
shape = Aon_Xmt_Gaussian (&gaussian);
pulse = Aon_Pulse_Create (source_id, op_sim_time (), frequency, shape, gaussian.peak_power);

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_2</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>rest</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

/* Determine whether a zero or one should be transmitted. */
out = Aon_Xmt_Seq (&seq);
```c
if (out == 1)
{
    /* Copy and packetize pulse for transmission. */
    new_pulse = Aon_Pulse_Copy (pulse);
    new_pulse->timestamp = op_sim_time ();
    pkptr = Aon_Pulse_Packet_Create (new_pulse);
    /* Transmit pulse. */
    op_pk_send (pkptr, 0);
}

/* Set interrupt for next pulse transmission. */
if ((seq.state != initial_state) II (repeat))
    op_intrpt_schedule_self (op_sim_time () + spacing, 0);
```

<table>
<thead>
<tr>
<th>transition</th>
<th>rest -&gt; rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_3</td>
</tr>
<tr>
<td>condition</td>
<td>string</td>
</tr>
<tr>
<td>executive</td>
<td>string</td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.3 aon_xmt0_sech


All Optical Network Model Suite

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (ps)</td>
</tr>
<tr>
<td>peak power</td>
<td>promoted</td>
<td>double</td>
<td>0.1 (W)</td>
</tr>
<tr>
<td>frequency</td>
<td>promoted</td>
<td>double</td>
<td>192 (THz)</td>
</tr>
<tr>
<td>chirp</td>
<td>promoted</td>
<td>double</td>
<td>0.0</td>
</tr>
<tr>
<td>time</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (sec.)</td>
</tr>
<tr>
<td>source ID</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
</tbody>
</table>

Header Block

/* AON Model Suite */
/* Greg Campbell */
#include cmath.h
#include aon_base.ex.h
#include aon_xmt.ex.h

State Variable Block

/* State variable */
Objid my_id;
double frequency;
double xmt_time;
AonT_Xmt_SeCH \sech;
int source_id;

Temporary Variable Block

Packet* pkptr;
CmathTComplex* shape;
AonT_Pulse* pulse;

forced state init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_lma_obj_attr_get (my_id, "t0", &sech.t0);
op_lma_obj_attr_get (my_id, "peak power", &sech.peak_power);
op_lma_obj_attr_get (my_id, "frequency", &sech.frequency);
op_lma_obj_attr_get (my_id, "chirp", &sech.chirp);
op_lma_obj_attr_get (my_id, "time", &xmt_time);
op_lma_obj_attr_get (my_id, "source ID", &source_id);
/* Send single pulse */
shape = Aon_Xmt_Sech (&sech);
pulse = Aon_Pulse_Create (source_id, op_sim_time(),
  frequency, shape, sech.peak_power);

pkptr = Aon_Pulse_Packet_Create (pulse);
op_pk_send_delayed (pkptr, 0, xmt_time);

<table>
<thead>
<tr>
<th>transition</th>
<th>init -&gt; rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_2</td>
</tr>
<tr>
<td>condition</td>
<td>string</td>
</tr>
<tr>
<td>executive</td>
<td>string</td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unforced state</th>
<th>rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>rest</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
</tr>
</tbody>
</table>
### A.4 aon_xmt_sech_seq

#### Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (ps)</td>
</tr>
<tr>
<td>peak power</td>
<td>promoted</td>
<td>double</td>
<td>0.1 (W)</td>
</tr>
<tr>
<td>frequency</td>
<td>promoted</td>
<td>double</td>
<td>192 (THz)</td>
</tr>
<tr>
<td>chirp</td>
<td>promoted</td>
<td>double</td>
<td>0.0</td>
</tr>
<tr>
<td>start time</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (psec.)</td>
</tr>
<tr>
<td>source ID</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>spacing</td>
<td>promoted</td>
<td>double</td>
<td>100 (psec.)</td>
</tr>
<tr>
<td>initial state</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>pn connections</td>
<td>promoted</td>
<td>integer</td>
<td>1</td>
</tr>
<tr>
<td>state bits</td>
<td>promoted</td>
<td>integer</td>
<td>5</td>
</tr>
<tr>
<td>repeat</td>
<td>promoted</td>
<td>integer</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Header Block

```c
/* AON Model Suite */
/* Greg Campbell */
#include <cmath.h>
#include aon_base.ex.h
#include aon_.nt.ex.h
```

#### State Variable Block

```c
/* State variable */
Objid my_id;
double frequency;
double start_time;
AonT_Xmt_Gaussian sech;
AonT_Xmt_Seq seq;
int source_id;
double spacing;
int repeat;
AonT_Pulse* pulse;
int initial_state;
```

#### Temporary Variable Block

```c
Packet* pkptr;
CmathT_Complex*shape;
AonT_Pulse* new_pulse;
it out;
```

#### forced state init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>
enter execs Init

/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_ima_obj_attrget (my_id, 'to', &sech.t0);
op_ima_obj_attrget (my_id, 'peak power', &sech.peak_power);
op_ima_obj_attrget (my_id, 'frequency', &frequency);
op_ima_obj_attrget (my_id, 'chirp', &sech.chirp);
op_ima_obj_attrget (my_id, 'start time', &starttime);
op_ima_obj_attrget (my_id, 'source ID', &source_id);
op_ima_obj_attrget (my_id, 'spacing', &spacing);
op_ima_obj_attrget (my_id, 'initial state', &initialstate);
op_ima_obj_attrget (my_id, 'pn connections', &seq.connections);
op_ima_obj_attrget (my_id, 'state bits', &seq.n);
op_ima_obj_attrget (my_id, 'repeat', &repeat);

/* Set initial state. */
seq.state = initial_state;

/* Set self-interrupt for start time. */
op_intrpt_schedule_self (start_time, 0);

/* Generate pulse template. */
shape = Aon_Xmt_Gaussian (&sech);
pulse = Aon_Pulse_Create (source_id, opsim_time (),
frequency, shape, sech.peak_power);

transition init -> rest

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_2</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

unforced state rest

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>rest</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

exit execs rest

/* Determine whether a zero or one should be transmitted. */
out = Aon_Xmt_Seq (&seq);

if (out == 1)
Process Model Report: aon_xmt_sech_seq

All Optical Network Model Suite

/* Copy and packetize pulse for transmission. */
new_pulse = Aon_Pulse_Copy (pulse);
new_pulse->timestamp = op_sim_time ();
pkptr = Aon_Pulse_Packet_Create (new_pulse);

/* Transmit pulse. */
op_pk_send (pkptr, 0);

/* Set interrupt for next pulse transmission. */
if ((seq.state != initial_state) || (repeat))
    op_intrpt_schedule_self (op_sim_time () + spacing, 0);

<table>
<thead>
<tr>
<th>transition</th>
<th>rest -&gt; rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_3</td>
</tr>
<tr>
<td>condition</td>
<td>string</td>
</tr>
<tr>
<td>executive</td>
<td>string</td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.5 aon_fib

Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T Raman</td>
<td>promoted</td>
<td>double</td>
<td>0.005 (ps)</td>
</tr>
<tr>
<td>B1 at freq1</td>
<td>promoted</td>
<td>double</td>
<td>4,875 (ps/km)</td>
</tr>
<tr>
<td>B1 at freq2</td>
<td>promoted</td>
<td>double</td>
<td>4,871.67 (ps/km)</td>
</tr>
<tr>
<td>B2 at freq1</td>
<td>promoted</td>
<td>double</td>
<td>-20.0 (ps2/km)</td>
</tr>
<tr>
<td>B2 at freq2</td>
<td>promoted</td>
<td>double</td>
<td>0.0 (ps2/km)</td>
</tr>
<tr>
<td>B3</td>
<td>promoted</td>
<td>double</td>
<td>0.0 (ps3/km)</td>
</tr>
<tr>
<td>alpha</td>
<td>promoted</td>
<td>double</td>
<td>0.2 (dB/km)</td>
</tr>
<tr>
<td>Length</td>
<td>promoted</td>
<td>double</td>
<td>100 (km)</td>
</tr>
<tr>
<td>granularity</td>
<td>promoted</td>
<td>double</td>
<td>10 (iter/L)</td>
</tr>
<tr>
<td>A eff</td>
<td>promoted</td>
<td>double</td>
<td>65 (micron2)</td>
</tr>
<tr>
<td>n2</td>
<td>promoted</td>
<td>double</td>
<td>3.2E-16 (cm2/W)</td>
</tr>
<tr>
<td>freq1</td>
<td>promoted</td>
<td>double</td>
<td>192 (THz)</td>
</tr>
<tr>
<td>freq2</td>
<td>promoted</td>
<td>double</td>
<td>225 (THz)</td>
</tr>
<tr>
<td>Gmax</td>
<td>promoted</td>
<td>double</td>
<td>1E-16 (km/W)</td>
</tr>
<tr>
<td>Fmax</td>
<td>promoted</td>
<td>double</td>
<td>12 (THz)</td>
</tr>
</tbody>
</table>

Header Block

/* AON Model Suite */
/* Greg Campbell */
#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_fib.ex.h"

State Variable Block

/* State variable */
AonT_Fib_Desc  \fib_desc;
AonT_Port_Pulse*  \port[2];
AonT_Port_Noise_Out*  \noise_out[2];
double  \last_time;

Temporary Variable Block

int  event_type;
Pkt*  pkpt;
int  port_index;
int  type;
AonT_Pulse  *pulse;
AonT_Noise  *noise;
Objid  my_id;

forced state  init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td></td>
<td>textlist</td>
<td>(See below,)</td>
</tr>
</tbody>
</table>
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_ima_obj_attr_get (my_id, "T Raman", &(fib_desc.T_Raman));
op_ima_obj_attr_get (my_id, "freq1", &(fib_desc.freql));
op_ima_obj_attr_get (my_id, "freq2", &(fib_desc.freq2));
op_ima_obj_attr_get (my_id, "B1 at freq1", &(fib_desc.B1_freq1));
op_ima_obj_attr_get (my_id, "B1 at freq2", &(fib_desc.B1_freq2));
op_ima_obj_attr_get (my_id, "B2 at freq1", &(fib_desc.B2_freq1));
op_ima_obj_attr_get (my_id, "B2 at freq2", &(fib_desc.B2_freq2));
op_ima_obj_attr_get (my_id, "B3", &(fib_desc.B3));
op_ima_obj_attr_get (my_id, "alpha", &(fib_desc.alpha));
op_ima_obj_attr_get (my_id, "Length", &(fib_desc.Length));
op_ima_obj_attr_get (my_id, "granularity", &(fib_desc.granularity));
op_ima_obj_attr_get (my_id, "A eff", &(fib_desc.A_eff));
op_ima_obj_attr_get (my_id, "n2", &(fib_desc.n2));
op_ima_obj_attr_get (my_id, "Gmax", &(fib_desc.Gmax));
op_ima_obj_attr_get (my_id, "Fmax", &(fib_desc.Fmax));

/* Initialize variables. */
fib_desc.alpha = 1.0 - cmathdB (fib_desc.alpha);
noiseout[0] = Aon_Port_Noise_Out_Create();
noiseout[1] = Aon_Port_Noise_Out_Create();
port[0] = Aon_Port_Pulse_Create();
port[1] = Aon_Port_Pulse_Create();
last_time = 0.0;


<table>
<thead>
<tr>
<th>transition</th>
<th>Init -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_O</td>
</tr>
<tr>
<td>condition</td>
<td>string</td>
</tr>
<tr>
<td>executive</td>
<td>string</td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unforced state</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>s:steady</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
</tr>
</tbody>
</table>
exit execs (See below.) textlist (See below.)

status unforced toggle unforced

```c
Aon_Fib_Prop_Port (port[0], &fib_desc, last_time, op_sim_time());
Aon_Fib_Prop_Port (port[1], &fib_desc, last_time, op_sim_time());
last_time = op_sim_time();

/* Get event */
event_type = op_intrpt_type();

if (event_type == OPC_INTRPT_SELF)
{
    /* Do module specific actions. */
    port_index = op_intrpt_code();
    pulse = Aon_Fib.Exit_Pulse (port[port_index], &fib_desc,
        op_sim_time());
    pkptr = Aon_Pulse_Packet_Create (pulse);
    Aon_Pulse_Packet_Send_Delayed (pkptr, port_index, 0.0);

    if (event_type == OPC_INTRPT_STRM)
    {
        port_index = op_intrpt_strm();

        if (port_index > 1)
            op_sim_end("Invalid port index",*** **);

        pkptr = op_pk-get (port_index);
        type = Aon_Event_Packet_Type (pkptr);

        if (type == AONCP_PKT_PULSE)
        {
            pulse = Aon_Pulse_Packet_Get (pkptr);
            Aon_Port_Pulse_Append (port[port_index], pulse);
            op_intrpt_schedule_self (op_sim_time () + Aon_Fib_Delay (pulse, &fib_desc),
                1 - port_index);
            Aon_Pulse_Packet_Destroy (pkptr);
        }
        else
        {
            noise = Aon_Noise_Packet_Get (pkptr);
            noise->power = noise->power * exp ((-1.0)*fib_desc.alpha);
            Aon_Port_Noise_Out_Handle_Abs_Reuse
                (noise_out1 - port_index1, pkptr, 1 - port_index,
                    Aon_Fib_B1 ((AonI_Low_Freq +
                        ((double) noise->freq_bin / (double) AonI_N_Segment) *
                            (AonI_High_Freq - AonI_Low_Freq)),
                        &fib_desc) * fib_desc-Length);
        }
    }
    else
    {
```
```
### Transition: steady -> steady

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>tr_1</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>Condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>Executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>Color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>Drawing Style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>
### A.6 aon_fbc

**Process Model Report: aon_fbc**

All Optical Network Model Suite

---

#### Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core radius</td>
<td>promoted</td>
<td>double</td>
<td>8.0 (micron)</td>
</tr>
<tr>
<td>Length</td>
<td>promoted</td>
<td>double</td>
<td>10 (micron)</td>
</tr>
<tr>
<td>delta r</td>
<td>promoted</td>
<td>double</td>
<td>0.0 (micron)</td>
</tr>
<tr>
<td>Power Loss</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (dB)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
</tbody>
</table>

#### Header Block

/* AON Model Suite */
/* Greg Campbell */

#include "math.h"
#include "aon_base.ex.h"
#include "aon_fbc.ex.h"

#### State Variable Block

/* State variable */
AonT_Port_NoiseOut* noise_out[4];
AonT_Port_NoiseIn* noisein[4];
AonT_FBC_Desc* fbc_desc;
double delay;

#### Temporary Variable Block

int event_type;
Packet* pkptr;
int port_index;
int type;
AonT_Pulse* pulse;
AonT_Pulse* new_pulse;
AonT_Noise* noise;
AonT_Noise* new_noise;
Objid my_id;
double r;
double delta_r;
double z;
double a;

#### forced state Init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_lma_obj_attr_get(my_id, "Core radius", &r);
op_lma_obj_attr_get(my_id, "delta r", &delta_r);
op_lma_obj_attr_get(my_id, "Length", &L);
op_lma_obj_attr_get(my_id, "Power Loss", &a);
op_lma_obj_attr_get(my_id, "Delay", &delay);

/* Initialize variables. */
fbc_desc = Aon_FBC_Create(r, delta_r, L, a);
note_out[0] = Aon_PortNoise_Out_Create();
note_out[1] = Aon_PortNoise_Out_Create();
note_in[0] = Aon_Port_Noise_In_Create();
note_in[1] = Aon_Port_Noise_In_Create();
note_in[2] = Aon_Port_Noise_In_Create();
note_in[3] = Aon_Port_Noise_In_Create();

transition
init -> steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_0</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

unforced state steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>steady</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

exit execs steady

/* Get event */
event_type = op_intrpt_type();

if (event_type == OPC_INTRPT_SELF)
{
    /* Do module specific actions. */
}

if (event_type == OPC_INTRPT_STMT)
```c
10  { 
    port_index = op_intrpt_strm();
    if (port_index > 3)
        op_sim_end('Invalid port index', '***');
    pkptr = op_pkget(port_index);
    type = Aon_Event_Packet_Type(pkptr);

20   if (type == AONC_PKT_PULSE)
        { 
            pulse = Aon_Pulse_Packet_Get(pkptr);
            new_pulse = Aon_Pulse_Copy(pulse);
            Aon_FBC_Pulse1(pulse, fbc_desc);
            Aon_FBC_Pulse2(new_pulse, fbc_desc);
            Aon_Pulse_Packet_Send_Delayed(pkptr, 
                (port_index + 2) % 4, delay);
            pkptr = Aon_Pulse_Packet_Create(new_pulse);
            Aon_Pulse_Packet_Send_Delayed(pkptr, 
                (3 - port_index), delay);
        }
    else 
        { 
            noise = Aon_Noise_Packet_Get(pkptr);
            noise->power = Aon_Port_Noise_In_Handle 
                (noise_in [port_index], noise);
            new_noise = Aon_Noise_Copy(noise);
            Aon_FBC_Noise1(noise, fbc_desc);
            Aon_FBC_Noise2(new_noise, fbc_desc);
35       Aon_Noise_Packet_Create(new_noise);
            Aon_Noise_Packet_Create(new_noise);
            Aon_Packet_Out_Dif_Reuse 
                (port_index + 2 % 4, pkptr, 
                    (port_index + 2) % 4, delay);
            pkptr = Aon_Noise_Packet_Create(new_noise);
            Aon_Noise_Packet_Out_Dif_Reuse 
                (noise_out [3 - port_index], pkptr, 
                    (3 - port_index), delay);
        }
}
```

**transition steady -> steady**

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_1</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

112
A.7 aon_stc

<table>
<thead>
<tr>
<th>Process Model Attributes</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>promoted</td>
<td>integer</td>
<td>2</td>
</tr>
<tr>
<td>insertion loss</td>
<td>promoted</td>
<td>double</td>
<td>0.0 (dB)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
</tbody>
</table>

Header Block

/* AON Model Suite */
/* Greg Campbell */
#include "math.h"
#include "aon_base.ex.h"
#include "aon_stc.ex.h"

State Variable Block

/* State variable */
AonT_STC_Desc* stc_desc;
AonT_Port_Noise_Out_Ptr* noise_out;
AonT_Port_Noise_In_Ptr* noise_in;

Temporary Variable Block

int event_type;
Packet* pkpu;
int port_index;
int type;
AonT_Pulse* pulse;
AonT_Pulse* new_pulse;
AonT_Noise* noise;
AonT_Noise* new_noise;
Objid myid;
i;
double loss;
int N;
double delay;

forced state

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

entrée execs

/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();
/* Determine module specific attributes. */
op_ima_obj_attr_get (myid, *N', &N);
op_ima_obj_attr_get (myid, *delay', &delay);
op_ima_obj_attr_get (myid, *insertion loss', &loss);

/* Initialize variables. */
stc_desc = Aon_STC_Create (N, loss, delay);

noise_out = (AonT_Port_Noise_Out_Ptr*) malloc (2 * N * sizeof (AonT_Port_Noise_Out_Ptr));
noise_in = (AonT_Port_Noise_In_Ptr) malloc (2 * N * sizeof (AonT_Port_Noise_In_Ptr));

for (i = 0; i < 2*N; i++)
{
  (*(noise_out + i)) = AonPort_Noise_Out_Create (0);
  (*(noise_in + i)) = AonPort_Noise_In_Create (0);
}

transition init -> steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_0</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

unforced state steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>steady</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

exit execs steady

/* Get event */
event_type = op_intrpt_type ();

if (event_type == OPC_INTRPT_SELF)
{
  /* Do module specific actions. */
}

if (event_type == OPC_INTRPT_STRM)
{
  port_index = op_intrpt_strm ();
  if (port_index >= 2*stc_desc->N)
    op_sim_end ('Invalid port index', '\*', '\*', '\*');
  pkptr = op_pk_get (port_index);
type = Aon_Event_Packet_Type (pkptr);

if (type == AONC_PKT_PULSE)
    pulse = Aon_Pulse_Packet_Get (pkptr);
    Aon_STC_Propagate (pulse, stc_desc);
    if (port_index < stc_desc->N)
        Aon_Pulse_Packet_Send_Delayed (pkptr,
            stc_desc->N, stc_desc->delay);
    else
        Aon_Pulse_Packet_Send_Delayed (pkptr,
            0, stc_desc->delay);
    for (i = 1; i < stc_desc->N; i++)
        if (i == 1; i < stc_desc->N; i++)
            new_pulse = Aon_Pulse_Copy (pulse);
            pkptr = Aon_Pulse_Packet_Create (new_pulse);
            if (port_index < stc_desc->N)
                Aon_Pulse_Packet_Send_Delayed (pkptr,
                    i + stc_desc->N, stc_desc->delay);
            else
                Aon_Pulse_Packet_Send_Delayed (pkptr,
                    i, stc_desc->delay);
    else
        noise = Aon_Noise_Packet_Get (pkptr);
        noise->power = Aon_Port_Noise_In_Handle
            (*(noise_in + port_index), noise);
        Aon_STC_Noise_Propagate (noise, stc_desc);
        if (port_index < stc_desc->N)
            Aon_Port_Noise_Out_Handle_Dif_Reuse
                (*(noise_out + stc_desc->N), pkptr,
                    stc_desc->N, stc_desc->delay);
        else
            Aon_Port_Noise_Out_Handle_Dif_Reuse
                (*(noise_out + 0), pkptr,
                    0, stc_desc->delay);
    for (i = 1; i < stc_desc->N; i++)
        if (i == 1; i < stc_desc->N; i++)
            new_noise = Aon_Noise_Copy (noise);
            pkptr = Aon_Noise_Packet_Create (new_noise);
            if (port_index < stc_desc->N)
All Optical Network Model Suite

```c
...{
    Aon_Port_Noise_Out_Handle_Dif_Reuse
    (*(noise_out + i + stc_desc->N), pkptr,
     i + stc_desc->N, stc_desc->delay);
}
else{
    Aon_Port_Noise_Out_Handle_Dif_Reuse
    (*(noise_out + i), pkptr,
     i, stc_desc->delay);
}
}
```

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
### Process Model Report: aon_amp

**All Optical Network Model Suite**

---

#### Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain</td>
<td>promoted</td>
<td>double</td>
<td>10 (dB)</td>
</tr>
<tr>
<td>Saturation</td>
<td>promoted</td>
<td>double</td>
<td>0.1 (W)</td>
</tr>
<tr>
<td>Noise Coef</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (dB)</td>
</tr>
<tr>
<td>Relax Time</td>
<td>promoted</td>
<td>double</td>
<td>1,000 (ps)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
<tr>
<td>delta noise percent</td>
<td>promoted</td>
<td>double</td>
<td>10 (percent)</td>
</tr>
</tbody>
</table>

---

#### Header Block

```c
/* AON Model Suite */
/* Greg Campbell */

#include "math.h"
#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_amp.ex.h"
```

---

#### State Variable Block

```c
/* State variable */
double old_time;
AonT_Amp_Desc *amp;
Eventhandle update_event;
```

---

#### Temporary Variable Block

```c
int event_type;
Packet* pkptr;
int port_index;
int type;
AonT_Pulse *pulse;
AonT_Noise *noise;
Objid my_id;
double next_update_time;
double amp_gain;
double amp_noise;
double amp_sat;
double amp_tau;
double amp_delay;
double amp_d_noise;
```

---

#### forced state init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>
```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

/* Create amplifier description structure. */
amp = Aon_Amp_DescCreate();

/* Determine module specific attributes. */
op_ima_obj_attr_get(myid, "Gain", &amp..gain);
op_ima_obj_attr_get(myid, "Saturation", &amp_sat);
op_ima_obj_attr_get(myid, "Noise Coef", &amp_noise);
op_ima_obj_attr_get(myid, "Relax Time", &amp_tau);
op_ima_obj_attr_get(myid, "Delay", &amp_delay);
op_ima_obj_attr_get(myid, "delta noise percent", &amp_d_noise);

/* Initialize variables. */
amp->gain = cmath_dB((-1.0) * amp_gain);
amp->sat = amp_sat;
amp->noise = cmath_dB((-1.0) * amp_noise);
amp->tau = amptau;
amp->delay = ampdelay;
amp->d_noise = ampd_noise / 100.0;

/* Set time of last update to 0.0. */
old_time = 0.0;
Aon_Amp_Noise_Update(amp, opsimtime());
```

### Transition Table: init -> steady

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_0</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

### Unforced State: steady

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>steady</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

```c
/* Determine average pulse power. */
amp->pulse_power = amp->pulse_power * exp((old_time - opsim_time()) / amp->tau);
/* Get event */
```
```c
event_type = op_intrpt_type ();

if (event_type == OPC_INTRPT_SELF)
{
    if (op_intrpt_code () != AONC AMP_UPDATE)
        AonAmpPulse_Power Interrupt Get (amp);
}

if (event_type == OPC_INTRPT_STRM)
{
    /* Cancel pending update event, if there is one. */
    if (op_ev_valid (update_event))
        op_ev_cancel (update_event);

    port_index = op_intrpt_strm ();

    if (port_index != 0)
        op_sim_end ("Invalid port index", ",");

    pk.ptr = op_pk (port_index);

    type = Aon_Event_Packet_Type (pk.ptr);

    if (type == AONC_PKT_PULSE)
    {
        pulse = Aon_Pulse_Packet_Get (pk.ptr);
        Aon_Amp_Pulse_Power_Interrupt_Set (amp, pulse);
        Aon_Amp_Pulse (amp, pulse);
        Aon_Pulse_Send_Delayed (pk.ptr, 0, amp->delay);
    }
    else
    {
        noise = Aon_Noise_Packet_Get (pk.ptr);
        amp->cv_noise = amp->cv_noise + noise->power -
        (* (amp->noise_in->noise_array + noise->freq_bin));
        (* (amp->noise_in->noise_array + noise->freq_bin)) = noise->power;
        Aon_Noise_Packet_Destroy (pk.ptr);
    }
}

Aon_Amp_Noise_Update (amp, op_sim_time ());
old_time = op_sim_time ();

if (amp->pulse_power > AonI_Min_Power)
{
    /* next_update_time = op_sim_time () - amp->tau *
        log (1.0 - amp->d_noise); */
    next_update_time = op_sim_time () + Aon_Amp_Next_Update (amp);
    update_event = op_intrpt_schedule_self (next_update_time,
    AONC_AMP_UPDATE);
}
```

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td>RGB333</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>color</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.9 aon_ase

Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>promoted</td>
<td>double</td>
<td>0.05 (THz)</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>promoted</td>
<td>double</td>
<td>0.01 (THz)</td>
</tr>
<tr>
<td>Attenuation</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (dB)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
</tbody>
</table>

Header Block

```c
/* AON Model Suite */
/* Greg Campbell */
#include <cmath.h>
#include "aon_base.ex.h"
#include "aon_ase.ex.h"
```

State Variable Block

```c
AonT_Port_Noise_Out* noiseout[2];
AonT_ASE_Desc* ase_desc;
double delay;
```

Temporary Variable Block

```c
int event_type;
Packer* pkptr;
int port_index;
int type;
AonT_Pulse* pulse;
AonT_Noise* noise;
Objid my_id;
double W;
double FSR;
double a;
```

forced state Init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

enter execs Init

```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();
/* Determine module specific attributes. */
```
```c
op_ima_obj_attr_get (my_id, "Attenuation", &a);
op_ima_obj_attr_get (my_id, "FSR", &FSR);
op_ima_obj_attr_get (my_id, "Bandwidth", &W);
op_ima_obj_attr_get (my_id, "Delay", &delay);

/* Initialize variables. */
ase_desc = Aon_ASE_Create (FSR, W, a);
n noise_out[0] = Aon_Port_Noise_Out_Create (0);
n noise_out[1] = Aon_Port_Noise_Out_Create (0);
```

### Transition: init -> steady

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_0</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

### Unforced State: steady

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>steady</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

```c
/* Get event */
event_type = op_intrpt_type (0);

if (event_type == OPC_INTRPT_SELF)
{
    /* Do module specific actions. */
}

if (event_type == OPC_INTRPT_STRM)
{
    port_index = op_intrpt_strm (0);

    if (port_index > 1)
        op_sim_end ("Invalid port index.\n", ");

    pkptr = op_pk_get (port_index);
    type = Aon_Event_Packet_Type (pkptr);

    if (type == AON_PKT_PULSE)
        pulse = Aon_Pulse_Packet_Get (pkptr);
        Aon_ASE_Pulse (pulse, ase_desc);
        Aon_Pulse_Packet_Send_Delayed (pkptr,
            1 - port_index, delay);
```
... else
{
    noise = Aon_Noise_Packet_Get (pkptr);
    Aon_ASE_Noise (noise, ase_desc);
    Aon_Port_Noise_Out_Handle_Abs_Reuse (noise_out [port_index],
        pkptr, 1 - port_index, delay);
}

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.10 aon_fabry

Process Model Report: aon_fabry

Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>promoted</td>
<td>double</td>
<td>0.05 (THz)</td>
</tr>
<tr>
<td>Finesse</td>
<td>promoted</td>
<td>double</td>
<td>30 (none)</td>
</tr>
<tr>
<td>Tmax</td>
<td>promoted</td>
<td>double</td>
<td>1.0 (none)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
</tbody>
</table>

Header Block

```c
/* AON Model Suite */
/* Greg Campbell */

#include "math.h"
#include "aon_base.ex.h"
#include "aon_fab.ex.h"
```

State Variable Block

```c
/* State variable */
AonT_Port_Noise_Out* noiseout[2];
AonT_Fab_Desc* fab_desc;
double delay;
```

Temporary Variable Block

```c
int event_type;
Packet* pkptr;
int port_index;
int type;
AonT_Pulse *pulse;
AonT_Noise *noise;
Objid my_id;
double Finesse;
double FSR;
double Tmax;
```

forced state init

```
attribute | value | type | default value |
-----------|-------|------|---------------|
name       | init  | string| st            |
enter execs| (See below.) | textlist | (See below.) |
exit execs | (empty) | textlist | (empty)      |
status     | forced | toggle| unforced      |
```

enter execs init

```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
```
All Optical Network Model Suite

```c
op_ima_obj_attr_get (my_id, "Finesse", &Finesse);
10 op_ima_obj_attr_get (my_id, "FSR", &FSR);
op_ima_obj_attr_get (my_id, "Tmax", &Tmax);
op_ima_obj_attr_get (my_id, "Delay", &delay);

/* Initialize variables. */
fab_desc = Aon_Fab_Create (FSR, Finesse, Tmax);
noiseout(0] = Aon_PorT_Noise_Out_Create ();
noise_out[l] = Aon_PortNoise_OutCreate ();
```

### transition init -> steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_0</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

### unforced state steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>steady</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

### exit exec steady

```c
/* Get event */
event_type = op_intrpt_type ();

if (event_type == OPC_INTRPT_SELF)
{
    /* Do module specific actions. */
}

if (event_type == OPC_INTRPT_STRM)
{
    port_index = op_intrpt_strm ();

    if (port_index > 1)
        op_sim_end ("Invalid port index", ", ", ");

    pkptr = op_pk_get (port_index);

type = Aon_Event_Packet_Type (pkptr);

if (type == AONCPKT_PULSE)
    pulse = Aon_Pulse_Packet_Get (pkptr);
    Aon_Fab_Pulse (pulse, fab_desc);
    Aon_Pulse_Packet_Send_Delayed (pkptr,
        1 - port_index, delay);
```
else
{
    noise = Aon_Noise_Packet_Get (pkptr);
    Aon_Fab_Noise (noise, fab_desc);
    Aon_Port_Noise_Out_Handle_Abs_Reuse (noise_out [port_index],
        pkptr, 1 - port_index, delay);
}

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td>string</td>
</tr>
<tr>
<td>executive</td>
<td>string</td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.11 aon_mzf

Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>promoted</td>
<td>double</td>
<td>0.2 (THz)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
</tbody>
</table>

Header Block

```c
/* AON Model Suite */
/* Greg Campbell */

#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_mzf.ex.h"
```

State Variable Block

```c
/* State variable */
AonT_PortNoise_Out* noise_out[4];
AonT_PortNoise_In* noise_in[4];
AonT_MZF_Desc* mzfdesc;
```

Temporary Variable Block

```c
int event_type;
Packet* pkptr;
int port_index;
int type;
AonT_Pulse* pulse;
AonT_Pulse* new_pulse;
AonT_Noise* noise;
AonT_Noise* new_noise;
Objid my_id;
```

forced state init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

enter execs init

```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_ima_obj_attr_get (my_id, *FSR*, &FSR);
```
op_ima_obj_attr_get (my_id, "Delay", &delay);

/* Initialize variables. */

mzf_desc = Aon_MZF_Create (FSR);
noise_out[0] = Aon_Port_Noise_Out_Create ();
nnoise_out[1] = Aon_Port_Noise_Out_Create ();
noise_in[0] = Aon_Port_Noise_In_Create ();
nnoise_in[1] = Aon_Port_Noise_In_Create ();
nnoise_in[2] = Aon_Port_Noise_In_Create ();
nnoise_in[3] = Aon_Port_Noise_In_Create ();

/* Get event */
eventtype = op_intrpttype ();
if (eventtype == OPC_INTRPT_SELF)
{
    /* Do module specific actions. */
}
if (eventtype == OPC_INTRPT_STRM)
{
    port_index = opintrpt_strm ();
    if (port_index > 3)
        op_sim_end (*Invalid port index*, *\ldots*);
    pkptr = op_pk_get (port_index);
    type = Aon_Event_Packet_Type (pkptr);
    if (type == AONC_PKT_PULSE)
pulse = Aon_Pulse_Packet_Get (pkptr);
new_pulse = Aon_Pulse_Copy (pulse);
Aon_MZF_Pulse1 (pulse, mzf_desc);
Aon_MZF_Pulse2 (new_pulse, mzf_desc);
Aon_Pulse_Packet_Send_Delayed (pkptr,
(port_index + 2) % 4, delay);
pkptr = Aon_Pulse_Packet_Create (new_pulse);
Aon_Pulse_Packet_Send_Delayed (pkptr,
(3 - port_index), delay);
}
else
{
noise = Aon_Noise_Packet_Get (pkptr);
noise->power = AonPort_Noise_In_Handle
(noise_in [port_index], noise);
new_noi se = Aon_Noise_Copy (noise);
Aon_MZF_Noise1 (noise, mzf_desc);
Aon_MZF_Noise2 (new_noi se, mzf_desc);
Aon_Port_Noise_Out_Handle_Dif_Reuse
(noise_out [(port_index + 2) % 4], pkptr,
(port_index + 3) % 4, delay);
pkptr = Aon_Noise_Packet_Create (new_noi se);
Aon_Port_Noise_Out_Handle_Dif_Reuse
(noise_out [(3 - port_index)], pkptr,
(3 - port_index), delay);
}

<p>| transition steady -&gt; steady |</p>
<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_1</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.12 aon_wdm

Process Model Report: aon_wdm
Tue May 30 14:45:55 1995
Page 1 of 4

All Optical Network Model Suite

Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>promoted</td>
<td>double</td>
<td>32 (THz)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
<tr>
<td>Attenuation</td>
<td>promoted</td>
<td>double</td>
<td>0.0 (dB)</td>
</tr>
</tbody>
</table>

Header Block

/* AON Model Suite */
/* Greg Campbell */

#include <cmath.h>
#include "aon_base.ex.h"
#include "aon_wdm.ex.h"

State Variable Block

/* State variable */
AonT_Port_Noise_Out* noise_out[3];
AonT_Port_Noise_In* noisein[3];
AonT_WDM_Desc* wdm_desc;
double delay;

Temporary Variable Block

int event_type;
Packet* pkptr;
int port_index;
int type;
AonT_Pulse *pulse;
AonT_Pulse *new_pulse;
AonT_Noise *noise;
AonT_Noise *new_noise;
Objid my_id;
double FSR;
double a;

defined state

Init attribute value type default value
name init string st
enter execs (See below.) textlist (See below.)
exit execs (empty) textlist (empty)
status forced toggle unforced

defined state

Init attribute value type default value

enter execs

/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

130
/* Determine module specific attributes. */
op_ima_obj_attr_get (my_id, "FSR", &FSR);
op_ima_obj_attr_get (my_id, "Attenuation", &a);
op_ima_obj_attr_get (my_id, "Delay", &delay);

/* Initialize variables. */
wdm_desc = Aon_WDM_Create (FSR, a);
n_noise_out[0] = Aon_Port_Noise_Out_Create ();
n_noise_out[1] = Aon_Port_Noise_Out_Create ();
n_noise_in[0] = Aon_Port_Noise_In_Create ();
n_noise_in[1] = Aon_Port_Noise_In_Create ();
n_noise_in[2] = Aon_Port_Noise_In_Create ();

<table>
<thead>
<tr>
<th>transition</th>
<th>init -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_0</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unforced state</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>steady</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
</tr>
</tbody>
</table>

/* Get event */
if (event_type == OPC_INTRPT_SELF)
    { /* Do module specific actions. */
    }

if (event_type == OPC_INTRPT_STRM)
    {
        port_index = op_intrpt_strm ();
        if (port_index &gt; 2)
            op_sim_end ('Invalid port index', ...);
        pkptr = op_pkget (port_index);
        type = Aon_Event_Packet_Type (pkptr);
        if (type == AONC_PKT_PULSE)
```
{ pulse = Aon_Pulse_Packet_Get (pkptr);
  if (port_index == 0)
  {
    Aon_WDM_Pulse1 (pulse, wdm_desc);
    Aon_Pulse_Packet_Send_Delayed (pkptr, 2, delay);
  }
  if (port_index == 1)
  {
    Aon_WDM_Pulse2 (pulse, wdm_desc);
    Aon_Pulse_Packet_Send_Delayed (pkptr, 2, delay);
  }
  if (port_index == 2)
  {
    new_pulse = Aon_Pulse_Copy (pulse);
    Aon_WDM_Pulse1 (new_pulse, wdm_desc);
    Aon_Pulse_Packet_Send_Delayed (pkptr, 0, delay);
    Aon_WDM_Pulse2 (new_pulse, wdm_desc);
    pkptr = Aon_Pulse_Packet_Create (new_pulse);
    Aon_Pulse_Packet_Send_Delayed (pkptr, 2, delay);
  }
  else
  {
    noise = Aon_Noise_Packet_Get (pkptr);
    noise->power = Aon_Port_Noise_In_Handle (noise_in [port_index], noise);
    if (port_index == 0)
    {
      Aon_WDM_Noise1 (noise, wdm_desc);
      Aon_Pulse_Packet_Send_Delayed (pkptr, 2, delay);
    }
    if (port_index == 1)
    {
      Aon_WDM_Noise2 (new_noise, wdm_desc);
      Aon_Pulse_Packet_Send_Delayed (pkptr, 2, delay);
    }
    if (port_index == 2)
    {
      new_noise = Aon_Noise_Copy (noise);
      Aon_WDM_Noise1 (noise, wdm_desc);
      Aon_Pulse_Packet_Send_Delayed (pkptr, 0, delay);
      pkptr = Aon_Pulse_Packet_Create (new_noise);
      Aon_WDM_Noise2 (noise, wdm_desc);
      Aon_Pulse_Packet_Send_Delayed (pkptr, 1, delay);
    }
  }
}
```
### transition steady -> steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_1</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.13 aon_rou

**Process Model Attributes**

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>promoted</td>
<td>integer</td>
<td>2 (unitless)</td>
</tr>
<tr>
<td>FSR</td>
<td>promoted</td>
<td>double</td>
<td>0.5 (THz)</td>
</tr>
<tr>
<td>Attenuation</td>
<td>promoted</td>
<td>double</td>
<td>0.0 (dB)</td>
</tr>
<tr>
<td>Extinction Ratio</td>
<td>promoted</td>
<td>double</td>
<td>16 (dB)</td>
</tr>
<tr>
<td>Delay</td>
<td>promoted</td>
<td>double</td>
<td>10 (ps)</td>
</tr>
</tbody>
</table>

**Header Block**

```c
/* AON Model Suite */
/* Greg Campbell */
#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_rou.ex.h"
```

**State Variable Block**

```c
/* State variable */
AonT_Rou_Desc* rou_desc;
AonT_Port_Noise_Out_Ptr* noise_out;
AonT_Port_Noise_In_Ptr* noise_in;
```

**Temporary Variable Block**

```c
int event_type;
Packet* pkptr;
Packet* new_pkptr;
int port_index;
int type;
AonT_Pulse* pulse;
AonT_Pulse* new_pulse;
AonT_Noise* noise;
AonT_Noise* new_noise;
Objid my_id;
int i;
double loss;
int N;
double FSR;
double k;
double delay;
int out_port;
```

**Forced state**

<table>
<thead>
<tr>
<th>attribute</th>
<th>init</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>enter execs</td>
<td></td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td></td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
<td></td>
</tr>
</tbody>
</table>
```

**enter exec** init

```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
Aon_Simulation_Data_Get();

/* Determine module specific attributes. */
op_ima_obj_attr_get (my_id, "N", &N);
op_ima_obj_attr_get (my_id, "FSR", &FSR);
10 op_ima_obj_attr_get (my_id, "Attenuation", &loss);
op_ima_obj_attr_get (my_id, "Extinction Ratio", &k);
op_ima_obj_attr_get (my_id, "Delay", &delay);

/* Initialize variables. */
15 rou_desc = Aon_Rou_Create (N, FSR, loss, k, delay);

noise_out = (AonT_Port_Noise_Out_Ptr) malloc (2 * N * sizeof (AonT_Port_Noise_Out_Ptr));
nhise_i = (AonT_Port_Noise_In_Ptr) malloc (2 * N * sizeof (AonT_Port_Noise_In_Ptr));

for (i = 0; i < 2*N; i++)
{ (*noise_out + i) = Aon_Port_Noise_Out_Create 0;
(*noise_in + i) = Aon_Port_Noise_In_Create();
}
```

```

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>tr_0</td>
<td>string</td>
<td>tr</td>
</tr>
<tr>
<td>condition</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
<td>toggle</td>
<td>spline</td>
</tr>
</tbody>
</table>

**unforced state** steady

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>steady</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

```

```
/* Get event */
event_type = op_intrpt_type();

if (event_type == CPC_INTRPT_SELF)
5 {
    /* Do module specific actions. */
}
```

```
if (event_type == OPC_INTRPT_STRM)
{
    port_index = op_intrpt_strm();
    if (port_index >= 2*roudesc->N)
        op_sim_end(‘Invalid port index’, , , ,);  
    pkptr = op_pk_get(port_index);
    type = Aon_Event_Packet_Type(pkptr);

    if (type == AONC_PKT_PULSE)  
        pulse = Aon_Pulse_Packet_Get(pkptr);
        for (i = 1; i < rou_desc->N; i++)
        {
            new_pulse = Aon_Pulse_Copy(pulse);
            Aon_Rou_Pulse(new_pulse, rou_desc, i);
            new_pkptr = Aon_Pulse_Packet_Create(new_pulse);

            if (port_index < rou_desc->N)
                out_port = ((port_index + i) % rou_desc->N) + rou_desc->N;
            else
                out_port = (port_index + i) % rou_desc->N;
        }  
        Aon_Pulse_Packet_Send_Delayed(new_pkptr, out_port, rou_desc->delay);

    Aon_Rou_Pulse(pulse, rou_desc, 0);
    out_port = (port_index + rou_desc->N) % (2 * rou_desc->N);
    Aon_Pulse_Packet_Send_Delayed(pkptr, out_port, rou_desc->delay);
}
else
{
    noise = Aon_Noise_Packet_Get(pkptr);
    noise->power = Aon_Port_Noise_In_Handle(*noise_in + port_index), noise);
    for (i = 1; i < rou_desc->N; i++)
    {
        new_noise = Aon_Noise_Copy(noise);
        Aon_Rou_Noise(new_noise, rou_desc, i);
        new_pkptr = Aon_Noise_Packet_Create(new_noise);

        if (port_index < rou_desc->N)
            out_port = ((port_index + i) % rou_desc->N) + rou_desc->N;
    }
else
{
    out_port = (port_index + i) % rou_desc->N;
}
Aon_Port_Noise_Out_Handle_Dif_Reuse
(*(noise_out + out_port), new_pkptr,
    out_port, rou_desc->delay);
}
Aon_Rou_Noise (noise, rou_desc, 0);
out_port = (port_index + rou_desc->N) %
    (2 * rou_desc->N);
Aon_Port_Noise_Out_Handle_Dif_Reuse
(*(noise_out + out_port), new_pkptr,
    out_port, rou_desc->delay);
}

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
A.14 aonprobe


All Optical Network Model Suite

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eye width</td>
<td>promoted</td>
<td>double</td>
<td>100 (ps)</td>
</tr>
<tr>
<td>coherent</td>
<td>promoted</td>
<td>integer</td>
<td>0 (N/A)</td>
</tr>
<tr>
<td>Signal ID</td>
<td>promoted</td>
<td>integer</td>
<td>0 (N/A)</td>
</tr>
</tbody>
</table>

Header Block

/* AON Model Suite */
/* Greg Campbell */
#include <cmath>
#include <aon_base.ex.h>
#include <aonrcv.ex.h>

State Variable Block

/* State variable */
AonT_Port_Pulse* port;
AonT_Port_Noise_In* noise_in;
double old_time;
5 double rcv_noise;
double sim_duration;
int pulse_num;
AonTRcv_Desc rcv-desc;

Temporary Variable Block

int event_type;
Packet* pexpr;
int port_index;
int type;
5 AonT_Pulse *pulse;
AonT_Pulse *pulse_copy;
AonT_Noise *noise;
Objid my_id;

forced state Init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below,)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

enter execs Init

/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
5 Aon_Simulation_Data_Get();

138
/* Determine module specific attributes. */
op_ima_sim_attr_get (OPC_DMA_DOUBLE, "duration", &sim_duration);
op_ima_obj_attr_get (my_id, "eye_width", &rcv_desc.eye_width);
op_ima_obj_attr_get (my_id, "coherent", &rcv_desc.coherent);
op_ima_obj_attr_get (my_id, "signal ID", &rcv_desc.signal_id);

/* Initialize variables. */
rcv_desc.eye_origin = -1.0;

/* Create input port. */
port = Aon_Port._Pulse_Create();
noise_in = Aon_Port_Noise_In_Create();

/* Set time of last update to 0.0. */
old_time = 0.0;

/* Set current noise level to 0.0. */
rcv_noise = 0.0;

/* Schedule an interrupt to finish out duration. */
op_intrptschedulesel (sim.duration - 1E-9, 0);

/* Set pulse number to 0. */
pulse_num = 0;

if (rcv_desc.eyeorigin == -1.0)
    rcv_desc.eye_origin = opsim-time 0 + AonI_Duration / 2.0 + rcvdesc.eye_width / 2.0;

/* Get event */
event_type = opintrpt_type 0;
if (eventtype == OPC_INTRPT_SELF)
    /* Do module specific actions. */

---

<table>
<thead>
<tr>
<th>transition</th>
<th>Init -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_0</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unforced state</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>steady</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
</tr>
</tbody>
</table>

```c
if (event_type == OPC_INTRPT_SELF)
```
Aon_Rcv_Update (port, rcv_noise, old_time,
        op_sim_time (); &rcv_desc);
    old_time = op_sim_time ();
    }

if (event_type == OPC_INTRPT_STRM)
{
    port_index = op_intrpt_strm ();
    if (port_index != 0)
        op_sim_end ('Invalid port index', **, **, **);

    pkptr = op_pk_get (port_index);

type = Aon_Event_Packet_Type (pkptr);

    if (type == AONCPKT_PULSE)
    {
        Aon_Rcv_Update (port, rcv_noise, old_time,
            op_sim_time (); &rcv_desc);
        old_time = op_sim_time ();
        pulse = Aon_Pulse_Packet_Get (pkptr);
        pulse_copy = Aon_Pulse_Copy (pulse);
        Aon_Rcv_Pulse (pulse_copy, pulse_num);
        Aon_Port_Pulse_Append (port, pulse_copy);
    }
    else
    {
        noise = Aon_Noise_Packet_Get (pkptr);
        rcv_noise += Aon_Port_Noise_InHandle (noise.in, noise);
        (* (noise.in->noise_array + noise->freq_bin)) = noise->power;
        Aon_Rcv_Update (port, rcv_noise, old_time,
            op_sim_time (); &rcv_desc);
        old_time = op_sim_time ();
    }

    op_pk_send (pkptr, 0);

    }

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
**Process Model Report: aon_rcv**

**All Optical Network Model Suite**

---

### Process Model Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eye width</td>
<td>promoted</td>
<td>double</td>
<td>100 (ps)</td>
</tr>
<tr>
<td>coherent</td>
<td>promoted</td>
<td>integer</td>
<td>0 (N/A)</td>
</tr>
<tr>
<td>Signal ID</td>
<td>promoted</td>
<td>integer</td>
<td>0 (N/A)</td>
</tr>
</tbody>
</table>

---

### Header Block

```c
/* AON Model Suite */
/* Greg Campbell */

#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_rcv.ex.h"
```

---

### State Variable Block

```c
/* State variable */
AonT_Port_Pulse* port;
AonT_Port_Noise_In* noise_in;
double old_time;
5 double rcv_noise;
double sim_duration;
int pulse_num;
AonT_Rcv_Desc rcvdesc;
```

---

### Temporary Variable Block

```c
int event_type;
Packet* pkptr;
int port_index;
int type;
5 AonT_Pulse *pulse;
AonT_Noise *noise;
Objid my_id;
```

---

### forced state init

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>type</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>init</td>
<td>string</td>
<td>st</td>
</tr>
<tr>
<td>enter execs</td>
<td>(See below.)</td>
<td>textlist</td>
<td>(See below.)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(empty)</td>
<td>textlist</td>
<td>(empty)</td>
</tr>
<tr>
<td>status</td>
<td>forced</td>
<td>toggle</td>
<td>unforced</td>
</tr>
</tbody>
</table>

---

```c
/* Determine unique ID. */
my_id = op_id_self();

/* Determine simulation data. */
5 Aon_Simulation_Data_Get();
op_ima_sim_attr_get (OPC_IMA_DOUBLE, "duration", &sim_duration);
```
/* Determine module specific attributes. */
op_ima_obj_attr_get (my_id, 'eye width', &(rcvdesc.eyewidth));
/* Initialize variables. */
rcv_desc.eye_origin = -1.0;
/* Create input port. */
port = Aon_Port_Pulse_Create();
nnoise_in = Aon_Port_Noise_In_Create();
/* Set time of last update to 0.0. */
old_time = 0.0;
/* Set current noise level to 0.0. */
rcv_noise = 0.0;
/* Schedule an interrupt to finish out duration. */
op_intrpt_schedule_self (sim_duration - 1E-9, 0);
/* Set pulse number to 0. */
pulse_num = 0;

<table>
<thead>
<tr>
<th>transition</th>
<th>init -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_0</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unforced state</th>
<th>steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>steady</td>
</tr>
<tr>
<td>enter execs</td>
<td>(empty)</td>
</tr>
<tr>
<td>exit execs</td>
<td>(See below.)</td>
</tr>
<tr>
<td>status</td>
<td>unforced</td>
</tr>
</tbody>
</table>

exit exec steady

if (rcv_desc.eye_origin == -1.0)
{
    rcv_desc.eye_origin = op_sim_time() + AonI_Duration / 2.0 +
                          rcv_desc.eye_width / 2.0;
}
/* Get event */
event_type = op_intrpt_type();
if (event_type == OPC_INTRPT_SELF)
{
    /* Do module specific actions. */
    Aon_Rcv_Update (port, rcv_noise, old_time, ...
All Optical Network Model Suite

```c
op_sim_time(), &rcv_desc);
old_time = op_sim_time();
}
if (eventtype == OPC_INTRPT_STMT)
{
    port_index = op_intrpt_strm();
    if (port_index != 0)
        op_sim_end("Invalid port index", ",");
    pkptr = op_pk_get(port_index);
    type = Aon_Event_Packet_Type(pkptr);
    if (type == AONC_PKT_PULSE)
        Aon_Rcv_Update(port, rcv._noise, old_time,
                       op_sim_time(), &rcv_desc);
    oldtime = op_sim_time();
    pulse = AonPulsePacketGet(pkptr);
    Aon_Rcv_Pulse(pulse, pulse_num);
    Aon_Port_Pulse_Append(port, pulse);
    Aon_Pulse_Packet_Destroy(pkptr);
    else
        noise = Aon_Noise_Packet_Get(pkptr);
        rcv_noise += Aon_PortNoise_In_Handle(noise_in, noise);
        (*(noise_in->noise_array + noise->freq_bin)) = noise->power;
        Aon_Noise_Packet_Destroy(pkptr);
        Aon_Rcv_Update(port, rcv_noise, oldtime,
                       op_sim_time(), &rcv_desc);
        old_time = op_sim_time();
}
```

<table>
<thead>
<tr>
<th>transition</th>
<th>steady -&gt; steady</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>value</td>
</tr>
<tr>
<td>name</td>
<td>tr_1</td>
</tr>
<tr>
<td>condition</td>
<td></td>
</tr>
<tr>
<td>executive</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>RGB333</td>
</tr>
<tr>
<td>drawing style</td>
<td>spline</td>
</tr>
</tbody>
</table>
Appendix B: Supporting Code

This appendix contains all of the supporting code in the AON Model Suite. Detailed information about the concepts behind the models including model attributes can be found in chapter four. The following supporting code files are included:

- `aon_xmt.ex.h` and `aon_xmt.ex.c` Support for the Transmitter models.
- `aon_fib.ex.h` and `aon_fib.ex.c` Support for the Fiber model.
- `aon_fbc.ex.h` and `aon_fbc.ex.c` Support for the Fused Biconical Coupler model.
- `aon_stc.ex.h` and `aon_stc.ex.c` Support for the Star Coupler model.
- `aon_amp.ex.h` and `aon_amp.ex.c` Support for the Amplifier model.
- `aon_ase.ex.h` and `aon_ase.ex.c` Support for the ASE Filter model.
- `aon_fab.ex.h` and `aon_fab.ex.c` Support for the Fabry-Perot Filter model.
- `aon_mzf.ex.h` and `aon_mzf.ex.c` Support for the Mach-Zehnder Filter model.
- `aon_wdm.ex.h` and `aon_wdm.ex.c` Support for the WDM model.
- `aon_rou.ex.h` and `aon_rou.ex.c` Support for the Router model.
- `aon_rcv.ex.h` and `aon_rcv.ex.c` Support for the Receiver and Probe models.
- `aon_lin.ex.h` and `aon_lin.ex.c` Support for linear transfer functions.
- `cmath.ex.h` and `cmath.ex.c` Support for complex mathematics.
- `aon_ps.ex.h`, `aon_propdel.ps.c`, `aon_proprcv.ps.c`, `aon_txdel.ps.c` `aon_txrvc.ps.c` Pipeline stage models
B.1 Transmitter Support Code

All of the transmitter process models use the same basic pulse shape generation functions. These functions and their accompanying structures are found in aon_xmt.ex.h and aon_xmt.ex.c.

**aon_xmt.ex.h**

```c
/* Greg Campbell */
/* AON Model Suite */
/* aon_xmt.ex.h */
/* Transmitters */

/**** Typedefs ****/

typedef struct
{
  double t0;
  double peak_power;
  int m;
  double chirp;
} AonT_Xmt_Gaussian;

typedef struct
{
  double t0;
  double peak_power;
  double chirp;
} AonT_Xmt_Seich;

/**** Function Prototypes ****/

CmathT_Complex* Aon_Xmt_Gaussian (AonT_Xmt_Gaussian *gaussian);
CmathT_Complex* Aon_Xmt_Seich (AonT_Xmt_Seich *sech);
```

**aon_xmt.ex.c**

```c
/* Greg Campbell */
/* AON Model Suite */
/* aon_xmt.ex.c */

#include "/lidsfs/usr/local3/opnet-2.5-sol/sys/include/opnet.h"
#include "math.h"
#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_xmt.ex.h"

CmathT_Complex* Aon_Xmt_Gaussian (AonT_Xmt_Gaussian *gaussian)
{
  CmathT_Complex *shape;
  int i;
```
double t_over_t0;

shape = (CmathT_Complex*) malloc (AonI_Len * sizeof (CmathT_Complex));

for (i = 0; i < AonI_Len; i++)
{
    t_over_t0 = (double)(i - AonI_Len / 2) * AonI_Duration / (AonI_Len * gaussian->tO);
    (shape + i)->r = sqrt(gaussian->peak_power) * exp ((-0.5) * pow (t_over_t0, (2*gaussian->m)));
    if (gaussian->chirp != 0.0)
    {
        (shape + i)->theta = ((-0.5)* pow (t_over_t0, (2*gaussian->m)));
    }
    else
    {
        (shape + i)->theta = 0.0;
    }
}

return (shape);
}

CmathT_Complex* Aon_Xmt_Sec (AonT_Xmt_Sech *sech)
{
    CmathT_Complex *shape;
    int i;
    double t_over_t0;

    shape = (CmathT_Complex*) malloc (AonI_Len * sizeof (CmathT_Complex));

    for (i = 0; i < AonI_Len; i++)
    {
        t_over_t0 = (double)(i - AonI_Len / 2) * AonI_Duration / (AonI_Len * sech->tO);
        (shape + i)->r = sqrt(sech->peak_power) * (1.0 / cosh (t_over_t0));
        if (sech->chirp != 0.0)
        {
            (shape + i)->theta = -0.5 * sech->chirp * t_over_t0 * t_over_t0;
        }
        else
        {
            (shape + i)->theta = 0.0;
        }
    }

    return (shape);
}
B.2 Optical Fiber Support Code

The optical fiber *process model* and the optical fiber model in the links use the same basic pulse propagation functions. These functions and their accompanying structures are found in `aon_fib.ex.h` and `aon_fib.ex.c`.

**aon_fib.ex.h**

```c
/* Greg Campbell */
/* AON Model Suite */

/**** Defines ****/
#define AONC_FIB_DISPERSION 0
#define AONC_FIB_BPM 1
#define AONC_FIB_KPM 2
#define AONC_FIB_T_RAMAN_DEF 0.0005
#define AONC_FIB_F1_DEF 192.0
#define AONC_FIB_F2_DEF 225.0
#define AONC_FIB_B1_F1_DEF 4875.0
#define AONC_FIB_B1_F2_DEF 4871.7
#define AONC_FIB_B2_F1_DEF -20.0
#define AONC_FIB_B2_F2_DEF 0.0
#define AONC_FIB_B3_DEF 0.0
#define AONC_FIB_ALPHA_DEF 0.2
#define AONC_FIB_LENGTH_DEF 100
#define AONC_FIB_GRANULARITY_DEF 10
#define AONC_FIB_A_EFF_DEF 65.0
#define AONC_FIB_N2_DEF 3.2E-16
#define AONC_FIB_GRMAX_DEF 1E-16
#define AONC_FIB_FRMAX_DEF 12.0

/**** Global Variables ****/
#ifdef AON_FIB_DECS
List AonI_Fib_List;
int AonI_Fib_List_Init = 0;
#else
extern List AonI_Fib_List;
extern int AonI_Fib_List_Init;
#endif

/**** Typedefs ****/
typedef struct
{
    double T_Raman;
    double f1;
    double f2;
    double B1_f1;
    double B1_f2;
    double B2_f1;
    double B2_f2;
    double B3;
    double alpha;
    double Length;
    double granularity;
    double A_eff;

```
typedef struct
{
    int     link_objid;
    AonT_Fib_Desc* fib_desc;
    int     xmtl_objid;
    AonT_Port_Pulse* port1;
    double  last_time1;
    int     xmt2_objid;
    AonT_Port_Pulse* port2;
    double  last_time2;
} AonT_Fib_Link;

typedef struct
{
    double  time;
    int     type;
    double  d_time;
    int     pulse1;
    int     pulse2;
    int     offset;
    double  length;
} AonT_Fib_Event;

int     aon_fib_event_comp (AonT_Fib_Event* aptr, AonT_Fib_Event* bptr);
AonT_Pulse*Aon_Fib_Exit_Pulse (AonT_Port_Pulse *port, AonT_Fib_Desc* fib_desc,
    double  time);
void     Aon_Fib_Prop_Port (AonT_Port_Pulse *port, AonT_Fib_Desc* fib_desc,
    double  last_time, double  time);
int     aon_fib_events_xpm_add (List* event_list, AonT_Port_Pulse* port,
    AonT_Fib_Desc* fib_desc, int pulsel, int pulse2,
    double  last_time, double  time);
void     aon_fib_event_dispersion_add (List* event_list, int pulse_index,
    double  time, double  length);
void     aon_fib_event_spm_add (List* event_list, int pulse_index,
    double  time, double  length);
void     aon_fib_event_xpm_add (List* event_list, int pulsel, int pulse2,
    double  time, double  d_time, int offset);
void     aon_fib_event_process (AonT_Fib_Event* event, AonT_Port_Pulse* port,
    AonT_Fib_Desc* fibdesc);
void     Aon_Fib_Pulse_Insert (Packet* pkptr);
void     Aon_Fib_Pulse_Remove (Packet* pkptr);
void     Aon_Fib_Prop_Self (AonT_Pulse* pulse, AonT_Fib_Desc* fib_desc,
    double  length);
void     Aon_Fib_Dispersion (AonT_Pulse* pulse, AonT_Fib_Desc* fib_desc,
    double  h);
void     Aon_Fib_SPM (AonT_Pulse* pulse, AonT_Fib_Desc* fib_desc, double  h);
void     Aon_Fib_XPM (AonT_Pulse* pulse1, AonT_Pulse* pulse2,
    AonT_Fib_Desc* fib_desc, double  d_time, int offset);
double   Aon_Fib_Gamma (AonT_Pulse* pulse, AonT_Fib_Desc* fib_desc);
double   Aon_Fib_B2 (double  freq, AonT_Fib_Desc* fib_desc);
double   Aon_Fib_B1 (double  freq, AonT_Fib_Desc* fib_desc);
double   Aon_Fib_Delay (AonT_Pulse* pulse, AonT_Fib_Desc* fib_desc);
AonT_Fib_Link* Aon_Fib_Link_Attr_Get (Objid link_objid);
#include <math.h>
#include "cmath.h"
#include /lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h
#include "aon_base.ex.h"#define AON_FIB_DECS
#include "aon_fib.ex.h"
#define AONC_FIBC 3.0E8

int aon_fib_event_comp (AonT_Fib_Event* aptr, AonT_Fib_Event* bptr)
{
    if (aptr->time < bptr->time)
        return (1);
    else if (aptr->time > bptr->time)
        return (-1);
    else if (aptr->type == AONC_FIB_DISPERSION)
        return (1);
    else
        return (-1);
}

AonT_Pulse* Aon_Fib_Exit_Pulse (AonT_Port_Pulse *port, AonT_Fib_Desc *fib_desc,
                                double time)
{
    AonT_Pulse* pulse;
    AonT_Pulse* out_pulse;
    AonT_Port_Entry* port_entry;
    int num_pulse, i;
    double dist, max_dist;
    max_dist = -1.0;
    num_pulse = op_prg_list_size (&(port->input));
    for (i = 0; i < num_pulse; i++)
    {
        port_entry = (AonT_Port_Entry*) op_prg_list_access (&(port->input), i);
        pulse = port_entry->pulse;
        dist = (time - port_entry->entry_time) / (Aon_FibB1 (pulse->freq, fib_desc));
        if (dist > max_dist)
        {
            max_dist = dist;
            out_pulse = pulse;
        }
    }
    return (out_pulse);
}

void Aon_Fib_Prop_Port (AonT_Port_Pulse *port, AonT_Fib_Desc *fib_desc,
                        double last_time, double time)
{
    int num_pulse;
    AonT_Port_Entry* port_entry;
    int i, j;
    double d_time;
static List event_list;
static int event_list_init = 0;
AonT_Pulse* pulse;
double L_nl, L_d;
double width;
double max_length, length;
int steps;
double step_time;
AonT_Fib_Event* event;

if (event_list_init == 0)
{
    op_prg_list_init (&event_list);
    event_list_init = 1;
}

/* Perform Split-Step Fourier Method. */
num_pulse = op_prg_list_size (&(port->input));
d_time = time - last_time;

for (i = 0; i < num_pulse; i++)
{
    port_entry = (AonT_Port_Entry*) op_prg_list_access (&(port->input), i);
    pulse = port_entry->pulse;

    /* First determine length scales for the pulse. */
    L_nl = 1.0 / (Aon_Fib_Gamma (pulse, fib_desc) * pulse->peak_power);
    if (Aon_FibB2 (pulse->freq, fib_desc) != 0.0)
    {
        width = Aon_Pulse_Width (pulse);
        L_d = (pow (width, 2.0) / fabs (Aon_Fib_B2 (pulse->freq, fib_desc)));
    } else
    { L_d = max_length * fib_desc->granularity;
    }
    if (L_d < L_nl)
    {
        length = L_nl / fib_desc->granularity;
    } else
    {
        length = L_d / fib_desc->granularity;
    }

    max_length = d_time / Aon_Fib_B1 (pulse->freq, fib_desc);
    steps = ceil (max_length / length);
    length = max_length / steps;
    step_time = d_time / steps;

    aon_fib_event_dispersion_add (&event_list, i, 0.0, length / 2.0);
    for (j = 0; j < steps; j++)
    {
        aon_fib_event_spm_ad (event_list, i, (j * step_time), length);
        if (j < (steps - 1))
        {
            aon_fib_event_dispersion_add (&event_list, i,
                ((j * step_time) + (step_time / 2.0)), length);
        }
    }

    aon_fib_event_dispersion_add (&event_list, i, d_time - step_time / 2.0,
        length / 2.0);
}
/** Create XPM and SRS events. */
for (i = 0; i < num_pulse; i++)
{
    for (j = i; j < num_pulse; j++)
    {
        aon_fib_events_xpm_add (&event_list, port, fib_desc,
            i, j, last_time, time);
    }
}

/*/ Sort events by time. */
op_prg_list_sort (&event_list, aon_fib_event_comp);

/*/ Process events. */
while (opprg_list_size (&event_list) != 0)
{
    event = (AonT_Fib_Event*) op_prg_list_remove
            (&event_list, OPC_LISTPOS_HEAD);
    aon_fib_event_process (event, port, fib_desc);
    free (event);
}

int 
aon_fib_events_xpm_add (List* event_list, AonT_Port_Pulse* port,
    AonT_Fib_Desc* fib_desc, int pulselind, int pulse2ind, double last_time,
    double time)
{
    AonT_Port_Entry* port_ent1;
    AonT_Port_Entry* port_ent2;
    AonT_Pulse* pulse1;
    AonT_Pulse* pulse2;
    double timel, time2;
    double Bl_1, Bl_2;
    double L_nl, length;
    int i, offset, steps;
    int intervals;
    double int1, int2, overlap, d_time;
    double start_time, end_time, cur_time, next_time;

    port_ent1 = (AonT_Port_Entry*) op_prg_list_access
            (&(port->input), pulselind); 
    port_ent2 = (AonT_Port_Entry*) op_prg_list_access
            (&(port->input), pulse2ind); 

    pulse1 = port_ent1->pulse; 
    pulse2 = port_ent2->pulse; 
    timel = port_ent1->entry_time; 
    time2 = port_ent2->entry_time; 

    Bl_1 = Aon_Fib_Bl (pulse1->freq, fib_desc); 
    Bl_2 = Aon_Fib_Bl (pulse2->freq, fib_desc); 

    if (Bl_1 == Bl_2)
    { /* Degenerate group velocities. */
        L_nl = 1.0 / (Aon_Fib_Gamma (pulse1, fib_desc) *
                        (pulse1->peak_power + pulse2->peak_power));

        length = (time - last_time) / Bl_1;
        steps = ceil (length / L_nl);

        if (timel > time2) /* Pulse 2 is ahead. */
        { .......

152
if (time1 > (time2 + AonI_Duration))
    {
    /* No overlap. */
    return (1);
    }
else
    {
    offset = (time1 - time2) / (AonI_Duration / AonI_Len);
    for (i = 0; i < steps; i++)
    {
    /* This procedure requires that offset be positive*/
    /* and that pulse 2 be ahead of pulse 1. */
    aon_fib_event_xpm_add (event_list, pulselind, pulse2ind,
               (double)(last_time + i * (time - last_time) / steps),
               (double)((time - last_time) / steps), offset);
    }
    }
else /* Pulse 1 is ahead. */
    {
    if (time2 > (time1 + AonI_Duration))
    {
    /* No overlap. */
    return (1);
    }
else
    {
    offset = (time2 - timel) / (AonI_Duration / AonI_Len);
    for (i = 0; i < steps; i++)
    {
    /* This procedure requires that offset be positive*/
    /* and that pulse 2 be ahead of pulse 1. */
    aon_fib_event_xpm_add (event_list, pulse2ind, pulselind,
               (double)(last_time + i * (time - last_time) / steps),
               (double)((time - last_time) / steps), offset);
    }
    }
    }
    return (1);
}

/* int1 is when the leading edge of pulse 1 meets the trailing*/
/* edge of pulse 2. The leading edge of 1 entered the fiber*/
/* at port_ent1's time. The trailing edge of 2 entered the */
/* fiber at port_ent2's time + AonI_Duration. */
int1 = (B1_2 * timel - B1_1 * (time2 + AonI_Duration)) / (B1_2 - B1_1);

/* int2 is when the leading edge of pulse 2 meets the trailing*/
/* edge of pulse 1. */
int2 = (B1_1 * time2 - B1_2 * (timel + AonI_Duration)) / (B1_1 - B1_2);

/* overlap is when the leading edge of pulse 1 meets the */
/* leading edge of pulse 2. */
overlap = (B1_2 * timel - B1_1 * time2) / (B1_2 - B1_1);

d_time = fabs (int1 - int2) / ((double)(2 * AonI.Len - 1));
if (int1 < int2)
    {
    /* Pulse 2 is slower than pulse 1.*/
    if (((int2 < last_time) || (int1 > time))}
if (last_time > int1)
    start_time = last_time;
else
    start_time = int1;

if (time < int2)
    end_time = time;
else
    end_time = int2;

cur_time = start_time;
while (cur_time < end_time)
{
    intervals = floor ((cur_time - int1) / d_time);
    next_time = int1 + (intervals + 1) * d_time;
    if (next_time > end_time)
        next_time = end_time;

    offset = floor ((cur_time - overlap) / d_time);
    if (offset > 0)
    { /* This procedure requires that offset be positive*/
      /* and that pulse 2 be ahead of pulse 1. */
        aon_fib_event_xpm_add (event_list, pulse2ind, pulselind,
                              cur_time, next_time - cur_time, offset);
    }
    else
    { /* This procedure requires that offset be positive*/
      /* and that pulse 2 be ahead of pulse 1. */
        aon_fib_event_xpmadd (event_list, pulselind, pulse2ind,
                              cur_time, next_time - cur_time, ((-1)*offset));
    }

    cur_time += d_time;
}
else /* Pulse 1 is slower than pulse 2.*/
{
    if (((int1 < last_time) || (int2 > time))
    { /* No overlap. */
        return (1);
    }

    if (last_time > int2)
        start_time = last_time;
    else
        start_time = int2;

    if (time < int1)
        end_time = time;
    else
        end_time = int1;

cur_time = start_time;
while (cur_time < end_time)
{
    intervals = floor ((cur_time - int2) / d_time);
    next_time = int2 + (intervals + 1) * d_time;
    next_time = end_time;
}
if (next_time > end_time)
    next_time = end_time;

offset = floor ((cur_time - overlap) / d_time);
if (offset > 0)
{
    /* This procedure requires that offset be positive*/
    /* and that pulse 2 be ahead of pulse 1. */
    aon_fib_event_xpm_add (event_list, pulselind, pulse2ind, 
                           cur_time, next_time - cur_time, offset);
}
else
{
    /* This procedure requires that offset be positive*/
    /* and that pulse 2 be ahead of pulse 1. */
    aon_fib_event_xpm_add (event_list, pulse2ind, pulselind, 
                           cur_time, next_time - cur_time, ((-1)*offset));
}

cur_time += d_time;
}

void
aon_fib_event_dispersion_add (List* event_list, int pulse_index, double time, 
                               double length)
{
    AonT_Fib_Event* event;

    event = (AonT_Fib_Event*) malloc (sizeof (AonT_Fib_Event));
    event->time = time;
    event->type = AONC_FIB_DISPERSION;
    event->d_time = 0.0;
    event->pulsel = pulse_index;
    event->pulse2 = 0;
    event->offset = 0;
    event->length = length;

    op_prg_list_insert (event_list, (void*) event, OPC_LISTPOS_TAIL);
}

void
aon_fib_event_spm_add (List* event_list, int pulse_index, double time, 
                        double length)
{
    AonT_Fib_Event* event;

    event = (AonT_Fib_Event*) malloc (sizeof (AonT_Fib_Event));
    event->time = time;
    event->type = AONC_FIB_SPM;
    event->d_time = 0.0;
    event->pulsel = pulse_index;
    event->pulse2 = 0;
    event->offset = 0;
    event->length = length;

    op_prg_list_insert (event_list, (void*) event, OPC_LISTPOS_TAIL);
}

void
aon_fib_event_xpm_add (List* event_list, int pulsel, int pulse2, double time, 
                        double d_time, int offset)
{ 
AonT_Fib_Event* event;

event = (AonT_Fib_Event*) malloc (sizeof (AonT_Fib_Event));

event->time = time;
event->type = AONC_FIB_XPM;
event->d_time = d_time;
event->pulsel = pulsel;
event->pulse2 = pulse2;
event->offset = offset;
event->length = 0.0;

op_prg_list_insert (event_list, (void*) event, OPC_LISTPOS_TAIL);
}

void aon_fib_event_process (AonT_Fib_Event* event, AonT_Port_Pulse* port,
AonT_FibDesc* fibdesc)
{
AonT_Pulse* pulse;
AonT_Pulse* pulse2;
pulse = (AonT_Pulse*) op_prg_list_access (&(port->input), event->pulsel);

if (event->type == AONC_FIB_DISPERSION)
{
Aon_Fib_Dispersion (pulse, fib_desc, event->length);
}
else if (event->type == AONC_FIB_SPM)
{
Aon_Fib_SPM (pulse, fib_desc, event->length);
}
else if (event->type == AONC_FIB_XPM)
{
pulse2 = (AonT_Pulse*) op_prg_list_access (&(port->input), event->pulse2);

Aon_Fib_XPM (pulse, pulse2, fib_desc, event->d_time, event->offset);
}
}

void Aon_Fib_Pulse_Insert (Packet* pkptr)
{
}

void Aon_Fib_Pulse_Remove (Packet* pkptr)
{
}

void Aon_Fib_Prop_Self (AonT_Pulse *pulse, AonT_Fib_Desc *fib_desc, double max_length)
{
  double L_d;
  double L_nl;
  double cur_length;
  double width;
  double length;
  double length_left;
  double B2_delay;

  cur_length = 0.0;

  L_nl = 1.0 / (Aon_Fib_Gamma (pulse, fib_desc) * pulse->peak_power);
if (Aon_Fib_B2 (pulse->freq, fib_desc) != 0.0)
{
    width = Aon_Pulse_Width (pulse);
    L_d = (pow (width, 2.0) / fabs (Aon_Fib_B2 (pulse->freq, fib_desc)));
} else
    L_d = max_length * fib_desc->granularity;

if (L_d < L_nl)
{
    length = L_nl / fib_desc->granularity;
} else
{
    length = L_d / fib_desc->granularity;
}

if (length > (max_length - cur_length))
    length = max_length - cur_length;

/* Perform dispersion assuming no non-linearity. */
Aon_Fib_Dispersion (pulse, fib_desc, (length/2.0));
length_left = length / 2.0;

while (cur_length < max_length)
{
    printf ("###cur_length = %lf\n", cur_length);
/*###*/Aon_Pulse_Peak_Power (pulse);

    /* Perform non-linearity assuming no dispersion. */
    Aon_Fib_SPM (pulse, fib_desc, length);

    Aon_Pulse_Peak_Power (pulse);
    cur_length += length;

    if (cur_length < max_length)
    {
        L_nl = 1.0 / (Aon_Fib_Gamma (pulse, fib_desc) * pulse->peak_power);
        if (Aon_Fib_B2 (pulse->freq, fib_desc) != 0.0)
        {
            width = Aon_Pulse_Width (pulse);
            L_d = (pow (width, 2.0) / fabs (Aon_Fib_B2 (pulse->freq, fib_desc)));
        } else
            L_d = max_length * fib_desc->granularity;

        if (L_d < L_nl)
            length = L_nl / fib_desc->granularity;
        else
            length = L_d / fib_desc->granularity;

        if (length > (max_length - cur_length))
            length = max_length - cur_length;
    } else
    length = 0.0;

    /* Perform dispersion assuming no non-linearity. */
    Aon_Fib_Dispersion (pulse, fib_desc, length_left + (length/2.0));
length_left = length / 2.0;
}

Aon_Pulse_Peak_Power (pulse);
}

void
Aon_FibDispersion (AonT_Pulse* pulse, AonT_Fib_Desc *fib_desc, double h)
{
    int i;
    CmathT_Complex tmp, d;
    static CmathT_Complex*fft_shape;
    static int fft_init;
    double freq;

    printf ("###dispersion\n");
    if (fft_init == 0)
    {
        fft_shape = (CmathT_Complex*) malloc (AonI_Len * sizeof (CmathT_Complex));
        fft_init = 1;
    }

    cmath_FFT (fft_shape, pulse->shape, AonINu);
    for (i = 0; i < AonI_Len; i++)
    {
        freq = (((i + AonI_Len/2) % AonI_Len) - AonI_Len/2) * 2.0 * CMATH_PI /
                AonI_Duration);
        d.r = exp ((-0.5)*h*fib_desc->alpha);
        d.theta = ((0.5) * h * (Aon_Fib_B2 (pulse->freq, fib_desc)) *
                    pow (2.0 * CMATH_PI * freq, 2.0)) - ((1.0/6.0) * h * fib_desc->B3 *
                    pow (2.0 * CMATH_PI * freq, 3.0));
        cmath_mult (fft_shape + i, fft_shape + i, &d);
    }

    cmath_inv_FFT (pulse->shape, fftshape, AonINu);
}

void
Aon_FibSPM (AonT_Pulse *pulse, AonT_Fib_Desc *fib_desc, double h)
{
    int i;
    double gamma;
    CmathT_Complex A;
    CmathT_Complex oper;
    CmathT_Complex tmpl;
    CmathT_Complex tmp2;
    CmathT_Complex A_inv;
    CmathT_Complex tmp_exp;
    CmathT_Complex tmp_umb;
    CmathT_Complex tmp_middle;
    CmathT_Complex A2A_diff;
    double A2_0, A2_1, A2_2;
    CmathT_Complex A2A_0, A2A_1, A2A_2;

    printf ("###non_linear\n");
    /*cmath_vector_print (pulse->shape, AonI_Len);*/

    gamma = Aon_Fib_Gamma (pulse, fib_desc);

    A2_1 = pow ((pulse->shape + AonI_Len - 1)->r, 2.0);
    A2_2 = pow ((pulse->shape)->r, 2.0);
    A2A_1.r = A2_1*(pulse->shape + AonI_Len - 1)->r;
    A2A_1.theta = (pulse->shape + AonI_Len - 1)->theta;
A2A_2.r = A2_2*(pulse->shape)->r;
A2A_2.theta = (pulse->shape)->theta;

for (i = 0; i < AonI_Len; i++)
{
    A.r = (pulse->shape + i)->r;
    A.theta = (pulse->shape + i)->theta;
    A2_0 = A2_1;
    A2_1 = A2_2;
    A2_2 = pow ((pulse->shape + ((i + 1) % AonI_Len))->r, 2.0);
    A2A_0.r = A2A_1.r;
    A2A_0.theta = A2A_1.theta;
    A2A_1.r = A2A_2.r;
    A2A_1.theta = A2A_2.theta;
    A2A_2.r = A2_2*(pulse->shape + ((i + 1) % AonI_Len))->r;
    A2A_2.theta = (pulse->shape + ((i + 1) % AonI_Len))->theta;
    /* Set tmpl to 2i/wO. */
    tmpl.r = ((2.0) / (pulse->freq * 2.0 * CMATH_PI));
    tmpl.theta = CMATH_PI / 2.0;
    /* Determine 1/A. */
    A_inv.r = 1.0 / (pulse->shape + i)->r;
    A_inv.theta = (-1.0) * (pulse->shape + i)->theta;
    /* Set tmpl to 2i/wOA. */
    cmath_mult (&tmp2, &tmpl, &A_inv);
    /* Determine d/dα of A2A. */
    /* Subtract A2A_0 - A2A_2 because time is in reverse. */
    cmath_sub (&A2A_diff, &A2A_0, &A2A_2);
    A2A_diff.r = A2A_diff.r / (AonI_Duration * 2.0 / (double) AonI_Len);
    /* Set tmp_middle to 2i/wOA * d/dα of A2A. */
    cmath_mult (&tmp_middle, &tmp2, &A2A_diff);
    /* set tmpl to A2 - Tr*d/dα A2 */
    tmpl.r = A2_1 -
        (fib_desc->T_Raman *
            (A2_0 - A2_2) / (AonI_Duration * 2.0 / (double) AonI_Len));
    tmpl.theta = 0.0;
    /* Set tmp_umb to tmp_middle plus tmpl. */
    cmath_add (&tmp_umb, &tmp_middle, &tmpl);
    /* Set tmp1 to i*h*gamma. */
    tmpl.r = gamma * h;
    tmpl.theta = CMATH_PI / 2.0;
    /* Set tmp_exp to D from page 45 in Agrawal. */
    cmath_mult (&tmp_exp, &tmpl1, &tmp_umb);
    /* Set the operator to e**D. */
    oper.r = exp (tmp_exp.r * cos (tmp_exp.theta));
    oper.theta = tmp_exp.r * sin (tmp_exp.theta);
    /* Multiply the pulse envelope by the operator. */
    cmath_mult ((pulse->shape + i), &A, &oper);
}
void Aon_Fib_XPM (AonT_Pulse* pulse1, AonT_Pulse* pulse2, AonT_Fib_Desc* fib_desc, double d_time, int offset)
{
    double length1, length2;
    double gamma1, gamma2;
    double S1_pow, S2_pow;
    double g_raman;
    double raman_amp;
    double raman_power;
    int i;
    /* Determine the interaction length for each pulse sample.*/
    length1 = d_time * Aon_Fib_B1 (pulse1->freq, fib_desc);
    length2 = d_time * Aon_Fib_B1 (pulse2->freq, fib_desc);
    /* Determine the raman gain constant dependent upon the */
    /* difference in pulse carrier frequencies. */
    g_raman = cmath_dB ((-1.0) * fib_desc->grmax * (fabs (pulse1->freq -
        pulse2->freq)) / fib_desc->frmax);
    /* Determine the gamma constant for each pulse. */
    gamma1 = Aon_Fib_Gamma (pulse1, fib_desc);
    gamma2 = Aon_Fib_Gamma (pulse2, fib_desc);
    /* Pulse 2 is always ahead of pulse 1. */
    /* Go through each overlapping sample... */
    for (i = 0; i < (AonT_Len - offset); i++)
    {
        /* Determine the power of the two samples in question.*/
        S1_pow = pow ((pulse1->shape + i + offset)->r, 2.0);
        S2_pow = pow ((pulse2->shape + i)->r, 2.0);
        /* Perform XPM calculation. */
        /* If frequencies are the same, really SPM. */
        if (pulse1->freq == pulse2->freq)
        {
            (pulse1->shape + i + offset)->theta +=
                length1 * S2_pow * 2.0 * gamma1;
            (pulse2->shape + i)->theta += length2 * S1_pow * 2.0 * gamma2;
        }
        else
        {
            (pulse1->shape + i + offset)->theta += length1 * S2_pow * gamma1;
            (pulse2->shape + i)->theta += length2 * S1_pow * 2.0 * gamma2;
        }
        /* Place the sample in a known state, with positive */
        /* amplitude. */
        cmath_principle_val (pulse1->shape + i + offset);
        cmath_principle_val (pulse2->shape + i);
        /* Perform the Raman gain calculations. */
        /* The higher frequency pulse amplifies the lower freq.*/
        if (pulse1->freq > pulse2->freq)
        {
            /* Amplify pulse 2.*/
            /* Determine the raman amplification. */
            raman_amp = exp (length2 * g_raman * S1_pow);
            (pulse2->shape + i)->r = (pulse2->shape + i)->r * g_raman;
            /* Determine the amount of power transferred. */
            raman_power = pow ((pulse2->shape + i)->r, 2.0) - S2_pow;
            /* By conservation, remove power from pulse 1. */
        }
(pulsel->shape + i + offset)->r = sqrt (S1_pow - raman_power);
}
else
{ /* Amplify pulse 1. */
 /* Determine the raman amplification. */
 raman_amp = exp (lengthl * g_raman * S2_pow);
(pulsel->shape + i + offset)->r = (pulsel->shape + i + offset)->r *
g_raman;

/* Determine the amount of power transferred. */
raman_power = pow ((pulsel->shape + i + offset)->r, 2.0) - S1_pow;

/* By conservation, remove power from pulse 1. */
(pulse2->shape + i)->r = sqrt (S2_pow - raman_power);
}
}

double Aon_Fib_Gamma (AonT_Pulse *pulse, AonT_Fib_Desc *fib_desc)
{
    double gamma;
    gamma = fib_desc->n2 * 1E-10 * pulse->freq * 2.0 * CMATH_PI * 1E3 / (AONC_FIB_C * 1E-12 *
             fib_desc->A_eff * 1E-18);
    return (gamma);
}

double Aon_Fib_Delay (AonT_Pulse* pulse, AonT_Fib_Desc* fib_desc)
{
    double delay;
    delay = Aon_Fib_B1 (pulse->freq, fib_desc) * fib_desc->Length;
    return (delay);
}

double Aon_Fib_B2 (double freq, AonT_Fib_Desc *fib_desc)
{
    double B2;
        (fib_desc->f2 - fib_desc->f1);
    return (B2);
}

double Aon_Fib_B1 (double freq, AonT_Fib_Desc *fib_desc)
{
    double B1;
    B1 = fib_desc->B1_f1 + (freq - fib_desc->f1) * (fib_desc->B1_f2 - fib_desc->B1_f1) /
        (fib_desc->f2 - fib_desc->f1);
    return (B1);
}

AonT_Fib_Link* Aon_Fib_Link_Attr_Get (Objid link_objid)
{
AonT_Fib_Link* link;
AonT_Fib_Desc* fib_desc;

link = (AonT_Fib_Link*) malloc (sizeof (AonT_Fib_Link));
link->link_objid = (int) link_objid;
link->fib_desc = (AonT_Fib_Desc*) malloc (sizeof (AonT_Fib_Desc));
fib_desc = link->fib_desc;
link->xmt1_objid = -1;
link->xmt2_objid = -1;

if (op_ima_obj_attr_exists (link_objid, "T Raman") == OPCTRUE)
op_ima_obj_attr_get (link_objid, "T Raman", &(fib_desc->T_Raman));
else
    fib_desc->T_Raman = AONC_FIB_T_RAMAN_DEF;

if (op_ima_obj_attr_exists (link_objid, "freq1") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "freq1", &(fib_desc->f1));
else
    fib_desc->f1 = AONC_FIB_F1_DEF;

if (op_ima_obj_attr_exists (link_objid, "freq2") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "freq2", &(fib_desc->f2));
else
    fib_desc->f2 = AONC_FIB_F2_DEF;

if (op_ima_obj_attr_exists (link_objid, "B1 at freq1") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "B1 at freq1", &(fib_desc->B1_f1));
else
    fib_desc->B1_f1 = AONC_FIB_B1_F1_DEF;

if (op_ima_obj_attr_exists (link_objid, "B1 at freq2") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "B1 at freq2", &(fib_desc->B1_f2));
else
    fib_desc->B1_f2 = AONC_FIB_B1_F2_DEF;

if (op_ima_obj_attr_exists (link_objid, "B2 at freq1") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "B2 at freq1", &(fib_desc->B2_f1));
else
    fib_desc->B2_f1 = AONC_FIB_B2_F1_DEF;

if (op_ima_obj_attr_exists (link_objid, "B2 at freq2") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "B2 at freq2", &(fib_desc->B2_f2));
else
    fib_desc->B2_f2 = AONC_FIB_B2_F2_DEF;

if (op_ima_obj_attr_exists (link_objid, "B3") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "B3", &(fib_desc->B3));
else
    fib_desc->B3 = AONC_FIB_B3_DEF;

if (op_ima_obj_attr_exists (link_objid, "alpha") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "alpha", &(fib_desc->alpha));
else
    fib_desc->alpha = AONC_FIB_ALPHA_DEF;

if (op_ima_obj_attr_exists (link_objid, "Length") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "Length", &(fib_desc->Length));
else
    fib_desc->Length = AONC_FIB_LENGTH_DEF;

if (op_ima_obj_attr_exists (link_objid, "granularity") == OPC_TRUE)
op_ima_obj_attr_get (link_objid, "granularity", &(fib_desc->granularity));
else
    fib_desc->granularity = AONC_FIB_GRANULARITY_DEF;
if (op_ima_obj_attr_exists(link_objid, "A eff") == OPC_TRUE)
    op_ima_obj_attr_get(link_objid, "A eff", &fib_desc->A_eff);
else
    fib_desc->A_eff = AONC_FIB_A_EFF_DEF;

if (op_ima_obj_attr_exists(link_objid, "n2") == OPC_TRUE)
    op_ima_obj_attr_get(link_objid, "n2", &fib_desc->n2);
else
    fib_desc->n2 = AONC_FIB_N2_DEF;

if (op_ima_obj_attr_exists(link_objid, "Grmax") == OPC_TRUE)
    op_ima_obj_attr_get(link_objid, "Grmax", &fib_desc->grmax);
else
    fib_desc->grmax = AONC_FIB_GRMAX_DEF;

if (op_ima_obj_attr_exists(link_objid, "Frmax") == OPC_TRUE)
    op_ima_obj_attr_get(link_objid, "Frmax", &fib_desc->frmax);
else
    fib_desc->frmax = AONC_FIB_FRMAX_DEF;

return (link);
B.3 Fused Biconical Coupler Support Code

The fused biconical coupler process model uses functions that determine the output of a fused biconical coupler due to an incident pulse. These functions and their accompanying structures are found in aon_fbc.ex.h and aon_fbc.ex.c.

**aon_fbc.ex.h**

/* Greg Campbell */
/* AON Model Suite */
/* aon_fbc.ex.h */
/* Fused Biconical Coupler Model support code*/

/**** Typedefs ****/

typedef struct
{
    double r;
    double delta_r;
    double z;
    double a;
} AonT_FBC_Desc;

AonT_FBC_Desc*
Aon_FBC_Create (double r, double delta_r, double z, double a);

void Aon_FBC_Pulsel (AonT_Pulse* pulse, AonT_FBC_Desc* fbc_desc);

void Aon_FBC_Noisel (AonT_Noise* noise, AonT_FBC_Desc* fbc_desc);

int Aon_FBC_Gainl (CmathT_Complex* g, double freq, void* void_fbc_desc);

void Aon_FBC_Pulse2 (AonT_Pulse* pulse, AonT_FBC_Desc* fbc_desc);

void Aon_FBC_Noise2 (AonT_Noise* noise, AonT_FBC_Desc* fbc_desc);

int AonFBCGain2 (CmathT_Complex* g, double freq, void* void_fbc_desc);

**aon_fbc.ex.c**

/* Greg Campbell */
/* AON Model Suite */
/* aon_fbc.ex.c */
/* Fused Biconical Coupler Model support code*/
/* See section 4.3 in thesis document */

#include <cmath.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"
#include "aon_fbc.ex.h"

AonT_FBC_Desc*
Aon_FBC_Create (double r, double delta_r, double z, double a)
{
    AonT_FBC_Desc*      fbc_desc;

fbc_desc = (AonT_FBCDesc*) malloc (sizeof (AonT_FBC_Desc));

fbc_desc->r = r;
fbc_desc->delta_r = delta_r;
fbc_desc->z = z;
fbc_desc->a = cmath_dB (a);

return (fbc_desc);
}

void
Aon_FBC_Pulse1 (AonT_Pulse* pulse, AonT_FBC_Desc* fbc_desc)
{
    Aon_Lin_Pulse (pulse, Aon_FBC_Gain1, (void*) fbc_desc);
}

void
Aon_FBC_Noisel (AonT_Noise* noise, AonT_FBCDesc* fbc_desc)
{
    Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_FBC_Gain1,
                    (void*) fbc_desc);
}

void
Aon_FBC_Pulse2 (AonT_Pulse* pulse, AonT_FBC_Desc* fbc_desc)
{
    Aon_Lin_Pulse (pulse, Aon_FBC_Gain2, (void*) fbc_desc);
}

void
Aon_FBC_Noise2 (AonT_Noise* noise, AonT_FBCDesc* fbc_desc)
{
    Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_FBC_Gain2,
                    (void*) fbc_desc);
}

Aon_FBC_Gain1 (CmathT_Complex* g, double freq, void* void_fbc_desc)
{
    AonT_FBC_Desc*       fbc_desc;
    double               C, F2;
    double               lambda;
    double               alpha;

    fbc_desc = (AonT_FBC_Desc*) void_fbc_desc;

    /* In this procedure the unit for time is picoseconds,*/
     * the unit for distance is microns. C has units */
    /* 1/distance, F2 is unitless. */
    /* wavelength equals c (speed of light) over frequency.*/
    /* frequency is in THz, so c is speed of light in */
    /* microns/picosecond = 3E2. */
    lambda = 3.0E2 / freq;

    /* If delta r is 0.0, the F term becomes unity. */
    if (fbc_desc->delta_r == 0.0)
    {
        F2 = 1.0;
    }
    else
    {
        /* See section 4.3 in thesis document. */
        F2 = 1.0 /
             (1.0 +
              (234.0*power ((fbc_desc->r / lambda), 3.0)) *
(pow (fbc_desc->delta_r / fbc_desc->r, 2.0))
);}

C = 21.0 * pow (lambda, 2.5) / pow (fbc_desc->r, 3.5);
alpha = sqrt (F2 * pow (sin (C*fbc_desc->z/sqrt (F2)), 2.0));
g->r = sqrt (1.0 - alpha) * fbc_desc->a;
g->theta = 0.0;
}

Aon_FBC_GAIN2 (CmathT_Complex* g, double freq, void* void_fbc_desc)
{
AonT_FBC_Desc* fbc_desc;
double C, F2;
double lambda;
double alpha;

type = (AonT_FBC_Desc*) void_fbc_desc;

/* In this procedure the unit for time is picoseconds, */
/* the unit for distance is microns. C has units */
/* 1/distance, F2 is unitless. */

/* wavelength equals c (speed of light) over frequency. */
/* frequency is in THz, so c is speed of light in */
/* microns/picosecond = 3E2. */
lambda = 3.0E2 / freq;

/* If delta r is 0.0, the F term becomes unity. */
if (fbc_desc->delta_r == 0.0)
{
    F2 = 1.0;
}
else
{
    /* See section 4.3 in thesis document. */
    F2 = 1.0 +
        (234.0*pow ((fbc_desc->r / lambda), 3.0)) *
        (pow (fbc_desc->delta_r / fbc_desc->r, 2.0))
    );
}

C = 21.0 * pow (lambda, 2.5) / pow (fbc_desc->r, 3.5);
alpha = sqrt (F2 * pow (sin (C*fbc_desc->z/sqrt (F2)), 2.0));
g->r = sqrt (alpha) * fbc_desc->a;
g->theta = CMATH_PI / 2.0;
B.4 Star Coupler Support Code

The star coupler process model uses functions that determine the output of a star coupler due to an incident pulse. These functions and their accompanying structures are found in `aon_stc.ex.h` and `aon_stc.ex.c`.

**aon_stc.ex.h**

```c
/* Greg Campbell */
/* AON Model Suite */
/* aon_stc.ex.h */

/** TypeDefs ****/
typedef struct
{
    int N;
    double delay;
    double insertion_loss;
} AonT_STC_Desc;

/** Function Prototypes ****/
AonT_STC_Desc*
    Aon_STC_Create (int N, double loss, double delay);
void    Aon_STC_Propagate (AonT_Pulse* pulse, AonT_STC_Desc* stc_desc);
void    Aon_STC_Noise_Propagate (AonT_Noise* noise, AonT_STC_Desc* stc_desc);
```

**aon_stc.ex.c**

```c
/* Greg Campbell */
/* AON Model Suite */
/* aon_stc.ex.h */

#include "lidsfs/usr/local3/opnet-2.5-sol/sys/include/opnet.h"
#define AON_BASE_DECS
#include <math.h>
#include cmath.h
#include aon_base.ex.h"
#include "aon_stc.ex.h"

AonT_STC_Desc*
    Aon_STC_Create (int N, double loss, double delay)
{
    AonT_STC_Desc* stc_desc;
    stc_desc = (AonT_STC_Desc*) malloc (sizeof (AonT_STC_Desc));
    stc_desc->N = N;
    stc_desc->insertion_loss = cmath_dB (loss);
    stc_desc->delay = delay;
```
void Aon_STC_Propagate (AonT_Pulse* pulse, AonT_STC_Desc* stc_desc)
{
    int i;
    CmathT_Complex split_self;

    split_self.r = sqrt ((1.0/(double)stc_desc->N) * stc_desc->insertion_loss);
    split_self.theta = 0.0;

    cmath_vector_mult_vector (pulse->shape, AonI_Len, pulse->shape, &split_self);
}

void AonSTC_Noise_Propagate (AonT_Noise* noise, AonT_STC_Desc* stc_desc)
{
    noise->power = noise->power * (1.0 / (double)stc_desc->N) * stc_desc->insertion_loss;
}
B.5 Optical Amplifier Support Code

The optical amplifier process model uses functions that determine the output of an optical amplifier. These functions and their accompanying structures are found in aon_amp.ex.h and aon_amp.ex.c.

*aon_amp.ex.h*

/* Greg Campbell */
/* AON Model Suite */
/* aon_xmt.ex.h */
/* Transmitters */

/**** Constants ****/
#define planck 6.626E-34
#define AONC_AMP_UPDATE
#define AONC_AMP_POWER
/**** Typedefs ****/
typedef struct
{
    List power_list;
    int low_pulse_num;
    int high_pulse_num;
} AonT_Amp_Power_Interrupt_Desc;
typedef struct
{
    double gain;
    double sat;
    double tau;
    double noise;
    double delay;
    double d_noise;
    double pulse_power;
    double rcv_noise;
    AonT_Amp_Power_Interrupt_Desc*power_list;
    AonT_Port_Noise_In* noise_in;
    AonT_Port_Noise_Out* noise_out;
} AonT_Amp_Desc;

/**** Function Prototypes ****/
AonT_Amp_Desc* Aon_Amp_Desc_Create ();
void Aon_Amp_Noise_Update (AonT_Amp_Desc *amp, double time);
void Aon_Amp_Pulse (AonT_Amp_Desc* amp, AonT_Pulse* pulse);
void Aon_Amp_Pulse_Power_Interrupt_Set (AonT_Amp_Desc* amp, AonT_Pulse* pulse);
void Aon_Amp_Pulse_Power_Interrupt_Get (AonT_Amp_Desc* amp);
double Aon_Amp_Next_Update (AonT_Amp_Desc* amp);
/* Greg Campbell */
/* AON Model Suite */
/* aon_amp.ex.c */

#include "lidsfs/usr/local3/opnet-2.5-sol/sys/include/opnet.h"
#include "math.h"
#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_amp.ex.h"

AonT_Amp_Desc*
Aon_Amp_Desc_Create()
{
  AonT_Amp_Desc* amp;
  amp = (AonT_Amp_Desc*) malloc(sizeof(AonT_Amp_Desc));

  amp->power_list = (AonT_Amp_PowerInterrupt_Desc*) malloc
  (sizeof(AonT_Amp_PowerInterrupt_Desc));
  opprg_list_init(&((amp)->power_list->power_list));
  amp->power_list->low_pulse_num = 0;
  amp->power_list->high_pulse_num = 0;

  amp->noise_in = Aon_Port_Noise_In_Create();
  amp->noise_out = Aon_Port_Noise_Out_Create();

  amp->rcv_noise = 0.0;
  amp->pulse_power = 0.0;

  return (amp);
}

void
Aon_Amp_Noise_Update (AonT_Amp_Desc *amp, double time)
{
  AonT_Noise* noise_bin;
  Packet* pkptr;
  double noise;
  double noise_tot;
  double W_in;
  double gain;
  double delta_f;
  double freq;
  int i;

  W_in = amp->rcv_noise + amp->pulse_power;

  gain = amp->gain / (1.0 + W_in / amp->sat);

  delta_f = ((AonI_High_Freq - AonI_Low_Freq) / AonI_N_Segment) * 1E12;
  for (i = 0; i < AonI_N_Segment; i++)
  {
    freq = (AonI_Low_Freq + ((double) i / (double) AonI_N_Segment) * 
    (AonI_High_Freq - AonI_Low_Freq)) * 1E12;
    noise = gain * planck * amp->noise * freq * delta_f;
    noise_tot = noise + gain * (*(amp->noise_in->noise_array + i));

    Aon_Port_Noise_Out_Handle_Abs (amp->noise_out, i, noise_tot, 0,
    amp->delay);

    /* if (i == 0) */
printf ("time = %lf ppower = %lf noise packet bin = %d noise = %lf, power =
%lf\n", time, amp->pulse_power, i, noise, noise_tot); */
}

if (time < 1001.0)
{
    printf ("amp sat = %lf\n", amp->sat);
    printf ("amp gain = %lf\n", amp->gain);
    printf ("gain = %lf W_in = %lf, delta_f = %lf\n", gain, W_in, delta_f);
}
}

void
Aon_Amp_Pulse (AonT_Amp_Desc* amp, AonT_Pulse* pulse)
{
    double W_in;
    double gain;
    int i;
    double pulse_power;

    pulse_power = amp->pulse_power;

    for (i = 0; i < AonI_Len; i++)
    {
        pulse_power = pulse_power * exp ((-1.0) * (AonI_Duration / (double) AonI_Len) / 
amp->tau);
        pulse_power += pow (((pulse->shape) + i)->r, 2.0) * 
(AonI_Duration / (double) AonI_Len) / amp->tau;
        W_in = amp->rcv_noise + amp->pulse_power;
        gain = amp->gain / (1.0 + W_in / amp->sat);
        ((pulse->shape) + i)->r = ((pulse->shape) + i)->r * sqrt (gain);
    }
}

void
Aon_Amp_Pulse_Power_Interrupt_Set (AonT_Amp_Desc* amp, AonT_Pulse* pulse)
{
    double pulse_power_tot;
    double delta_power;
    double W_in;
    double gain_old;
    int pulse_significant;
    List* pulse_power_list;
    double* list_entry;
    int i;
    double gain;

    pulse_significant = 0;

    W_in = amp->rcv_noise + amp->pulse_power;
    gain_old = amp->gain / (1.0 + W_in / amp->sat);
    pulse_power_tot = 0.0;

    for (i = 0; i < AonI_Len; i++)
    {
        W_in = W_in * exp ((-1.0) * (AonI_Duration / (double) AonI_Len) / amp->tau);
        delta_power = pow (((pulse->shape) + i)->r, 2.0) * 
(AonI_Duration / (double) AonI_Len) / amp->tau;
        W_in += delta_power;
        pulse_power_tot += delta_power;

        gain = amp->gain / (1.0 + W_in / amp->sat);
        if ((gain >= (gain_old * (1.0 + amp->d_noise))) ||
(gain <= (gain_old * (1.0 - amp->d_noise)))
{
    if (pulse_significant == 0)
    {
        pulse_power_list = op_prg_list_create();
        pulse_significant = 1;
    }

    list_entry = (double*) malloc (sizeof (double));
    *(list_entry) = pulse_power_tot;
    op_prg_list_insert (pulse_power_list, (void*) list_entry, OPC_LISTPOS_TAIL);
    op_intrpt_schedule_self (op_sim_time () +
        ((double)i/(double)AonI_Len)*AonI_Duration,
        amp->power_list->high_pulse_num + AONC_AMP_POWER);
    printf ("interrupt time = %lf, code = %d
", op_sim_time () +
        ((double)i/(double)AonI_Len)*AonI_Duration,
        amp->power_list->high_pulse_num + AONC_AMP_POWER);
    gain_old = gain;
    pulse_power_tot = 0.0;
}
}

if (pulse_significant)
{
    op_prg_list_insert (&(amp->power_list->power_list), pulse_power_list, OPC_LISTPOS_TAIL);
    amp->power_list->high_pulse_num++;
}

void
Aon_Amp_Pulse_Power_Interrupt_Get (AonT_Amp_Desc* amp)
{
    int pulse_num;
    int done;
    List* pulse_power_list;
    double* list_entry;
    done = 0;
    pulse_num = op_intrpt_code () - AONC_AMP_POWER;
    pulse_power_list = (List*) op_prg_list_access
    (&(amp->power_list->power_list), pulse_num - amp->power_list->low_pulse_num);
    list_entry = (double*) op_prg_list_remove (pulse_power_list, OPC_LISTPOS_HEAD);
    amp->pulse_power += (*(list_entry));
    while (done == 0)
    {
        if (op_prg_list_size (&(amp->power_list->power_list)))
        {
            pulse_power_list = (List*) op_prg_list_access
            (&(amp->power_list->power_list), OPC_LISTPOS_HEAD);
            if (op_prg_list_size (pulse_power_list) == 0)
            {
                pulse_power_list = (List*) op_prg_list_remove
                (&(amp->power_list->power_list), OPC_LISTPOS_HEAD);
                op_prg_list_free (pulse_power_list);
            }
        }
    }
}
amp->power_list->low_pulse_num++;
}
else
    done = 1;
}
else
    done = 1;
}
}

double
Aon_Amp_Next_Update (AonT_Amp_Desc* amp)
{
    double delta_t;
    double D;
    D = 1.0 + amp->d_noise;
    delta_t = (-1.0)*amp->tau*log
        ((((amp->sat*(1.0-D)+amp->rcv_noise+amp->pulse_power) / D) - amp->rcv_noise) / amp->pulse_power);

    return (delta_t);
}
B.6 ASE Filter Support Code

The ASE filter process model uses functions to determine the output of the ASE filter due to incident pulse and noise streams. These functions and their accompanying structures are found in aon_ase.ex.h and aon_ase.ex.c.

aon_ase.ex.h

/* Greg Campbell */
/* AON Model Suite */

/*** Typedefs ****/
typedef struct
{
    double FSR;
    double W;
    double a;
} AonT_ASE_Desc;

AonT_ASE_Desc* Aon_ASE_Create (double FSR, double W, double a);
void Aon_ASE_Pulse (AonT_Pulse* pulse, AonT_ASE_Desc* ase_desc);
void Aon_ASE_Noise (AonT_Noise* noise, AonT_ASE_Desc* ase_desc);
int Aon_ASE_Gain (CmathT_Complex* g, double freq,
    void* void_ase_desc);

aon_ase.ex.c

/* Greg Campbell */
/* AON Model Suite */

#include <cmath.h>
#include "math.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"
#include "aon_ase.ex.h"

AonT_ASE_Desc*
Aon_ASE_Create (double FSR, double W, double a)
{
    AonT_ASE_Desc* ase_desc;
    double b;

    ase_desc = (AonT_ASE_Desc*) malloc (sizeof (AonT_ASE_Desc));

    ase_desc->FSR = FSR;
    ase_desc->W = W;
    ase_desc->a = cmath_dB (a);

    return (ase_desc);
void Aon_ASE_Pulse (AonT_Pulse* pulse, AonT_ASE_Desc* ase_desc)
{
    Aon_Lin_Pulse (pulse, Aon_ASE_Gain, (void*) ase_desc);
}

void Aon_ASE_Noise (AonT_Noise* noise, AonT_ASE_Desc* ase_desc)
{
    Aon_Lin_Noise (&(noise->power), noise->freq_bin,
        Aon_ASE_Gain, (void*) ase_desc);
}

Aon_ASE_Gain (CmathT_Complex* g, double freq, void* void_ase_desc)
{
    AonT_ASE_Desc*    ase_desc;
    CmathT_Complex    tmp, tmp2;
    int              num_freq;

    ase_desc = (AonT_ASE_Desc*) void_ase_desc;

    /* move freq to principal value of freq [-FSR/2, FSR).*/
    num_freq = floor (freq / ase_desc->FSR);
    freq = freq - num_freq * ase_desc->FSR;
    if (freq > (ase_desc->FSR / 2.0))
    {
        freq = freq - ase_desc->FSR;
    }

    if (fabs (freq) < (ase_desc->W / 2.0))
    {
        g->r = ase.desc->a;
    }
    else
    {
        g->r = 0.0;
    }

    g->theta = 0.0;
}
B.7 Fabry-Perot Filter Support Code

The Fabry-Perot filter process model uses functions to determine the output of the Fabry-Perot filter due to incident pulse and noise streams. These functions and their accompanying structures are found in aon_fab.ex.h and aon_fab.ex.c.

*aon_fab.ex.h*

/* Greg Campbell */
/* AON Model Suite */

/**** Typedefs ****/

typedef struct
{
    double tau;
    double R;
    double A;
} AonT_Fab_Desc;

AonT_Fab_Desc* Aon_Fab_Create (double FSR, double finesse, double Tmax);
void Aon_Fab_Pulse (AonTPulse* pulse, AonT_Fab_Desc* fab_desc);
void Aon_Fab_Noise (AonT_Noise* noise, AonT_Fab_Desc* fab_desc);
int Aon_Fab_Gain (CmathT_Complex* g, double freq,
                             void* void_fab_desc);

*aon_fab.ex.c*

/* Greg Campbell */
/* AON Model Suite */

#include <math.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"
#include "aon_fab.ex.h"

AonT_Fab_Desc* Aon_Fab_Create (double FSR, double finesse, double Tmax)
{
    AonT_Fab_Desc* fab_desc;
    double b;
    fab_desc = (AonT_Fab_Desc*) malloc (sizeof (AonT_Fab_Desc));
    fab_desc->tau = 1.0 / (2.0 * FSR);
    b = (-1.0) * (pow ((CMATH_PI / finesse), 2.0) + 2.0);
    fab_desc->R = ((-1.0)*b - sqrt (pow (b, 2.0) - 4.0)) / 2.0;
}
fab_desc->A = (1.0 - fab_desc->R) * (1.0 - sqrt(Tmax));

return (fab_desc);
}

void
Aon_Fab_Pulse (AonT_Pulse* pulse, AonT_Fab_Desc* fab_desc)
{
    Aon_Lin_Pulse (pulse, Aon_Fab_Gain, (void*) fab_desc);
}

void
Aon_Fab_Noise (AonT_Noise* noise, AonT_Fab_Desc* fab_desc)
{
    Aon_Lin_Noise (&(noise->power), noise->freq_bin,
                   Aon_Fab_Gain, (void*) fab_desc);
}

Aon_Fab_Gain (CmathT_Complex* g, double freq, void* void_fab_desc)
{
    AonT_Fab_Desc* fab_desc;
    CmathT_Complex tmp, tmp2;
    fab_desc = (AonT_Fab_Desc*) void_fab_desc;
    tmp.r = fab_desc->R;
    tmp.theta = (-4.0) * CMATH_PI * freq * fab_desc->tau;
    tmp2.r = 1.0;
    tmp2.theta = 0.0;
    cmath_sub (&tmp, &tmp2, &tmp);
    tmp.r = 1.0 / tmp.r;
    tmp.theta = (-1.0) * tmp.theta;
    tmp2.r = 1.0 - fab_desc->A - fab_desc->R;
    tmp2.theta = 0.0;
    cmath_mult (g, &tmp, &tmp2);
    g->theta = g->theta - 2.0 * CMATH_PI * fab_desc->tau;
}
B.8 Mach-Zehnder Filter Support Code

The Mach-Zehnder filter *process model* uses functions to determine the output of the Mach-Zehnder filter due to incident pulse and noise streams. These functions and their accompanying structures are found in *aon_mzf.ex.h* and *aon_mzf.ex.c*.

*aon_mzf.ex.h*

```c
/* Greg Campbell */
/* AON Model Suite */

**** Typedefs ****

typedef struct
{
    double tau;
} AonT_MZF_Desc;

AonT_MZF_Desc* Aon_MZFCreate (double FSR);
void Aon_MZF_Pulse (AonT_Pulse* pulse, AonT_MZF_Desc* mzf_desc);
void Aon_MZF_Noise (AonT_Noise* noise, AonT_MZF_Desc* mzf_desc);
int Aon_MZF_Gain (CmathT_Complex* g, double freq, void* void_mzf_desc);
void Aon_MZF_Pulse2 (AonT_Pulse* pulse, AonT_MZF_Desc* mzf_desc);
void Aon_MZF_Noise2 (AonT_Noise* noise, AonT_MZF_Desc* mzf_desc);
int Aon_MZF_Gain2 (CmathT_Complex* g, double freq, void* void_mzf_desc);
```

*aon_mzf.ex.c*

```c
/* Greg Campbell */
/* AON Model Suite */

#include <math.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"
#include "aon_mzf.ex.h"

AonT_MZF_Desc* Aon_MZFCreate (double FSR)
{
    AonT_MZF_Desc* mzf_desc;
    mzf_desc = (AonT_MZF_Desc*) malloc (sizeof (AonT_MZF_Desc));
    mzf_desc->tau = 1.0 / FSR;
    return (mzf_desc);
}
```
Aon_MZF_Pulse1 (AonT_Pulse* pulse, AonT_MZF_Desc* mzf_desc)
    {
        Aon_Lin_Pulse (pulse, Aon_MZF_Gain1, (void*) mzf_desc);
    }

void Aon_MZF_Noisel (AonT_Noise* noise, AonT_MZF_Desc* mzf_desc)
    {
        Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_MZF_Gain1,
                        (void*) mzf_desc);
    }

void Aon_MZF_Pulse2 (AonT_Pulse* pulse, AonT_MZF_Desc* mzf_desc)
    {
        Aon_Lin_Pulse (pulse, Aon_MZF_Gain2, (void*) mzf_desc);
    }

void Aon_MZF_Noise2 (AonT_Noise* noise, AonT_MZF_Desc* mzf_desc)
    {
        Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_MZF_Gain2,
                        (void*) mzf_desc);
    }

Aon_MZF_Gain1 (CmathT_Complex* g, double freq, void* void_mzf_desc)
    {
        AonT_MZF_Desc* mzf_desc;
        CmathT_Complex tmp, tmp2;

        mzf_desc = (AonT_MZF_Desc*) void_mzf_desc;

        tmp.r = 0.5;
        tmp.theta = (-2.0) * CMATH_PI * freq * mzf_desc->tau;
        tmp2.r = 0.5;
        tmp2.theta = 0.0;
        cmath_sub (g, &tmp, &tmp2);
    }

Aon_MZF_Gain2 (CmathT_Complex* g, double freq, void* void_mzf_desc)
    {
        AonT_MZF_Desc* mzf_desc;
        CmathT_Complex tmp, tmp2;

        mzf_desc = (AonT_MZF_Desc*) void_mzf_desc;

        tmp.r = 0.5;
        tmp.theta = ((-2.0) * CMATH_PI * freq * mzf_desc->tau) - (CMATH_PI / 2.0);
        tmp2.r = 0.5;
        tmp2.theta = (-1.0) * CMATH_PI / 2.0;
        cmath_add (g, &tmp, &tmp2);
    }
B.9 Wavelength Division (De)Multiplexer Support Code

The wavelength division (de) multiplexer process model uses functions that determine the output of a WDM multiplexer due to an incident pulse. These functions and their accompanying structures are found in aon_wdm.ex.h and aon_wdm.ex.c.

*aon_wdm.ex.h*

/* Greg Campbell */
/* AON Model Suite */

/**** Typedefs ****/

typedef struct
{
    double FSR;
    double a;
} AonT_WDM_Desc;

AonT_WDM_Desc* AonWDM_Create (double FSR, double a)
{
    AonT_WDM_Desc* wdm_desc;
    wdm_desc = (AonT_WDM_Desc*) malloc (sizeof (AonT_WDM_Desc));
    wdm_desc->FSR = FSR;
    wdm_desc->a = cmath_dB (a);
    return (wdm_desc);
}

*aon_wdm.ex.c*

/* Greg Campbell */
/* AON Model Suite */

#include <math.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"
#include "aon_wdm.ex.h"

AonT_WDM_Desc* AonWDM_Create (double FSR, double a)
{
    AonT_WDM_Desc* wdm_desc;
    wdm_desc = (AonT_WDM_Desc*) malloc (sizeof (AonT_WDM_Desc));
    wdm_desc->FSR = FSR;
    wdm_desc->a = cmath_dB (a);
    return (wdm_desc);
}
void Aon_WDM_Pulse1 (AonT_Pulse* pulse, AonT_WDM_Desc* wdm_desc)
{
    Aon_Lin_Pulse (pulse, Aon_WDM_Gain1, (void*) wdm_desc);
}

void Aon_WDM_Noisel (AonT_Noise* noise, AonT_WDM_Desc* wdm_desc)
{
    Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_WDM_Gain1,
                    (void*) wdm_desc);
}

void Aon_WDM_Pulse2 (AonT_Pulse* pulse, AonT_WDM_Desc* wdm_desc)
{
    Aon_Lin_Pulse (pulse, Aon_WDM_Gain2, (void*) wdm_desc);
}

void Aon_WDM_Noise2 (AonT_Noise* noise, AonT_WDM_Desc* wdm_desc)
{
    Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_WDM_Gain2,
                    (void*) wdm_desc);
}

Aon_WDM_Gain1 (CmathT_Complex* g, double freq, void* void_wdm_desc)
{
    AonT_WDM_Desc* wdm_desc;
    CmathT_Complex tmp, tmp2;

    wdm_desc = (AonT_WDM_Desc*) void_wdm_desc;

    g->r = wdm_desc->a * sin (2.0*CMATH_PI*freq / wdm_desc->FSR);
    g->theta = 0.0;
}

Aon_WDM_Gain2 (CmathT_Complex* g, double freq, void* void_wdm_desc)
{
    AonT_WDM_Desc* wdm_desc;
    CmathT_Complex tmp, tmp2;

    wdm_desc = (AonT_WDM_Desc*) void_wdm_desc;

    g->r = wdm_desc->a * cos (2.0*CMATH_PI*freq / wdm_desc->FSR);
    g->theta = 0.0;
}
B.10 Wavelength Router Support Code

The wavelength router process model uses functions that determine the output of a wavelength router due to an incident pulse. These functions and their accompanying structures are found in aon_rou.ex.h and aon_rou.ex.c.

**aon_rou.ex.h**

```c
/* Greg Campbell */
/* AON Model Suite */

/** Typedefs ****/

typedef struct {
    int N;
    double FSR;
    double a;
    double k;
    double delay;
    int i;
} AonT_Rou_Desc;

AonT_Rou_Desc* Aon_Rou_Create (int N, double FSR, double a, double k, double delay);
void AonRou_Pulse (AonT_Pulse* pulse, AonT_Rou_Desc* rou_desc, int i);
void Aon_Rou_Noise (AonT_Noise* noise, AonT_Rou_Desc* rou_desc, int i);
int Aon_Rou_Gain (CmathT_Complex* g, double freq, void* void_rou_desc);
```

**aon_rou.ex.c**

```c
/* Greg Campbell */
/* AON Model Suite */

#include <math.h>
#include "cmath.h"
#include "/lidafs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"
#include "aon_rou.ex.h"

AonT_Rou_Desc* Aon_Rou_Create (int N, double FSR, double a, double k, double delay) {
    AonT_Rou_Desc* rou_desc;
    rou_desc = (AonT_Rou_Desc*) malloc (sizeof (AonT_Rou_Desc));
    rou_desc->N = N;
    rou_desc->FSR = FSR;
    rou_desc->a = cmath_dB (a);
    rou_desc->k = cmath_dB ((-1.0)*k);
}
```

182
rou_desc->delay = delay;
return (rou_desc);
}

void
Aon_Rou_Pulse (AonT_Pulse* pulse, AonT_Rou_Desc* rou_desc, int i)
{
    rou_desc->i = i;
    Aon_Lin_Pulse (pulse, Aon_Rou_Gain, (void*) rou_desc);
}

void
Aon_Rou_Noise (AonT_Noise* noise, AonT_Rou_Desc* rou_desc, int i)
{
    rou_desc->i = i;
    Aon_Lin_Noise (&(noise->power), noise->freq_bin, Aon_Rou_Gain,
    (void*) rou_desc);
}

int
Aon_Rou_Gain (CmathT_Complex* g, double freq, void* void_rou_desc)
{
    AonT_Rou_Desc* rou_desc;
    double tmp, tmp2;
    double FSR_N;
    int chan;
    rou_desc = (AonT_Rou_Desc*) void_rou_desc;
    chan = rou_desc->i;
    FSR_N = rou_desc->FSR / rou_desc->N;
    tmp = (freq - chan*FSR_N)*CMATH_PI;
    if (sin (tmp/rou_desc->FSR) != 0.0)
    {
        tmp2 = pow (((sin (tmp/FSR_N) / sin (tmp/rou_desc->FSR)) / 
    (double)(rou_desc->N)), 2.0);
    }
    else
    {
        tmp2 = 1.0;
    }
    g->r = sqrt (rou_desc->a / (((1.0 - rou_desc->k) * (tmp2)+ rou_desc->k));
    g->theta = 0.0;
}
B.11 Receiver and Probe Support Code

The probe and receiver process models both use functions that output statistics about the incident pulse stream. These functions and their accompanying structures are found in aon_rcv.ex.h and aon_rcv.ex.c.

**aon_rcv.ex.h**

```c
/* Greg Campbell */
/* AON Model Suite */
/* aon_rcv.ex.h */

/** Typedefs ****/
typedef struct
{
    double eye_origin;
    double eye_width;
    int coherent;
    int signal_id;
} AonT_Rcv_Desc;

/** Function Prototypes ****/
void Aon_Rcv_Update (AonT_Port_Pulse* port, double noise,
                     double old_time, double time, AonT_Rcv_Desc* rcv_desc);
void Aon_Rcv_Pulse (AonT_Pulse* pulse, int pulse_num);
```

**aon_rcv.ex.c**

```c
/* Greg Campbell */
/* AON Model Suite */
/* aon_rcv.ex.h */

#include "/lidfs/usr/local3/opnet-2.5-sol/sys/include/opnet.h"
#define AON_BASE_DECS
#include <math.h>
#include "cmath.h"
#include "aon_base.ex.h"
#include "aon_rcv.ex.h"

#define AONC_RCV_INSTANT 0
#define AONC_RCV_NOISE 1
#define AONC_RCV_EYE 2
#define AONC_RCV_AMP 3
#define AONC_RCV_PHASE 4
#define AONC_RCV_FFT_AMP 5
#define AONC_RCV_FFT_PHASE 6
#define AONC_RCV_PULSE_STAT_NUM 4

void Aon_Rcv_Update (AonT_Port_Pulse* port, double noise, double old_time,
```
double time, AonT_Rcv_Desc* rcv_desc
{
int i, j, k;
static CmathT_Complex *amp_sum;
static CmathT_Complex *noise_amp_sum;
static double *power_sum;
static double *noise_power_sum;
static int sum_init = 0;
double magnitude;
double noise_magnitude;
double cur_time;
AonT_Pulse *pulse;
AonT_Port_Entry *port_entry;
double eyes;
double eye_time;
double last_mag;
double last_noise_mag;

if (sum_init == 0)
{
amp_sum = (CmathT_Complex*)
malloc (AonI_Len * sizeof (CmathT_Complex));

power_sum = (double*)
malloc (AonI_Len * sizeof (double));

noise_amp_sum = (CmathT_Complex*)
malloc (AonI_Len * sizeof (CmathT_Complex));

noise_power_sum = (double*)
malloc (AonI_Len * sizeof (double));

sum_init = 1;
}

if (op_prg_list_size (&(port->input)) > 0)
{
cur_time = old_time;
}
else
{
cur_time = time;
op_stat_local_write_t (AONC_RCV_INSTANT, noise, cur_time);
}
op_stat_local_write_t (AONC_RCV_NOISE, noise, cur_time);

while (cur_time < time)
{
for (i = 0; i < AonI_Len; i++)
{
if (rcv_desc->coherent)
{
(amp_sum + i)->r = 0.0;
(amp_sum + i)->theta = 0.0;
(noise_amp_sum + i)->r = 0.0;
(noise_amp_sum + i)->theta = 0.0;
}
else
{
(*(power_sum + i)) = 0.0;
(*(noise_power_sum + i)) = 0.0;
}
}
for (i = 0; i < (op_prg_list_size(&(port->input))); i++)
{
    port_entry = (AonT_Port_Entry*)
        op_prg_list_access(&(port->input), i);
    pulse = port_entry->pulse;
    j = 0;
    k = (((cur_time - port_entry->entry_time) / AonI_Duration) * 
        AonI_Len);
    while (k < AonI_Len)
    {
        if (rcv_desc->coherent)
        {
            cmath_add((amp_sum + j), (amp_sum + j), 
                (pulse->shape + k));

            if (pulse->source != rcv_desc->signal_id)
            {
                cmath_add((noise_amp_sum + j), (noise_amp_sum + j), 
                    (pulse->shape + k));
            }
        }
        else
        {
            (*(power_sum + j)) += pow((pulse->shape + k)->r, 2.0);

            if (pulse->source != rcv_desc->signal_id)
            {
                (*(noise_power_sum + j)) += 
                    pow((pulse->shape + k)->r, 2.0);
            }
        }
        j++; k++;
    }

    if ((time - port_entry->entry_time) > AonI_Duration)
    {
        port_entry = (AonT_Port_Entry*)
            op_prg_list_remove(&(port->input), i);

        Aon_Port_Entry_Destroy(port_entry);
        i--; 
    }
}

j = 0;
lst_mag = -1.0;
lst_noise_mag = -1.0;
while (((cur_time < time) && (j < AonI_Len))
{
    if (rcv_desc->coherent)
    {
        magnitude = pow((amp_sum + j)->r, 2.0) + noise;
        noise_magnitude = pow((noise_amp_sum + j)->r, 2.0) + noise;
    }
    else
    {
        magnitude = (*(power_sum + j)) + noise;
        noise_magnitude = (*(noise_power_sum + j)) + noise;
    }

    if (fabs(magnitude - lst_mag) > AonI_Min_Power)
    {
        op_stat_local_write_t (AONC_RCV_INSTANT, magnitude, cur_time);

        eyes = floor((cur_time - rcv_desc->eye_origin) / 
            rcv_desc->eye_width);
eye_time = cur_time - rcv_desc->eye_origin -
   eyes * rcv_desc->eye_width;
op_stat_local_write_t (AONC_RCV_EYE, magnitude, eye_time);

last_mag = magnitude;
}

if (fabs (noise_magnitude - last_noise_mag) > AonI_MinPower)
{
op_stat_local_write_t (AONC_RCV_NOISE, noise_magnitude, cur_time);
last_noise_mag = noise_magnitude;
}

j++;
cur_time += (AonI_Duration / (double)AonI_Len);
}

/* If there are no more pulses left, skip to current time. */
if (op_prg_list_size (&(port->input)) == 0)
{
curtime = time;
op_stat_local_write_t (AONC_RCV_INSTANT, noise, cur_time);
}
}

void
Aon_Rcv_Pulse (AonT_Pulse* pulse, int pulse_num)
{
static int
fft_init = 0;
static CmathT_Complex* fft_shape;
double cur_time;
double freq;
int i;

if (fft_init == 0)
{
fft_shape = (CmathT_Complex*) malloc (AonI_Len * sizeof (CmathT_Complex));
fft_init = 1;
}

cmath_FFT (fft_shape, pulse->shape, AonI_Nu);

for (i = 0; i < AonI_Len; i++)
{
cur_time = (double)i * (AonI_Duration / (double)AonI_Len);
freq = pulse->freq + (((i * AonI_Len/2) % AonI_Len) - AonI_Len/2) * 2.0 * CMATH_PI
   / AonI_Duration);
cmath_principle_val (pulse->shape + i);
op_stat_local_write_t (pulse_num * AONC_RCV_PULSE_STAT_NUM + AONC_RCV_AMP,
   (pulse->shape + i)->r, cur_time);
op_stat_local_write_t (pulse_num * AONC_RCV_PULSE_STAT_NUM + AONC_RCV_PHASE,
   (pulse->shape + i)->theta, cur_time);
cmath_principle_val (fft_shape + i);
op_stat_local_write_t (pulse_num * AONC_RCV_FFT_STAT_NUM + AONC_RCV_FFT_AMP,
   (fft_shape + i)->r, freq);
op_stat_local_write_t (pulse_num * AONC_RCV_FFT_STAT_NUM + AONC_RCV_FFT_PHASE,
   (fft_shape + i)->theta, freq);
}

printf ("###here baby\n");
}
B.12 Complex Mathematics Support Code

Many of the functions in the AON Model Suite use functions that perform operations on complex numbers or arrays of complex numbers. These functions can be found in cmath.ex.h and cmath.ex.c.

cmath.ex.h

#define CMATH_PI 3.14159265
typedef struct
{
    double r;
    double theta;
} CmathT_Complex;

void cmath_assign (CmathT_Complex *a, double r, double theta);
void cmath_print (CmathT_Complex *a);
void cmath_add (CmathT_Complex *aplusb, CmathT_Complex *a, CmathT_Complex *b);
void cmath_sub (CmathT_Complex *aplusb, CmathT_Complex *a, CmathT_Complex *b);
void cmath_mult (CmathT_Complex *amultb, CmathT_Complex *a, CmathT_Complex *b);
void cmath_mult_scalar (CmathT_Complex *amultb, CmathT_Complex *a, double b);
void cmath_vector_print (CmathT_Complex *a, int len);
void cmath_vector_mult_vector (CmathT_Complex *amultb, int len,
                              CmathT_Complex *a, CmathT_Complex *b);
void cmath_vector_mult_scalar (CmathT_Complex *amultb, int len,
                              CmathT_Complex *a, double b);
void cmath_swap (CmathT_Complex *a, CmathT_Complex *b);
void cmath_copy (CmathT_Complex *a, CmathT_Complex *b);
void cmath_vector_copy (CmathT_Complex *a, CmathT_Complex *b, int len);
void cmath_W (CmathT_Complex *w, int k, int N);
void cmath_FFT (CmathT_Complex *fft, CmathT_Complex *a, int nu);
void cmath_inv_FFT (CmathT_Complex *fft, CmathT_Complex *a, int nu);
void cmath_principle_val (CmathT_Complex *a);
double cmath_DB (double a);

cmath.ex.c

#include "math.h"
#include "cmath.h"

void cmath_assign (CmathT_Complex *a, double r, double theta)
{
    a->r = r;
    a->theta = theta;
}

void cmath_principle_val (CmathT_Complex *a)
{
    int N;
    if (a->r < 0.0)
\{ 
    a->r = fabs (a->r); 
    a->theta += CMATH_PI; 
\}

N = ceil ((a->theta - CMATH_PI) / (2.0 * CMATH_PI));
a->theta -= (N*2.0*CMATH_PI);
}

tvoid cmath_print (CmathT_Complex *a)
{ 
    printf (*R: %Lf\tTHETA: %lf\n*, a->r, a->theta); 
}

tvoid cmath_add (CmathT_Complex *aplusb, CmathT_Complex *a, CmathT_Complex *b)
{ 
    double x, y;
    
    x = a->r * cos (a->theta) + b->r * cos (b->theta);
    y = a->r * sin (a->theta) + b->r * sin (b->theta);
    
    if (x == 0.0)
    { 
        aplusb->r = y;
        aplusb->theta = CMATH_PI / 2.0;
    }
    else
    { 
        aplusb->r = hypot (x, y);
        aplusb->theta = atan2 (y, x);
    }
}

tvoid cmath_sub (CmathT_Complex *asubb, CmathT_Complex *a, CmathT_Complex *b)
{ 
    double x, y;
    
    x = a->r * cos (a->theta) - b->r * cos (b->theta);
    y = a->r * sin (a->theta) - b->r * sin (b->theta);
    
    if (x == 0)
    { 
        asubb->r = y;
        asubb->theta = CMATH_PI / 2.0;
    }
    else
    { 
        asubb->r = hypot (x, y);
        asubb->theta = atan2 (y, x);
    }
}

tvoid cmath_mult (CmathT_Complex *amultb, CmathT_Complex *a, CmathT_Complex *b)
{ 
    amultb->r = a->r * b->r;
    amultb->theta = a->theta + b->theta;
}

tvoid cmath_mult_scalar (CmathT_Complex *amultb, CmathT_Complex *a, double b)
amultb->r = a->r * b;
amultb->theta = a->theta;
}

void
cmath_vector_print (CmathT_Complex *a, int len)
{
    int i;
    for (i = 0; i < len; i++)
    {
        cmath_print (a + i);
    }
}

void
cmath_vector_mult_vector (CmathT_Complex *amultb, int len, CmathT_Complex *a,
                          CmathT_Complex *b)
{
    int i;
    for (i = 0; i < len; i++)
    {
        cmath_mult (amultb + i, a + i, b);
    }
}

void
cmath_vector_mult_scalar (CmathT_Complex *amultb, int len, CmathT_Complex *a,
                          double b)
{
    int i;
    for (i = 0; i < len; i++)
    {
        (amultb + i)->r = (a + i)->r * b;
        (amultb + i)->theta = (a + i)->theta;
    }
}

void
cmath_swap (CmathT_Complex *a, CmathT_Complex *b)
{
    CmathTComplex tmp;
    tmp.r = a->r;
    tmp.theta = a->theta;
    a->r = b->r;
    a->theta = b->theta;
    b->r = tmp.r;
    b->theta = tmp.theta;
}

void
cmath_copy (CmathT_Complex *a, CmathT_Complex *b)
{
    a->r = b->r;
    a->theta = b->theta;
}

void
cmath_vector_copy (CmathT_Complex *a, CmathT_Complex *b, int len)
{  
  int i;
  for (i = 0; i < len; i++)
  {
    cmath_copy (a + i, b + i);
  }
}

int bitrev (int a, int nu)
{
  int bits;
  int i;
  bits = 0;
  for (i = 0; i < nu; i++)
  {
    bits = bits << 1;
    bits += (a & 1);
    a = a >> 1;
  }
  return (bits);
}

void cmath_W (CmathT_Complex *w, int k, int N)
{
  w->r = 1;
  w->theta = (-2)*(CMATH_PI)*k/N;
}

void cmath_FFT (CmathT_Complex *fft, CmathT_Complex *a, int nu)
{
  int len;
  int i, j, step, num_step;
  int rev;
  CmathT_Complex tmpl, tmp2;
  len = pow (2.0, (double)nu);
  for (i = 0; i < len; i++)
  {
    rev = bitrev (i, nu);
    cmath_copy ((fft + i), (a + rev));
  }
  step = 2;
  while (step <= len)
  {
    num_step = len / step;
    for (i = 0; i < num_step; i++)
    {
      for (j = ((step / 2) + 1); j < step; j++)
      {
        (fft + j + i*step)->theta -= 2*CMATH_PI*(j - (step / 2))/step;
      }
    }
  }
  /* ### Equivalent ###
  cmath_W (W, (j - (step / 2)), len);
  cmath_mult (tmp, W, (fft + j));
  cmath_copy ((fft + j), &tmp);
  */
  for (j = 0; j < (step / 2); j++)
void cmath_inv_FFT (CmathT_Complex *fft, CmathT_Complex *a, int nu)
{
    int len;
    int i, j, step, num_step;
    int rev;
    CmathT_Complex tmpl1, tmp2;

    len = pow (2.0, (double)nu);

    for (i = 0; i < len; i++)
    {
        rev = bitrev (i, nu);
        cmath_copy ((fft + i), (a + rev));
    }

    step = 2;
    while (step <= len)
    {
        num_step = len / step;
        for (i = 0; i < num_step; i++)
        {
            for (j = ((step / 2) + 1); j < step; j++)
            {
                (fft + j + i*step)->theta += 2*CMATH_PI*(j - (step / 2))/step;
            }
            for (j = 0; j < (step / 2); j++)
            {
                cmath_add (&tmpl1, (fft + j + i*step), (fft + j + i*step + (step / 2)));
                cmath_sub (&tmp2, (fft + j + i*step), (fft + j + i*step + (step / 2)));
                cmath_copy ((fft + j + i*step), &tmpl1);
                cmath_copy ((fft + j + i*step + (step / 2)), &tmp2);
            }
            step = 2 * step;
        }
        for (i = 0; i < len; i++)
        {
            (fft + i)->r = (fft + i)->r / (double) len;
        }
    }
}

double cmath_dB (double a)
{
    double dB;

    dB = pow (10.0, (-0.1)*a);

    return (dB);
}
B.13 Linear Transfer Function Support Code

Many of the functions in the AON Model Suite use functions that perform linear transfer function operations on the complex envelopes of pulses. These functions can be found in aon_lin.ex.h and aon_lin.ex.c.

*aon_lin.ex.h*

/* Greg Campbell */
/* AON Model Suite */

/**** Typedefs ****/

/**** Prototypes ****/

void Aon_Lin_Pulse (AonT_Pulse* pulse, Procedure lin_proc, void* lin_desc);
void Aon_Lin_Noise (double* noise_power, int noise_bin, Procedure lin_proc, 
                   void* lin_desc);

*aon_lin.ex.c*

/* Greg Campbell */
/* AON Model Suite */

#include <math.h>
#include "cmath.h"
#include "$/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_lin.ex.h"

void Aon_Lin_Pulse (AonT_Pulse* pulse, Procedure lin_proc, void* lin_desc)
{
    int i;
    CmathT_Complex g;
    static CmathT_Complex*fft_shape;
    static int fft_init;
    double freq;
    if (fft_init == 0)
    {
        fft_shape = (CmathT_Complex*) malloc (AonI_LEN * sizeof (CmathT_Complex));
        fft_init = 1;
    }
    cmath_FFT (fft_shape, pulse->shape, AonI_Nu);
    for (i = 0; i < AonI_LEN; i++)
    {
        freq = pulse->freq + (((i + AonI_LEN/2) % AonI_LEN) - AonI_LEN/2) * 
                          2.0 * CMATH_PI / AonI_Duration);
    }
lin_proc (&g, freq, lin_desc);
cmath_mult (fft_shape + i, fft_shape + i, &g);
}
cmath_inv_FFT (pulse->shape, fft_shape, AonI_Nu);
}

void
Aon_Lin_Noise (double* noise_power, int noise_bin, Procedure lin_proc, void* lin_desc)
{
CmathT_Complex g;
double freq = AonI_Low_Freq + (((double)noisebin + 0.5) / (double)AonI_N_Segment) * (AonI_High_Freq - AonI_Low_Freq);
lin_proc (&g, freq, lin_desc);
(*noise_power) = (*noise_power) * pow (g.r, 2.0);
B.14 Pipeline Stages

The Pipeline model stages that model the optical fiber in point-to-point links are based on external functions. These functions can be found in aon_ps.ex.h, aon_propdel.ps.c, aon_proprcv.ps.c, aon_txdel.ps.c, and aon_txrcv.ps.c.

*aon_ps.ex.h*

/* AON Model Suite */
/* Fiber pipeline stages. */

*aon_propdel.ps.c*

/* Greg Campbell */
/* AON Model Suite */
/* Fiber prop delay. */

#include <math.h>
#include "cmath.h"
#include "/ldfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_fib.ex.h"
#include "aon_ps.ex.h"

void
aon_propdel (Packet* pkptr)
{
    int tx_objid;
    int link_objid;
    int num_links;
    int i;
    AonT_Fib_Link* link;
    AonT_Port_Pulse* port;
    double *last_time;
    int type;
    AonT_Pulse* pulse;
    AonT_Noise* noise;
    /* Multiple channels not supported. */
    /* int ch_index; */

    if (AonI_Fib_List_Init == 0)
    {
        op_prs_list_init (&AonI_Fib_List);
        AonI_Fib_List_Init = 1;
    }

    tx_objid = op_td_get_int (pkptr, OPC_TDA_PT_TX_OBJID);
    link_objid = op_td_get_int (pkptr, OPC_TDA_PT_LINK_OBJID);

    /* Multiple channels not supported. */
/* ch_index = op_td_get_int (pkptr, OPC_TDA_PT_CH_INDEX); */

/* Find link... */
num_links = op_prg_list_size (&AonI_Fib_List);
for (i = 0; i < num_links; i++)
{
   link = (AonT_Fib_Link*) op_prg_list_access (&AonI_FibList, i);
   if (link->link_objid == link_objid)
      break;
}
if (i == num_links)
{
   /* Link not found. Instantiate new link. */
   link = Aon_Fib_Link_Attr_Get ((Objid) link_objid);
}
if (link->xmtl_objid == tx_objid)
{
   port = link->port1;
   last_time = &(link->last_timel);
}
else if (link->xmt2_objid == tx_objid)
{
   port = link->port2;
   last_time = &(link->last_time2);
}
else
{
   /* Port uninitialized. Instantiate port. */
   if (link->xmtl_objid == -1)
   {
      link->port1 = Aon_Port_Pulse_Create ();
      link->xmtl_objid = tx_objid;
      port = link->port1;
      link->last_time2 = op_sim_time ();
      last_time = &(link->last_time2);
   }
   else
   {
      link->port2 = Aon_Port_Pulse_Create ();
      link->xmt2_objid = tx_objid;
      port = link->port2;
      link->last_time2 = op_sim_time ();
      last_time = &(link->last_time2);
   }
}

pulse = Aon_Pulse_Packet_Get (pkptr);
Aon_Port_Pulse_Append (port, pulse);
op_tdset_dbl (pkptr, OPC_TDA_PT_PROP_DELAY, Aon_Fib_Delay (pulse, link->fib_desc));
```c
aon_proprcv(ps.c)

/* Greg Campbell */
/* AON Model Suite */
/* Fiber prop delay. */

#include <math.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_fib.ex.h"
#include "aon_ps.ex.h"

void aonproprcv(Packet* pkptr) {
    int tx_objid;
    int link_objid;
    int num_links;
    int i;
    AonT_Fib_Link* link;
    AonT_Port_Pulse* port;
    double *last_time;
    int type;
    AonT_Pulse* pulse;
    AonT_Noise* noise;
    /* Multiple channels not supported. */
    /* int ch_index; */

    if (AonI_Fib_List_Init == 0) {
        op_prg_list_init (&AonIFib_List);
        AonI_Fib_List_Init = 1;
    }

tx_objid = op_td_get_int (pkptr, OPC_TDA_PT_TX_OBJID);
link_objid = op_td_get_int (pkptr, OPC_TDA_PT_LINK_OBJID);

    /* Multiple channels not supported. */
    /* ch_index = op_td_get_int (pkptr, OPC_TDA_PT_CH_INDEX); */

    /* Find link... */
    num_links = op_prg_list_size (&AonI_Fib_List);
    for (i = 0; i < num_links; i++) {
        
    }
```

```c
else {
    noise = Aon_Noise_Packet_Get (pkptr);
    noise->power = noise->power * exp ((-1.0) * link->fib_desc->alpha * 
        link->fib_desc->Length);

    op_td_set_dbl (pkptr, OPC_TDA_PT_PROP_DELAY, Aon_Fib_B1 ((AonI_Low_Freq 
        + (double) noise->freqbin / (double) AonI_N_Segment) * 
        (AonI_High_Freq - AonI_Low_Freq)), link->fib_desc) * 
        link->fib_desc->Length);
    
    } 
```
link = (Aon_T_Fib_Link*) op_prv_list_access (&AonI_Fib_List, i);
if (link->link_objid == link_objid)
    break;
}

if (i == num_links)
{
    /* Link not found. Instantiate new link. */
}

if (link->xmt1_objid == tx_objid)
{
    port = link->port1;
    last_time = &(link->last_time1);
}
else if (link->xmt2_objid == tx_objid)
{
    port = link->port2;
    last_time = &(link->last_time2);
}
else
{
    /* Port uninitialized. Instantiate port. */
}

type = Aon_Event_Packet_Type (pkptr);

if (type == AONC_PKT_PULSE)
{
    if (*((last_time)) != op_sim_time ()
    {
        Aon_Fib_Prop_Port (port, link->fib_desc, *((last_time)),
            op_sim_time ());
        *((last_time)) = op_sim_time ();
    }

    pulse = Aon_Fib_Exit_Pulse (port, link->fib_desc, op_sim_time ());
op_pk_nfd_set (pkptr, *data", pulse, Aon_Noop, Aon_Noop, 0);
    op_td_set_int (pkptr, OPC_TDA_PT_PK_ACCEPT, OPC_TRUE);
}
else
{
    op_td_set_int (pkptr, OPC_TDA_PT_PK_ACCEPT, OPC_TRUE);
}

---

*aon_txdel.ps.c*

/* Greg Campbell */
/* AON Model Suite */
/* Fiber trans delay. */

#include <math.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_fib.ex.h"
#include "aon_ps.ex.h"
void
aon_txdel (Packet* pkptr)
{
    op_td_set_dbl (pkptr, OPC_TDA_PT_TX_DELAY, 0.0);
}

void
aon_txrcv (Packet* pkptr)
{
    op_td_set_int (pkptr, OPC_TDA_PT_NUM_ERRORS, 0);
}

---

**aon_txrcv.ps.c**

/* Greg Campbell */
/* AON Model Suite */
/* Fiber trans delay. */

#include <math.h>
#include "cmath.h"
#include "/lidsfs/usr/local3/opnet-2.4-sol/sys/include/opnet.h"
#include "aon_base.ex.h"
#include "aon_fib.ex.h"
#include "aon_ps.ex.h"

void
aon_txrcv (Packet* pkptr)
{
    op_td_set_int (pkptr, OPC_TDA_PT_NUM_ERRORS, 0);
}
Appendix C: Usage Comments

The AON Model Suite is based on the OPNET simulation platform, and there are two things that must be done to set it up properly:

- All files with the suffix `.ex.c` must be compiled by a C compiler. The resulting `.o` files must be left in a directory listed in the OPNET mod_dirs environment variable. For the procedures within these files to be accessible, each network model that uses AON Model Suite components must have the `.ex.o` files declared as external object files. This can be done by using the "Declare external object files" button in the OPNET Network Editor.

- All AON Model Suite components must have the begsim intrpt option set at the node level. This can be done by clicking the right mouse button on all components using AON Model Suite Process Models and selecting the begsim intrpt menu item.

- Links at the network level must have the appropriate models specified. The txdel model should be `aon_txdel`, the propdel model should be `aon_propdel`, the error model should be `aon_txrcv` and the ecc model should be `aon_proprcv`. Fiber parameters can be specified as extended attributes of the link. Any fiber parameters not found in the extended attributes are assumed to have the default value.
Bibliography


