10 Radio Transceiver Pipeline

This chapter describes the Radio Transceiver Pipeline, which Modeler uses to model wireless transmission of packets. The first section provides an overview of the radio transceiver pipeline and how it operates. The second section describes the operation of the default models supplied for each stage of the transceiver pipeline.

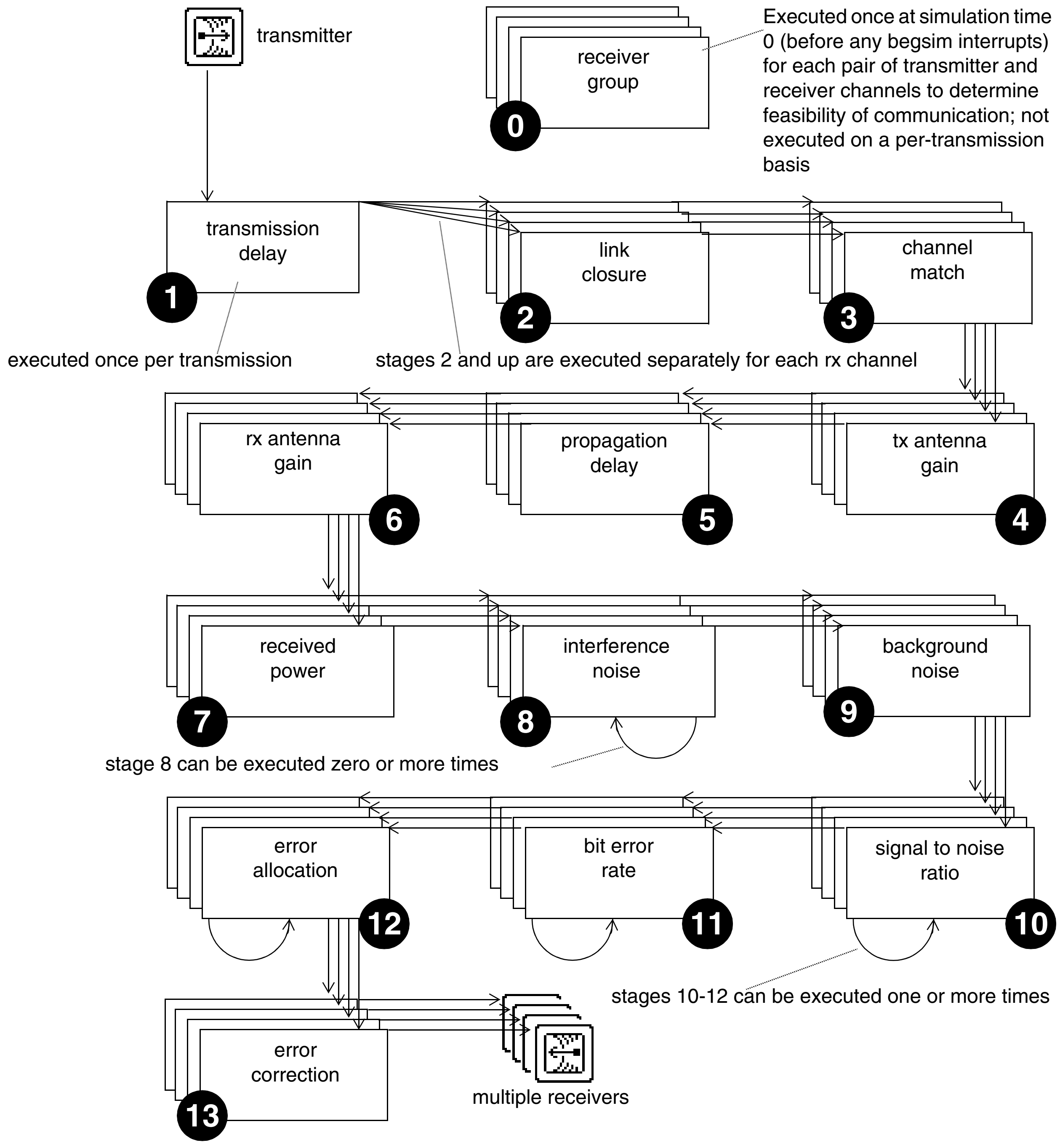
Overview

Because radio links provide a broadcast medium, each transmission can potentially affect multiple receivers throughout the network model. In addition, for a given transmission, the radio link to each receiver can exhibit different behavior and timing. As a result, a separate pipeline must be executed for each eligible receiver. This section describes the purpose of each stage; the specific operation of the default stages is described in Default Models on page WM-10-28.

The Radio Transceiver Pipeline consists of fourteen stages, most of which must be executed on a per-receiver basis whenever a transmission occurs. However, stage 0 (receiver group) is invoked only once for each pair of transmitter and receiver channels in the network, to establish a static binding between each transmitter channel and the set of receiver channels that it is allowed to communicate with. Stage 1 (transmission delay) is used to compute a result that is common to all destinations, and therefore can be executed just once per transmission. Finally, each individual pipeline sequence might not fully complete, depending on the result of stage 2 (closure), because this stage is responsible for determining if communication between the transmitter and receiver is possible on a dynamic basis. Similarly, stage 3 (channel match) might classify a transmission as irrelevant with regard to its effect on a particular receiver channel, thereby preventing the pipeline sequence from reaching the final stages.

Note from the following diagram that several of the later stages of the Radio Transceiver Pipeline might be executed multiple times for each receiver, due to interactions with multiple concurrent transmissions from other sources. To detect concurrent reception of multiple packets at the same receiver channel, the Simulation Kernel maintains two lists of “current” packets for each radio receiver channel object. The first list contains only packets that have been determined to be valid by the channel match stage; the second list contains invalid packets. Valid packets are those that meet the criteria of the channel for proper reception. Generally, this requires that the transmitter and receiver channels have matching characteristics, and possibly also that the receiver channel be able to synchronize to the packet’s signal. The primary reason for maintaining invalid and valid packets in separate lists is that this allows the Simulation Kernel to execute certain pipeline stages or to compute certain channel statistics only for valid packets. For example, there is no need to calculate signal-to-noise ratio, bit error rate, or error allocations for packets that cannot be accepted by the receiver. In general, all of the stages following the received power stage are applicable only to valid packets.

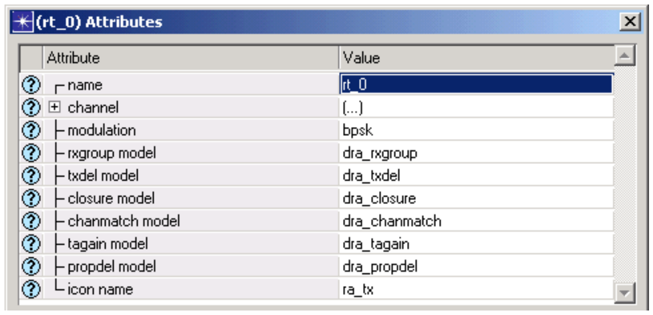
Figure 10-1 Radio Transceiver Pipeline Execution Sequence for One Transmission


transmitter
receiver group
0
Executed once at simulation time 0 (before any begsim interrupts) for each pair of transmitter and receiver channels to determine feasibility of communication; not executed on a per-transmission basis
transmission delay
1
link closure
2
channel match
3
executed once per transmission
stages 2 and up are executed separately for each rx channel
rx antenna gain
6
propagation delay
5
tx antenna gain
4
received power
7
interference noise
8
background noise
9
stage 8 can be executed zero or more times
error allocation
12
bit error rate
11
signal to noise ratio
10
stages 10-12 can be executed one or more times
error correction
13
multiple receivers

Stages 9 through 12 of the pipeline are invoked to evaluate a link's performance in response to changes in the signal condition. There is always at least one invocation of stages 10 through 12 to evaluate performance over the full duration of a valid packet. However, an additional invocation will occur for each of these stages (9–12) whenever an interfering packet arrives, to compute new signal conditions.

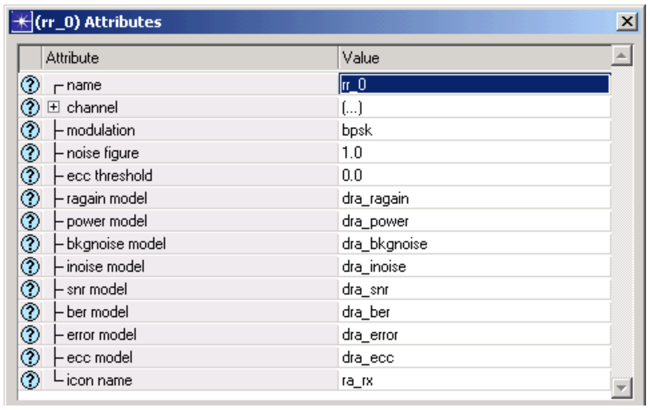
Because radio links do not exist as physical objects, the pipeline stages used to support a particular radio transmission must be associated with the radio transmitter and radio receiver that form the link. Certain stages are associated with the transmitter and others with the receiver, as shown in the following diagram. The default models used for each stage are described in Default Models on page WM-10-28.

Figure 10-2 Radio Transceiver Attributes for Specifying Pipeline Stages


(rt_0) Attributes

Attribute	Value
name	rt_0
channel	(...)
modulation	bpsk
rxgroup model	dra_rxgroup
txdel model	dra_txdel
closure model	dra_closure
chanmatch model	dra_chanmatch
tagain model	dra_tagain
propdel model	dra_propdel
icon name	ra_tx

6 Stages (0–5) associated with radio transmitter


(rr_0) Attributes

Attribute	Value
name	rr_0
channel	(...)
modulation	bpsk
noise figure	1.0
ecc threshold	0.0
ragain model	dra_ragain
power model	dra_power
bkgnoise model	dra_bkgnoise
inoise model	dra_inoise
snr model	dra_snr
ber model	dra_ber
error model	dra_error
ecc model	dra_ecc
icon name	ra_rx

8 Stages (6–13) associated with radio receiver

Reserved Transmission Data Attributes

Note—This section contains information of interest to Modeler users only.

The Simulation Kernel sets aside a number of TDAs to provide pipeline stages with access to a minimum standard set of values, and to support communication between itself and the pipeline stages as well as between the stages and each other. The TDAs reserved for these purposes in packets transmitted over radio links are defined in the following table. For compactness, the symbolic constants that represent the TDA's indices are shown in a short form without their common prefix. When actually used, each symbolic constant appears with the character sequence OPC_TDA_RA_ prepended to the names given in the table below.

Table 10-1 Radio Pipeline Reserved TDAs

Symbolic Constant	Data Type	Definition	Assigned By	Modifiable by Stages
ACTUAL_BER	double	proportion of erroneous bits in packet or segment thereof	stage 12	yes
BER	double	expected bit error probability for packet or segment thereof	stage 11	yes
BKGNOISE	double	background noise due to otherwise unmodeled sources (in watts)	stage 8	yes
CLOSURE	integer	ability of tx channel to communicate with rx channel	stage 2	yes
ECC_THRESH	double	maximum proportion of erroneous bits that receiver can correct	Kernel	yes
END_DIST	double	distance between tx and rx at time when transmission completes (in meters)	Kernel	yes
END_PROPDEL	double	radio signal propagation time between tx and rx at end of transmission (in seconds)	stage 5	yes
END_RX	double	simulation time at which packet completes reception (equal to transmission start time + transmission delay + end propagation delay)	Kernel	no
END_TX	double	simulation time at which packet completes transmission	Kernel	no
MATCH_STATUS	integer	categorization of compatibility between tx and rx channels as valid, noise, or ignored	stage 3	yes
ND_FAIL	double	earliest simulation time at which rx node failure occurred during reception of packet	Kernel (if failure occurs)	no
ND_RECOV	double	latest simulation time at which rx node recovery occurred during reception of packet	Kernel (if recovery occurs)	no
NOISE_ACCUM	double	accumulated noise power of interfering transmissions (in watts) for packet or segment thereof	stage 9	yes
NUM_COLLS	integer	number of collisions experienced by packet to date	reset by Kernel	yes

Table 10-1 Radio Pipeline Reserved TDAs (Continued)

Symbolic Constant	Data Type	Definition	Assigned By	Modifiable by Stages
NUM_ERRORS	integer	number of bit errors affecting packet to date	stage 12 and reset by Kernel	yes
PK_ACCEPT	integer	decision to accept/reject packet	stage 13	Yes
PROC_GAIN	double	processing gain provided by receiver system (in dB); used to compute effective SNR	Kernel	yes
RCVD_POWER	double	in-band power of received radio signal (in watts)	stage 7	yes
RX_BORESIGHT_PHI, RX_BORESIGHT_THETA	double	reference point on receiving antenna (in degrees); usually point of maximum gain	Kernel	yes
RX_BW	double	bandwidth of receiver channel (in Hz)	Kernel	yes
RX_CH_INDEX	integer	index of receiver channel	Kernel	no
RX_CH_OBJID	integer	object ID of radio rx channel	Kernel	no
RX_CODE	double	code used to identify receiver spreading technique and/or sequence	Kernel	yes
RX_DRATE	double	data rate of receiver channel (in bits per second)	Kernel	yes
RX_FREQ	double	base frequency of receiver channel (in Hz)	Kernel	yes
RX_GAIN	double	gain provided to signal by receiver's antenna	stage 6	yes
RX_GEO_X, RX_GEO_Y, RX_GEO_Z	double	receiver node's geocentric Cartesian coordinates (in meters)	Kernel	yes
RX_LAT, RX_LONG, RX_ALT	double	latitude, longitude (in degrees), and altitude (in meters) of receiving node	Kernel	yes
RX_MOD	pointer	reference to modulation table object to use in *op_tbl_mod_ber()*	Kernel	yes
RX_NOISEFIG	double	noise figure of receiver	Kernel	yes
RX_OBJID	integer	object ID of radio receiver	Kernel	no
RX_PATTERN	pointer	reference to antenna pattern object to use in *op_tbl_pat_gain()*	Kernel	yes
RX_PHI_POINT, RX_THETA_POINT	double	pointing angles (in degrees) based on target attributes of receiving antenna	Kernel	yes

Table 10-1 Radio Pipeline Reserved TDAs (Continued)

Symbolic Constant	Data Type	Definition	Assigned By	Modifiable by Stages
RX_REL_X, RX_REL_Y	double	receiver node's position in coordinate system of own subnet	Kernel	yes
SNR	double	ratio of signal power to noise power measured at receiver location (in dB) for packet or segment thereof	stage 10	yes
SNR_CALC_TIME	double	simulation time at which SNR was last calculated	stage 10	yes
START_DIST	double	distance between tx and rx at start of transmission (in meters)	Kernel	yes
START_PROPDEL	double	radio signal propagation time between tx and rx at start of transmission (in seconds)	stage 5	yes
START_RX	double	simulation time at which packet begins reception	Kernel	no
START_TX	double	simulation time at which packet begins transmission	Kernel	no
TX_BORESIGHT_PHI, TX_BORESIGHT_THETA	double	reference point on transmitting antenna (in degrees); usually point of maximum gain	Kernel	yes
TX_BW	double	bandwidth of transmitter channel (in Hz)	Kernel	yes
TX_CH_INDEX	integer	index of transmitter channel	Kernel	no
TX_CH_OBJID	integer	object ID of radio tx channel	Kernel	no
TX_CODE	double	code used to identify transmitter spreading technique and/or sequence	Kernel	yes
TX_DELAY	double	transmission delay of packet (in seconds)	stage 1	yes
TX_DRATE	double	data rate of transmitter channel (in bits per second)	Kernel	yes
TX_FREQ	double	base frequency of transmitter channel (in Hz)	Kernel	yes
TX_GAIN	integer	gain provided to signal by transmitter's antenna	stage 4	yes
TX_GEO_X, TX_GEO_Y, TX_GEO_Z	double	transmitter node's geocentric Cartesian coordinates (in meters)	Kernel	yes
TX_LAT, TX_LONG, TX_ALT	double	latitude, longitude (in degrees), and altitude (in meters) of transmitting node	Kernel	yes

Table 10-1 Radio Pipeline Reserved TDAs (Continued)

Symbolic Constant	Data Type	Definition	Assigned By	Modifiable by Stages
TX_MOD	pointer	reference to modulation table object to use in *op_tbl_mod_ber()*	Kernel	yes
TX_OBJID	integer	object ID of radio transmitter	Kernel	no
TX_PATTERN	pointer	reference to antenna pattern object to use in *op_tbl_pat_gain()*	Kernel	yes
TX_PHI_POINT, TX_THETA_POINT	double	pointing angles (in degrees) based on target attributes of transmitting antenna	Kernel	yes
TX_POWER	double	power of transmission (in watts)	Kernel	yes
TX_REL_X, TX_REL_Y	double	transmitter node's position in coordinate system of own subnet	Kernel	yes
End of Table 10-1				

Modeler users creating custom pipeline designs can use any higher-numbered TDAs (above the highest predefined TDA) that are not reserved by the Kernel to communicate additional information between stages. The highest reserved TDA index for the Radio Transceiver Pipeline is represented by the symbolic constant OPC_TDA_RA_MAX_INDEX. To get the lowest unreserved TDA index, add one to this constant.

Stage 0: Receiver Group

The receiver group stage is not actually part of the dynamic pipeline that processes transmissions. However, it is considered part of the pipeline because it computes results that influence the behavior of radio transmissions. It is specified by the “rxgroup model” attribute of the radio transmitter.

When a radio transmission begins, the Simulation Kernel models the broadcast nature of radio by implementing multiple radio links between the transmitting channel and a set of receiver channels. Every transmitter channel maintains its own *receiver group* of channels that are possible candidates for receiving transmissions from that object. The purpose of the receiver group stage is to create an initial receiver group for each transmitter channel. The Simulation Kernel checks every possible transmitter-receiver channel pair, and creates a receiver group for each transmitter channel.

This presents a problem, however. Because transceiver characteristics can change dynamically during a simulation, it is often difficult to determine at the outset which receiver channels are appropriate destinations for a given transmitter channel. Therefore, the receiver group stage includes a receiver channel unless it can determine, in advance, that a receiver channel will *never* be an appropriate destination for a transmitter channel. During a simulation, subsequent pipeline stages can use dynamic criteria to determine a receiver channel's eligibility for receiving a transmission. Possible reasons for disqualifying a receiver channel during simulation (despite its membership in the transmitting channel's receiver group) include:

- Disjunct frequency bands. A frequency band is defined by its base frequency and its bandwidth. If a transmitter channel and receiver channel's frequency bands do not overlap, then the transmitter channel's transmissions cannot affect the receiver channel—either as valid signals or as noise.
- Physical separation. A transmitter channel and receiver channel can be too far apart to establish a radio link with a sufficiently strong signal. Establishing a radio link also depends on factors like the elevation of the transmitter and receiver antennas and the transmission signal's power and frequency.
- Antenna nulls. The pipeline models the gain patterns of transmitter and receiver antennas in pipeline stages 4 and 6, respectively. When directional antennas are used, these stages' computed results might reduce the signal power significantly—so significantly that a simulation can reasonably ignore the effect of transmissions on the receiver channel in question.

In some network models, the Simulation Kernel can determine that certain transmitter–receiver channel pairs are absolutely unable to communicate. In such cases removing the receiver channel from the transmitter channel's receiver group leads to faster simulations, because it eliminates the execution of pipeline stages known to have no effect. Because the receiver group stage is called at simulation time 0, however, its results should not depend on factors that can change during the simulation. Examples of such potentially "iffy" criteria are the initial distance between two mobile nodes and the initial frequencies assigned to transmitter and receiver channels (assuming that a model can dynamically change these frequencies). By default, the receiver channel is included in such cases, and all receiver groups remain static throughout the simulation; subsequent pipeline stages then use dynamic criteria to evaluate transmitter–receiver channel connectivity.

The Simulation Kernel requires that the receiver group stage procedure accept the object IDs of a transmitter channel and a receiver channel, respectively. The procedure should return an integer value (OPC_TRUE or OPC_FALSE) to the Kernel; this value indicates whether the receiver channel is an eligible destination, and should be included in the receiver group. Syntax requirements for this stage are specified in Figure 10-3.

Figure 10-3 Templates for Radio Transceiver Pipeline: Receiver Group Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
int
rx_group_template (Objid tx_channel_objid, Objid rx_channel_objid)
    {
    int result;
    FIN (rx_group_template (tx_channel_objid, rx_channel_objid))

    FRET (result)
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
int
rx_group_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Objid tx_channel_objid;
Objid rx_channel_objid)

    {
    int result;

    FIN_MT (rx_group_template (tx_channel_objid, rx_channel_objid))

    FRET (result)
    }
```

You can use the Radio package of Kernel Procedures (described in Chapter 16 Radio Package on page DES-16-1) to change the default behavior of the receiver group stage, and dynamically change and recalculate receiver groups in response to simulation events. For example, you can remove a receiver channel from a receiver group if the receiver's node becomes disabled in the course of a simulation. You can use the procedure *op_radio_txch_rxgroup_compute()* to calculate or recalculate a given channel's receiver group (in essence, re-invoking the receiver group pipeline stage) at any time during a simulation. You can even "skip" the receiver group stage entirely, then create and update receiver groups as needed. (Skipping this pipeline stage involves setting the "rxgroup model" attribute to either dra_no_rxgroup or NONE instead of the default dra_rxgroup; this creates empty receiver groups for all transmitter channels.)

You can monitor changes in the number of receiver groups a receiver channel belongs to by enabling collection of the "receiver groups count" statistic for that channel (see Built-In Statistics on page WM-2-32).

Updating receiver groups dynamically can result in faster simulations, especially in network models with high levels of radio traffic. "Skipping" the pipeline stage and creating receiver groups as needed can also speed up simulations in networks with large numbers of transceiver channels.

The receiver group stage also is a convenient place to allocate memory and initialize any transmitter or receiver (channel) state information, if such state information will be used (that is, populated or accessed) in the following stages.

Stage 1: Transmission Delay

The transmission delay stage is numerically the second stage of the radio pipeline, but is the first to be executed on a dynamic basis as the result of a new transmission taking place. It is specified by the "txdel model" attribute of the radio transmitter, and is invoked immediately upon beginning transmission of a packet. This is the only stage for which a single execution is performed to support all pipelines that result from a new transmission (that is, the computation is shared by all resulting pipelines).

This stage is invoked to calculate the amount of time required for the entire packet to complete transmission. This result is the simulation time difference between the beginning of transmission of the first bit and the end of transmission of the last bit of the packet. The Simulation Kernel uses the result provided by this stage to schedule an end-of-transmission event for the transmitter channel that is used to send the packet. When this event occurs, the transmitter can begin transmission of the next packet in the channel's internal queue, if any are present; otherwise the transmitter channel becomes idle. In addition, the transmission delay result is used in conjunction with the result of the propagation delay stage to compute the time at which the packet completes reception at the link's destination (that is, the time at which the last bit finishes arriving is the time at which it finishes transmitting added to the propagation delay on the link).

The Simulation Kernel requires that the transmission delay stage procedure accept a packet address as its sole argument. The computed transmission delay is then expected by the Kernel within the TDA represented by the symbolic constant OPC_TDA_RA_TX_DELAY. The assigned value should be a nonnegative double-precision floating-point number. No other results are required. Syntax requirements for this stage are specified in Figure 10-4.

Figure 10-4 Templates for Radio Transceiver Pipeline: Transmission Delay Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
tx_delay_template (Packet* pk)
    {
    double result;

    FIN (tx_delay_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of transmission delay. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_dbl (pk, OPC_TDA_RA_TX_DELAY, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
tx_delay_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (tx_delay_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of transmission delay. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_dbl (pk, OPC_TDA_RA_TX_DELAY, result);

    FOUT
    }
```

Stage 2: Closure

The closure stage is the third stage of the pipeline, and is specified by the "closure model" attribute of the radio transmitter. It is invoked once for each receiver channel referenced in the transmitting channel's destination channel set (For information on destination channel sets, refer to Stage 0: Receiver Group on page WM-10-7.) These invocations take place immediately after the return of the transmission delay stage, with no simulation time elapsing in between. The purpose of this stage is to determine whether a particular receiver channel can be affected by a transmission. The ability of the transmission to reach the receiver channel is referred to as *closure* between the transmitter channel and the receiver channel, hence the name of the stage.

Note that the goal of the closure stage is not to determine if a transmission is valid or appropriate for a particular channel, but only if the transmitted signal can physically attain the candidate receiver channel and affect it in any way; thus, this stage applies to interfering transmissions (such as jamming) as well as to desired ones. Generally, the computations performed by this stage are based mostly on physical considerations, such as occlusion by obstacles and/or the surface of the earth.

The Simulation Kernel expects an integer value to be provided by this stage. The Kernel uses this result to determine whether to execute the remainder of the radio pipeline for the receiver channel in question. If the integer value is equal to the symbolic constant OPC_TRUE then closure is established and signal contact between the transmitter and receiver channels is possible; otherwise the value OPC_FALSE should be used to indicate that contact is not possible.

The Simulation Kernel requires that the closure stage procedure accept a packet address as its sole argument. The closure indication is then expected by the Kernel within the TDA represented by the symbolic constant OPC_TDA_RA_CLOSURE. No other results are required. Syntax requirements for this stage are specified in Figure 10-5.

Figure 10-5 Templates for Radio Transceiver Pipeline: Closure Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
closure_template (Packet* pk)
    {
    int result;

    FIN (closure_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of closure indication. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_int (pk, OPC_TDA_RA_CLOSURE, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
closure_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    int result;

    FIN_MT (closure_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of closure indication. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_int (pk, OPC_TDA_RA_CLOSURE, result);

    FOUT
    }
```

Stage 3: Channel Match

The channel match stage is the fourth stage of the pipeline, and is specified by the “chanmatch model” attribute of the radio transmitter. It is invoked once for each receiver channel that satisfies the criteria of the link closure stage. These invocations take place immediately after the return of the link closure stage, with no simulation time elapsing in between. The purpose of this stage is to classify the transmission with respect to the receiver channel. One of three possible categories must be assigned to the packet, as defined below:

- Valid. Packets in this category are considered compatible with the receiver channel and will possibly be accepted and forwarded to other modules in the receiving node, provided that they are not affected by an excessive number of errors. Classification as a valid packet usually depends at least on agreement between transmitter and receiver channels concerning the values of certain key attributes.

- Noise. This classification is used to identify packets whose data content cannot be received, but that have an impact on the receiver channel’s performance by generating interference. Packets are generally classified as noise as a result of incompatibilities between the transmitter and receiver channel configurations.

- Ignored. If a transmission is determined to have no effect whatsoever on a receiver channel’s state or performance, then it should be identified using this classification. The Simulation Kernel will then discontinue the pipeline execution between the transmitter and receiver channels for this particular transmission (future transmissions between the channels are not prevented).

The Simulation Kernel requires that the channel match stage procedure accept a packet address as its sole argument. The classification is then expected by the Kernel within the integer TDA represented by the symbolic constant OPC_TDA_RA_MATCH_STATUS. One of the symbolic constants

OPC_TDA_RA_MATCH_VALID, OPC_TDA_RA_MATCH_NOISE, or OPC_TDA_RA_MATCH_IGNORE should be used to represent the valid, noise, and ignored categories, respectively. No other results are required. Syntax requirements for this stage are specified in Figure 10-6.

Figure 10-6 Templates for Radio Transceiver Pipeline: Channel Match Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
channel_match_template (Packet* pk)
    {
    int result;

    FIN (channel_match_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of channel match classification. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_int (pk, OPC_TDA_RA_MATCH_STATUS, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
channel_match_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    int result;

    FIN_MT (channel_match_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of channel match classification. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_int (pk, OPC_TDA_RA_MATCH_STATUS, result);

    FOUT
    }
```

Stage 4: Transmitter Antenna Gain

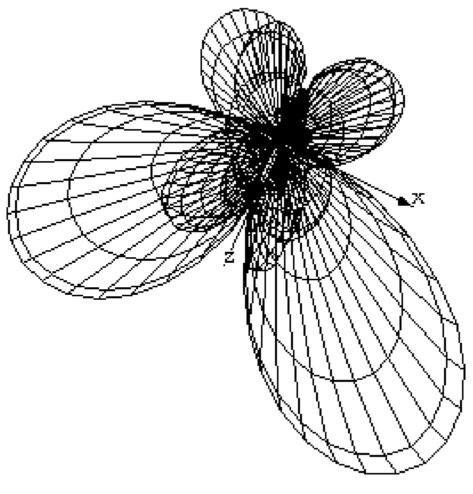
The transmitter antenna gain stage is the fifth stage of the radio transceiver pipeline, and is specified by the “tagain model” attribute of the radio transmitter. It is executed separately for each destination channel, except those that failed the link closure stage and those with respect to which the channel match stage has classified the transmission as “ignored”. Invocation of the transmitter antenna gain stage takes place immediately after return from the channel match stage with no simulation time elapsing in between. The purpose of the transmitter antenna gain stage is to compute the gain provided by the transmitter’s associated antenna, based on the direction of the vector leading from the transmitter to the receiver. The Simulation Kernel does not itself use this result, but it is typically factored into the received power computation of stage 7.

Antenna gain characterizes the phenomenon of magnification or reduction of the transmitted signal energy in a manner which depends on the direction of the signal path. This “shaping” of the transmitted energy is due to the physical characteristics of the antenna structure and possible phase manipulations of the signal in certain antennas consisting of multiple radiating elements.

Antennas that provide no gain to a transmitted signal in any direction are referred to as *isotropic*, because they have a perfectly symmetric behavior with respect to all possible signal paths. The gain of an antenna in a particular direction is measured in comparison to an isotropic antenna. It is defined as the ratio of signal power produced by the antenna at a given distance and the isotropic power that would be measured at the same distance. Gain is a unitless quantity that is usually given in decibels (dB).

By measuring or computing gain over a range of directions, a three-dimensional antenna pattern can be constructed that characterizes the antenna’s effect with respect to the various directions of transmission. Tools for calculation of antenna patterns are not provided, but the Antenna Pattern Editor supports the capture of antenna pattern data in an empirical form (i.e., the pattern must already be known and can be entered in numerical form). In addition, the EMA programmatic interface supports specification of antenna pattern data by a C language program, allowing patterns to be formed by analytical techniques or database retrieval (that is, other antenna-specific applications can be used to generate pattern data that can be used by simulations). Regardless of the method used to capture antenna pattern data, the Antenna Pattern Editor displays the pattern in 3-D, as shown in the following example. Refer to Chapter 8 Antenna Pattern Editor on page WM-8-1 chapter for more information about the Antenna Pattern Editor.

Figure 10-7 Three-Dimensional Antenna-Gain Pattern

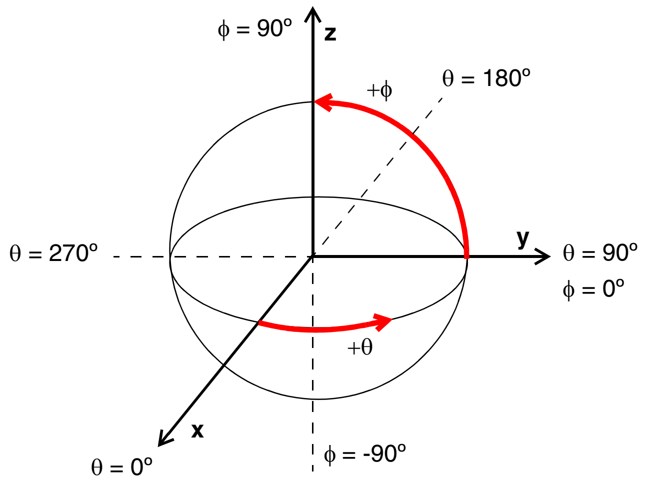


In general, the transmitter antenna gain stage first calculates the direction vector separating the transmitter and the receiver and then uses knowledge of the antenna pattern to determine the antenna gain provided to the transmission. Pointing of the antenna is typically also accounted for by this pipeline stage. *Pointing* is controlled by a set of antenna attributes. These include pointing ref.phi and pointing ref.theta, which designate a particular part of the antenna pattern that is to be directed at a selected point in space. Pipeline stages can obtain these attributes of the transmitter antenna via the TDAs OPC_TDA_RA_TX_BORESIGHT_PHI and OPC_TDA_RA_TX_BORESIGHT_THETA.

In addition, the antenna attributes target latitude, target longitude, and target altitude define a location in space at which the antenna's reference point is to be directed. Pipeline stages can obtain the values of these attributes directly from the antenna object, if necessary. However, the Simulation Kernel provides information derived from these attributes, which is in many cases sufficient for antenna gain calculations; this information is contained in the TDAs OPC_TDA_RA_TX_PHI_POINT and OPC_TDA_RA_TX_THETA_POINT, which represent the direction angles ϕ and θ of the vector separating the transmitter and the target location. These angles are not to be confused with the ϕ and θ angles used to define the antenna pattern. Rather, these target pointing angles are defined relative to the geocentric Cartesian coordinate system (see Geocentric Coordinate System on page MC-7-27).

The following figure illustrates the meaning of these angles within this coordinate system.

Figure 10-8 Relationship of Pointing Angles Phi and Theta to Geocentric Cartesian Coordinate System (Right-Sphere)



The Simulation Kernel reserves a TDA, represented by the symbolic constant OPC_TDA_RA_TX_GAIN, to store the result of the transmitter antenna gain stage and to make it available to later stages of the pipeline. The Simulation Kernel requires that the stage's procedure accept a packet address as its sole argument. Syntax requirements for this stage are specified in Figure 10-9.

Figure 10-9 Templates for Radio Transceiver Pipeline: Transmitter Antenna Gain Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
tx_antenna_gain_template (Packet* pk)
    {
    double result;

    FIN (tx_antenna_gain_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of tx antenna gain. */

    /* place result in TDA to make available for later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_TX_GAIN, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
tx_antenna_gain_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (tx_antenna_gain_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of tx antenna gain. */

    /* place result in TDA to make available for later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_TX_GAIN, result);

    FOUT
    }
```

Stage 5: Propagation Delay

The propagation delay stage is the sixth stage of the radio transceiver pipeline, and is specified by the "propdel model" attribute of the radio transmitter. It is invoked for each receiver channel that successfully passed both the link closure and channel match stages. This invocation takes place after the return of the transmitter antenna gain stage with no simulation time elapsing in between. The purpose of this stage is to calculate the amount of time required for the packet's signal to travel from the radio transmitter to the radio receiver. This result is generally dependent on the distance between the source and the destination. The Kernel uses this result to schedule a beginning-of-reception event for the receiver channel that the packet is destined for. In addition, the propagation delay result is used in conjunction with the result of the transmission delay stage to compute the time at which the packet completes reception (i.e., the time at which the last bit finishes arriving is the time at which the packet begins transmission added to the sum of the transmission delay and the propagation delay).

The Simulation Kernel requires that the propagation delay stage procedure accept a packet address as its sole argument. The computed propagation delay is then expected by the Kernel within the TDA represented by the symbolic constants OPC_TDA_RA_START_PROPDEL and OPC_TDA_RA_END_PROPDEL. The assigned values should be nonnegative double-precision floating-point numbers. No other results are required. Syntax requirements for this stage are specified in Figure 10-10.

Figure 10-10 Templates for Radio Transceiver Pipeline: Propagation Delay Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
prop_delay_template (Packet* pk)
    {
    double start_prop_delay, end_prop_delay;

    FIN (prop_delay_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of propagation delay. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_dbl (pk, OPC_TDA_RA_START_PROPDEL, start_prop_delay);
    op_td_set_dbl (pk, OPC_TDA_RA_END_PROPDEL, end_prop_delay);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
prop_delay_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double start_prop_delay, end_prop_delay;

    FIN_MT (prop_delay_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of propagation delay. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_dbl (pk, OPC_TDA_RA_START_PROPDEL, start_prop_delay);
    op_td_set_dbl (pk, OPC_TDA_RA_END_PROPDEL, end_prop_delay);

    FOUT
    }
```

Stage 6: Receiver Antenna Gain

The receiver antenna gain stage is the seventh stage of the radio transceiver pipeline. It is the earliest stage associated with the radio receiver rather than the transmitter, being specified by the receiver's "ragain model" attribute. It is executed separately for each eligible destination channel, and its invocation takes place at the time that the leading edge of the packet arrives at the receiver location (i.e., after the propagation delay has elapsed).

The purpose of the receiver antenna gain stage is to compute the gain provided by the receiver's associated antenna, based on the direction of the vector leading from the receiver to the transmitter. The Simulation Kernel does not itself use this result, but it is typically factored into the received power computation of stage 7.

The concept of receiver antenna gain is identical to that of transmitter antenna gain, except that it is due to the physical configuration and implementation of the antenna associated with the receiver object. For information on antenna gain and antenna patterns, refer to Stage 4: Transmitter Antenna Gain on page WM-10-13.

The Simulation Kernel reserves a TDA, represented by the symbolic constant OPC_TDA_RA_RX_GAIN, to store the result of this stage and to make it available to later stages of the pipeline. The Simulation Kernel requires that the stage's procedure accept a packet address as its sole argument. Syntax requirements for this stage are specified in Figure 10-11.

Figure 10-11 Templates for Radio Transceiver Pipeline: Receiver Antenna Gain Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
rx_antenna_gain_template (Packet* pk)
    {
    double result;

    FIN (rx_antenna_gain_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of rx antenna gain. */

    /* place result in TDA to make available for later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_RX_GAIN, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
rx_antenna_gain_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (rx_antenna_gain_template (Packet* pk))

    /* extract required information from packet. */
    /* perform calculation of rx antenna gain. */

    /* place result in TDA to make available for later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_RX_GAIN, result);

    FOUT
    }
```

Stage 7: Receiver Power

The receiver power stage is the eighth stage of the radio transceiver pipeline, and is specified by the “power model” attribute of the radio receiver. It is executed separately for each eligible destination channel; invocation takes place immediately after return of the receiver antenna gain model, with no simulation time elapsing in between. The purpose of this stage is to compute the received power of the arriving packet’s signal (in watts).

For packets that are classified as valid, the received power result is a key factor in determining the ability of the receiver to correctly capture the information in the packet. For packets that are classified as noise, received power still must usually be evaluated to support calculation of relative strengths of valid and noise packets. (Packets are classified as valid, invalid, or ignored by the channel match model. For more information about this classification, see Stage 3: Channel Match on page WM-10-12.)

In general, the calculation of received power is based on factors such as the power of the transmitter, the distance separating the transmitter and the receiver, the transmission frequency, and transmitter and receiver antenna gains. The Simulation Kernel requires that the received power stage procedure accept a packet address as its sole argument. The computed received power is then expected by the Kernel within the TDA represented by the symbolic constant OPC_TDA_RA_RCVD_POWER. The assigned value should be a nonnegative double-precision floating-point number. No other results are required. Syntax requirements for this stage are specified in Figure 10-12.

Figure 10-12 Templates for Radio Transceiver Pipeline: Received Power Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
received_power (Packet* pk)
    {
    double result;

    FIN (received_power (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of received power (in watts). */

    /* place result in TDA for Kernel & later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_RCVD_POWER, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
received_power_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (received_power (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of received power (in watts). */

    /* place result in TDA for Kernel & later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_RCVD_POWER, result);

    FOUT
    }
```

Stage 8: Interference Noise

The interference noise stage is the ninth stage of the radio transceiver pipeline, and is specified by the “inoise model” attribute of the radio receiver. It might be executed for a packet under two circumstances: the packet is valid and arrives at its destination channel while another packet is already being received, or the packet is valid and already being received when another packet (valid or invalid) arrives. Clearly, the first circumstance can occur at most once for each packet, and the second can occur any number of times depending upon the transmission activities of other transmitters in the model. Note that a single invocation of this stage can be shared by the pipelines of the two packets if both are valid (the call syntax appearing below provides the addresses of both packets to the interference stage so that mutual effects can be evaluated).

The purpose of this stage is to account for the interactions between transmissions that arrive concurrently at the same receiver channel. The Simulation Kernel reserves a TDA, represented by the symbolic constant OPC_TDA_RA_NOISE_ACCUM, for the purpose of storing the current level of noise from all interfering transmissions. This accumulator is maintained only for valid packets (as determined by the channel match stage) because there is generally no need to evaluate link quality for noise packets. Thus, the interference noise stage is expected to augment the value of this accumulator in each valid packet by the received power of the interfering packet. When a packet (valid or invalid) completes reception, the Kernel automatically subtracts its received power from the noise accumulator of all valid packets that are still arriving at the channel. In this manner, the accumulator reflects only the current noise level.

The Kernel requires that the interference noise stage procedure accept two packet addresses as arguments. The first packet address represents the packet that arrived earliest, and the second represents the new arrival that caused the stage to be invoked. Syntax requirements for this stage are specified in Figure 10-13.

Figure 10-13 Templates for Radio Transceiver Pipeline: Interference Noise Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
interference_noise_template (Packet* earlier_pk, Packet* later_pk)
    {
    int       e_status, l_status;
    double    e_power, l_power;

    FIN (interference_noise_template (earlier_pk, later_pk))

    /* extract match status and received power from each packet */
    e_status = op_td_get_int (earlier_pk, OPC_TDA_RA_MATCH_STATUS);
    e_power = op_td_get_dbl (earlier_pk, OPC_TDA_RCVD_POWER);
    l_status = op_td_get_int (later_pk, OPC_TDA_RA_MATCH_STATUS);
    l_power = op_td_get_dbl (later_pk, OPC_TDA_RCVD_POWER);

    /* if earlier packet is valid, augment noise accumulator */
    if (e_status == OPC_TDA_RA_MATCH_VALID)
        op_td_increment_dbl (earlier_pk, OPC_TDA_RA_NOISE_ACCUM, l_power);

    /* Similarly, if later packet is valid, */
    /* augment its noise accumulator */
    if (l_status == OPC_TDA_RA_MATCH_VALID)
        op_td_increment_dbl (later_pk, OPC_TDA_RA_NOISE_ACCUM, e_power);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
interference_noise_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* earlier_pk,
    Packet* later_pk)
    {
    int       e_status, l_status;
    double    e_power, l_power;

    FIN_MT (interference_noise_template (earlier_pk, later_pk))

    /* extract match status and received power from each packet */
    e_status = op_td_get_int (earlier_pk, OPC_TDA_RA_MATCH_STATUS);
    e_power = op_td_get_dbl (earlier_pk, OPC_TDA_RCVD_POWER);
    l_status = op_td_get_int (later_pk, OPC_TDA_RA_MATCH_STATUS);
    l_power = op_td_get_dbl (later_pk, OPC_TDA_RCVD_POWER);

    /* if earlier packet is valid, augment noise accumulator */
    if (e_status == OPC_TDA_RA_MATCH_VALID)
        op_td_increment_dbl (earlier_pk, OPC_TDA_RA_NOISE_ACCUM, l_power);

    /* Similarly, if later packet is valid, */
    /* augment its noise accumulator */
    if (l_status == OPC_TDA_RA_MATCH_VALID)
        op_td_increment_dbl (later_pk, OPC_TDA_RA_NOISE_ACCUM, e_power);

    FOUT
    }
```

In addition to the OPC_TDA_RA_NOISE_ACCUM TDA, the Kernel provides a TDA to keep track of the number of collisions experienced by each packet. This TDA is represented by the symbolic constant OPC_TDA_RA_NUM_COLLS and is provided for convenience in developing application-specific pipeline stages that might need this result when deciding whether to accept or reject a packet in the final stages of the pipeline. Although the interference stage is an appropriate location to update this TDA's value, the Kernel makes no use of this TDA and it is therefore not mandatory to maintain its accuracy.

Stage 9: Background Noise

The background noise stage is the tenth stage of the radio transceiver pipeline, and is specified by the “bkgnoise model” attribute of the radio receiver. It is executed immediately after return of the received power stage, with no simulation time elapsing in between. The purpose of this stage is to represent the effect of all noise sources except for other concurrently arriving transmissions (because these are accounted for by the interference noise stage). The expected result is the sum of the power (in watts) of other noise sources, measured at the receiver’s location and in the receiver channel’s band. Typical background noise sources include thermal or galactic noise, emissions from neighboring electronics, and otherwise unmodeled radio transmissions (such as commercial radio, amateur radio, or television, depending on frequency).

The Simulation Kernel does not itself make use of the result of this stage, but it does reserve a TDA, represented by the symbolic constant OPC_TDA_RA_BKGNOISE, for the purpose of storing the result and communicating it to later pipeline stages. Normally, the background noise value is later added to other noise sources to compute a total noise level in the signal-to-noise ratio stage. The Kernel requires that the background noise stage procedure accept a packet address as its sole argument. Syntax requirements for this stage are specified in Figure 10-14.

Figure 10-14 Templates for Radio Transceiver Pipeline: Background Noise Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
bkg_noise_template (Packet* pk)
    {
    double result;

    FIN (bkg_noise_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of background noise power (in watts). */

    /* place result in TDA to make available to later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_BKGNOISE, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
bkg_noise_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (bkg_noise_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of background noise power (in watts). */

    /* place result in TDA to make available to later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_BKGNOISE, result);

    FOUT
    }
```

Stage 10: Signal-to-Noise Ratio

The signal-to-noise ratio (SNR) stage is the eleventh stage of the radio transceiver pipeline, and is specified by the “snr model” attribute of the radio receiver. It can be executed for a valid packet (as determined by the channel match stage) under three circumstances:

- the packet arrives at its destination channel
- the packet is already being received and another packet (valid or invalid) arrives
- the packet is already being received and another packet (valid or invalid) completes reception

Clearly, the first circumstance occurs exactly once for each packet, and the second and third can occur any number of times depending upon the transmission activities of other transmitters in the model. The three types of invocations define intervals over which a packet’s average power SNR is taken to be constant (of course, this is an approximation when mobility is involved, because SNR would be continuously varying).

The purpose of SNR stage is to compute the current average power SNR result for the arriving packet. This calculation is usually based on values obtained during earlier stages, including received power, background noise, and interference noise. The SNR of the packet is an important performance measure that supports determination of the receiver’s ability to correctly receive the packet’s content. The result computed by this stage is used by the Kernel to update standard output results of receiver channels and usually also by later stages of the pipeline.

The Simulation Kernel requires that the SNR stage procedure accept a packet address as its sole argument. The computed SNR is then expected by the Kernel within the TDA represented by the symbolic constant OPC_TDA_RA_SNR. The assigned value should be a double-precision floating-point number expressed in decibels (dB). No other results are required. Syntax requirements for this stage are specified in Figure 10-15.

Figure 10-15 Templates for Radio Transceiver Pipeline: Signal-to-Noise Ratio Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
snr_template (Packet* pk)
    {
    double result;

    FIN (snr_template (Packet* pk))
    /* extract required information from packet. */

    /* perform calculation of average-power SNR (in dB). */

    /* place result in TDA to make available to Kernel & later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_SNR, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
snr_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (snr_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of average-power SNR (in dB). */

    /* place result in TDA to make available to Kernel & later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_SNR, result);

    FOUT
    }
```

Stage 11: Bit Error Rate

The bit error rate (BER) stage is the twelfth stage of the radio transceiver pipeline, and is specified by the "ber model" attribute of the radio receiver. It can be executed for a valid packet (as determined by the channel match stage) under three circumstances:

- the packet completes reception at its destination channel
- the packet is already being received and another packet (valid or invalid) arrives
- the packet is already being received and another packet (valid or invalid) completes reception

These circumstances correspond to the ends of periods during which the packet's SNR is taken to be constant. (For more information on constant-SNR intervals, refer to Stage 10: Signal-to-Noise Ratio on page WM-10-23.)

The purpose of the BER stage is to derive the probability of bit errors during the past interval of constant SNR. This is not the empirical rate of bit errors, but the expected rate, usually based on the SNR. In general, the bit error rate provided by this stage is also a function of the type of modulation used for the transmitted signal.

The Simulation Kernel requires that the BER stage procedure accept a packet address as its sole argument. The computed BER is then expected by the Kernel within the TDA represented by the symbolic constant OPC_TDA_RA_BER. The assigned value should double-precision floating-point number between zero and one (inclusive). No other results are required. Syntax requirements for this stage are specified in Figure 10-16.

Figure 10-16 Templates for Radio Transceiver Pipeline: Bit-Error-Rate Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
ber_template (Packet* pk)
    {
    double result;

    FIN (ber_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of bit-error-rate. */

    /* place result in TDA for Kernel & later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_BER, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
ber_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    double result;

    FIN_MT (ber_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of bit-error-rate. */

    /* place result in TDA for Kernel & later stages. */
    op_td_set_dbl (pk, OPC_TDA_RA_BER, result);

    FOUT
    }
```

Stage 12: Error Allocation

The error allocation stage is the thirteenth stage of the radio transceiver pipeline, and is specified by the “error model” attribute of the radio receiver. It is always executed immediately upon return from the bit error rate stage. The purpose of the error allocation stage is to estimate the number of bit errors in a packet segment where the bit error probability has been calculated and is constant. This segment might be the entire packet, if no changes in bit error probability occur over the course of the packet’s reception. Bit error count estimation is usually based on the bit error probability (obtained from stage 11) and the length of the affected segment.

The Simulation Kernel requires that the error allocation stage procedure accept a packet address as its sole argument. The Kernel maintains only a bit error accumulator TDA within the packet; therefore, the pipeline stage is expected to add the number of new errors into the existing value found in the integer TDA represented by the symbolic constant OPC_TDA_RA_NUM_ERRORS. The added value should be between zero and the number of bits in the affected packet segment. In addition, the Kernel expects the error allocation stage to provide the empirical bit-error-rate over the packet segment; this value is obtained by dividing the number of bit errors in the segment by the size of the segment. The empirical bit error rate should be placed in the double-precision TDA represented by the symbolic constant OPC_TDA_RA_ACTUAL_BER. The Kernel uses this value to update the bit error rate result of the receiver channel object. Syntax requirements for the error allocations stage are specified in Figure 10-17.

Figure 10-17 Templates for Radio Transceiver Pipeline: Error Allocation Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
err_alloc_template (Packet* pk)
    {
    int       num_added_errs, segment_length;
    double    empirical_ber;

    FIN (err_alloc_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of number of bit errors in segment. */

    /* Add errors to bit-error accumulator. */
    op_td_increment_int (pk, OPC_TDA_RA_NUM_ERRORS, num_added_errs);

    op_td_set_dbl (pk, OPC_TDA_RA_ACTUAL_BER,
        num_added_errs / segment_length);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
err_alloc_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    int       num_added_errs, segment_length;
    double    empirical_ber;

    FIN_MT (err_alloc_template (Packet* pk))

    /* extract required information from packet. */

    /* perform calculation of number of bit errors in segment. */

    /* Add errors to bit-error accumulator. */
    op_td_increment_int (pk, OPC_TDA_RA_NUM_ERRORS, num_added_errs);

    op_td_set_dbl (pk, OPC_TDA_RA_ACTUAL_BER,
        num_added_errs / segment_length);

    FOUT
    }
```

Stage 13: Error Correction

The error correction stage is the fourteenth and final stage of the pipeline, and is specified by the “ecc model” attribute of the radio receiver. It is invoked when a packet completes reception, immediately after the final return of the error allocation stage, with no simulation time elapsing in between. Exactly one invocation of this stage occurs for each packet that is considered valid (as determined by the channel match stage). The purpose of this stage is to determine whether or not the arriving packet can be accepted and forwarded via the channel’s corresponding output stream to one of the receiver’s neighboring modules in the destination node. This is usually dependent upon whether the packet has experienced collisions, the result computed in the error allocation stage, and the ability of the receiver to correct the errors affecting the packet (hence the name of the stage). Based on the determination of this stage, the Kernel will either destroy the packet, or allow it to proceed into the destination node. In addition, this result affects error and throughput results collected for the receiver channel.

The Simulation Kernel requires that the error correction stage procedure accept a packet address as its sole argument. The determination to accept or reject the packet is then expected by the Kernel within the TDA represented by the symbolic constant OPC_TDA_RA_PK_ACCEPT. The assigned value should be an integer equal to the constant OPC_TRUE to indicate acceptance; otherwise the integer value OPC_FALSE should be assigned to indicate rejection. No other results are required. Interface requirements for this stage are specified in Figure 10-18.

Figure 10-18 Templates for Radio Transceiver Pipeline: Error Correction Stage

Not MT-safe (will execute sequentially in a parallel simulation):

```
void
error_correction_template (Packet* pk)
    {
    int result;

    FIN (error_correction_template (Packet* pk))

    /* extract required information from packet. */

    /* determine if packet should be accepted or rejected. */

    /* place result in TDA to return to Simulation Kernel. */

    op_td_set_int (pk, OPC_TDA_RA_PK_ACCEPT, result);

    FOUT
    }
```

MT-safe (can execute in parallel in a parallel simulation):

```
void
error_correction_template_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet* pk)
    {
    int result;

    FIN_MT (error_correction_template (Packet* pk))

    /* extract required information from packet. */
    /* determine if packet should be accepted or rejected. */

    /* place result in TDA to return to Simulation Kernel. */
    op_td_set_int (pk, OPC_TDA_RA_PK_ACCEPT, result);

    FOUT
    }
```

Default Models

This section contains detailed descriptions of the Radio Transceiver Pipeline default models supplied with Modeler.

Modeler supports dynamic modeling of radio links between communication nodes. The characteristics and availability of these links can be subject to time varying factors such as the mobility of communication nodes, the modification of transmitter and receiver attributes, and the interference of other concurrent transmissions.

The Simulation Kernel relies on a 14-stage computational pipeline to evaluate the characteristics of radio communication when transmissions occur. Default procedures are provided to occupy each stage of the pipeline. These are intended to be useful for a variety of common radio situations, and can be replaced by user-supplied procedures for other cases.

Table 10-2 Radio Transceiver Pipeline Default Stages

Pipeline Stage	Attribute Name	Default Stage Name
0: Receiver Group	rxgroup model	dra_rxgroup dra_rxgroup_no_rxstate [1]
1: Transmission Delay	txdel model	dra_txdel
2: Link Closure	closure model	dra_closure
3: Channel Match	chanmatch model	dra_chanmatch
4: Transmitter Antenna Gain	tagain model	dra_tagain
5: Propagation Delay	propdel model	dra_propdel
6: Receiver Antenna Gain	ragain model	dra_ragain
7: Received Power	power model	dra_power dra_power_no_rxstate [1]
8: Interference Noise	inoise model	dra_inoise
9: Background Noise	bkgnoise model	dra_bkgnoise
10: Signal-to-Noise Ratio	snr model	dra_snr

Table 10-2 Radio Transceiver Pipeline Default Stages (Continued)

Pipeline Stage	Attribute Name	Default Stage Name
11: Bit Error Rate	ber model	dra_ber
12: Error Allocation	error model	dra_error
13: Error Correction	ecc model	dra_ecc dra_ecc_no_rxstate [1]
End of Table 10-2		

1. These stages can be used as defaults if no receiver channel state information is used.

The "None" Pipeline Stage

If some of the pipeline stages are not important to the goals of your simulation, you can skip these stages. To skip a pipeline stage, assign "NONE" to the pipeline stage attribute in a transmitter or receiver module. During simulation, the Simulation Kernel skips each stage set to "NONE" and sets the corresponding TDA to a default value in place of a calculated result. The following table lists the default values used by the Radio Transceiver Pipeline.

Table 10-3 Radio Transceiver Pipeline Default Values

Pipeline Stage	TDA	Default Value
0: Receiver Group	none	empty receiver group [1]
1: Transmission Delay	OPC_TDA_RA_TX_DELAY	0 seconds
2: Link Closure	OPC_TDA_RA_CLOSURE	OPC_TRUE
3: Channel Match	OPC_TDA_RA_MATCH_STATUS	OPC_TDA_RA_MATCH_VALID
4: Transmitter Antenna Gain	OPC_TDA_RA_TX_GAIN	0 dB
5: Propagation Delay	OPC_TDA_RA_START_PROPDEL OPC_TDA_RA_END_PROPDEL	0 seconds 0 seconds
6: Receiver Antenna Gain	OPC_TDA_RA_RX_GAIN	0 dB
7: Received Power	OPC_TDA_RA_RCVD_POWER	(transmitter power) times (transmitter antenna gain) times (receiver antenna gain) [2]
8: Interference Noise	none	no interference with other packets
9: Background Noise	OPC_TDA_RA_BKGNOISE	0 watts
10: Signal-to-Noise Ratio	OPC_TDA_RA_SNR	1000 dB [3]

Table 10-3 Radio Transceiver Pipeline Default Values (Continued)

Pipeline Stage	TDA	Default Value
11: Bit Error Rate	OPC_TDA_RA_BER	0
12: Error Allocation	OPC_TDA_RA_NUM_ERRORS OPC_TDA_RA_ACTUAL_BER	0 OPC_TDA_RA_BER
13: Error Correction	OPC_TDA_RA_PK_ACCEPT	OPC_TRUE
End of Table 10-3		

1. Same behavior as using the dra_no_rxgroup model.
2. Calculated as OPC_TDA_RA_TX_POWER $\times 10^{\text{OPC\_TDA\_RA\_TX\_GAIN}} \times 10^{\text{OPC\_TDA\_RA\_RX\_GAIN}}$.
3. Arbitrary very large value, equivalent to no noise.

You can change the default value assigned by a NONE pipeline stage by setting the value of the corresponding simulation preference, as listed in Table 10-4.

Table 10-4 Simulation Preferences for Radio TDA Defaults

Pipeline Stage	TDA	Simulation Preference
0: Receiver Group	none	none
1: Transmission Delay	OPC_TDA_RA_TX_DELAY	psnone_tda.ra.txdel.tx_delay
2: Link Closure	OPC_TDA_RA_CLOSURE	psnone_tda.ra.closure.closure
3: Channel Match	OPC_TDA_RA_MATCH_STATUS	psnone_tda.ra.chanmatch.match_status
4: Transmitter Antenna Gain	OPC_TDA_RA_TX_GAIN	psnone_tda.ra.tagain.tx_gain
5: Propagation Delay	OPC_TDA_RA_START_PROPDEL OPC_TDA_RA_END_PROPDEL	psnone_tda.ra.propdel.start_propdel psnone_tda.ra.propdel.end_propdel
6: Receiver Antenna Gain	OPC_TDA_RA_RX_GAIN	psnone_tda.ra.ragain.rx_gain
7: Received Power	OPC_TDA_RA_RCVD_POWER	psnone_tda.ra.power.rcvd_power [1]
8: Interference Noise	none	none
9: Background Noise	OPC_TDA_RA_BKGNOISE	psnone_tda.ra.bkgnoise.bkgnoise
10: Signal-to-Noise Ratio	OPC_TDA_RA_SNR	psnone_tda.ra.snr.snr

Table 10-4 Simulation Preferences for Radio TDA Defaults (Continued)

Pipeline Stage	TDA	Simulation Preference
11: Bit Error Rate	OPC_TDA_RA_BER	psnone_tda.ra.ber.ber
12: Error Allocation	OPC_TDA_RA_NUM_ERRORS	psnone_tda.ra.error.num_errors
	OPC_TDA_RA_ACTUAL_BER	psnone_tda.ra.error.actual_ber
13: Error Correction	OPC_TDA_RA_PK_ACCEPT	psnone_tda.ra.ecc.pk_accept
End of Table 10-4		

1. Values greater than or equal to 0.0 are used as-is. Values less than 0.0 result in the default computation of OPC_TDA_RA_TX_POWER $\times 10^{\mathrm{OPC\_TDA\_RA\_TX\_GAIN}} \times 10^{\mathrm{OPC\_TDA\_RA\_RX\_GAIN}}$.

Default Receiver Group Model (dra_rxgroup)

This section describes the functionality and implementation of the default model occupying stage zero of the Radio Transceiver Pipeline. This model is named dra_rxgroup. The source code is provided in the file `dra_rxgroup.ps.c`, which resides in the `<reldir>/models/std/wireless` directory. An alternate version of this stage, for networks not using receiver channel state information, is in the file `dra_rxgroup_no_rxstate.ps.c`.

Invocation Context

The Receiver Group Model pipeline stage is invoked once at simulation time 0 (before any begsim interrupts) to evaluate connectivity between each radio transmitter channel and each radio receiver channel in the network. It is different from other pipeline stages in that its invocation does not dynamically occur with each new packet transmission, nor does it operate on packets. (However, you can use the Radio package of Kernel Procedures to dynamically change and recalculate a transmitter channel's receiver group in response to simulation events.) This initialization stage expects two arguments: the object ID of a transmitter channel and the object ID of a receiver channel.

Because this stage is invoked at simulation time 0, it is used to initialize the state information for each receiver channel. The receiver channel state information is used to pass information between stages in the default radio pipeline models.

The result of the Receiver Group Model invocation for a transmitter and receiver channel is a single integer value, interpreted as a Boolean (i.e., OPC_TRUE or OPC_FALSE) which indicates if there is the potential for communication between the transmitter channel and the receiver channel.

Receiver Eligibility

The purpose of the Receiver Group Model is to filter out ineligible receiver channels with respect to each transmitter channel, thus improving simulation performance. In the default model, dra_rxgroup, all receivers are considered as potential destinations. That is, this model always returns OPC_TRUE indicating that the receiver channel and transmitter channel have the potential to communicate. The source code for dra_rxgroup is shown in the following listing.

Figure 10-19 Source Code for dra_rxgroup Model

```
/* dra_rxgroup.ps.c */
/* Default receiver group model for radio link Transceiver */
/* Pipeline. This model populates the state information of */
/* the receiver channels to be used in power and ecc       */
/* pipeline stage models. If you don't want the receiver   */
/* channel state information to be set and used by radio   */
/* pipeline stage models, then use "*_no_rxstate" version  */
/* of rxgroup, power and ecc stage models in your node     */
/* models.                                                 */

/****************************************/
/*      Copyright (c) 1993-2004        */
/*     by OPNET Technologies, Inc.     */
/*      (A Delaware Corporation)       */
/*    7255 Woodmont Av., Suite 250     */
/*     Bethesda, MD 20814, U.S.A.      */
/*        All Rights Reserved.         */
/****************************************/

#include "opnet.h"
#include "dra.h"

#if defined (__cplusplus)
extern "C"
#endif


int
dra_rxgroup_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Objid PRG_ARG_UNUSED(tx_obid),
    Objid rx_obid)
    {
    DraT_Rxch_State_Info*rxch_state_ptr;

    /** Determine the potential for communication between   **/
    /** given transmitter and receiver channel objects.     **/
    /** Also create and initialize the receiver channel's   **/
    /** state information to be used by other pipeline      **/
    /** stages during the simulation.                       **/
    FIN_MT (dra_rxgroup (tx_obid, rx_obid));

    /* Unless it is already done, initialize the receiver    */
    /* channel's state information.*/
    if (op_ima_obj_state_get (rx_obid) == OPC_NIL)
        {
#if defined (OPD_PARALLEL)
        /* Channel state information doesn't exist. Lock     */
        /* the global mutex before continuing.               */
        op_prg_mt_global_lock ();

        /* Check again since another thread may have         */
        /* already set up the state information.             */
        if (op_ima_obj_state_get (rx_obid) == OPC_NIL)
            {
#endif /* OPD_PARALLEL */
            /* Create and set the initial state information  */
            /* for the receiver channel. State information   */
            /* is used by other pipeline stages to           */
            /* access/update channel specific data           */
            /* efficiently.                                  */
            rxch_state_ptr = (DraT_Rxch_State_Info *)
                op_prg_mem_alloc (sizeof (DraT_Rxch_State_Info));
            rxch_state_ptr->signal_lock = OPC_FALSE;
            op_ima_obj_state_set (rx_obid, rxch_state_ptr);
#if defined (OPD_PARALLEL)
            }

        /* Unlock the global mutex.                */
        op_prg_mt_global_unlock ();
#endif /* OPD_PARALLEL */
        }

    /* By default, all receivers are considered as  */
    /* potential destinations.                      */
    FRET (OPC_TRUE)
    }
```

Default Transmission Delay Model (dra_txdel)

This section describes the functionality and implementation of the default model occupying stage one of the Radio Transceiver Pipeline. This model is named dra_txdel. The source code is provided in the file `dra_txdel.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The Transmission Delay Model is invoked by the Simulation Kernel immediately upon the start of a packet transmission. Its purpose is to compute the time required for the radio transmitter of interest to completely process and transmit the packet. This is the simulated time interval separating the beginning of transmission of the first bit and the end of transmission of the last bit of the packet.

The result of the Transmission Delay Model invocation for each packet is a single double precision floating point value which represents the transmission delay for the packet. The Simulation Kernel expects that this result will be placed in the packet's OPC_TDA_RA_TX_DELAY transmission data attribute.

Transmission Delay Computation

For all transmitted packets, the transmission delay is computed based on the channel data rate and the length of the packet. dra_txdel first obtains the data rate of the channel which is available in the OPC_TDA_RA_TX_DRATE transmission data attribute of the packet. The packet's length in bits is obtained and divided by the data rate to obtain transmission delay. This value is stored in the OPC_TDA_RA_TX_DELAY attribute of the packet as the final result of the procedure. The source code for dra_txdel is given in the following listing.

Figure 10-20   Source Code for dra_txdel Model

```
/* dra_txdel.ps.c */
/* Default transmission delay model for radio link Transceiver Pipeline */

/****************************************/
/*       Copyright (c) 1993-2002       */
/*      by OPNET Technologies, Inc.     */
/*      (A Delaware Corporation)        */
/*  7255 Woodmont Av., Suite 250        */
/*     Bethesda, MD 20814, U.S.A.       */
/*         All Rights Reserved.         */
/***************************************/

#include "opnet.h"


#if defined (__cplusplus)
extern "C"
#endif
void
dra_txdel_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    OpT_Packet_Sizepklen;
    double      tx_drate, tx_delay;

    /** Compute the transmission delay associated with the   **/
    /** transmission of a packet over a radio link.          **/
    FIN_MT (dra_txdel (pkptr));

    /* Obtain the transmission rate of that channel. */
    tx_drate = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_DRATE);

    /* Obtain length of packet. */
    pklen = op_pk_total_size_get (pkptr);

    /* Compute time required to complete transmission of packet. */
    tx_delay = pklen / tx_drate;

    /* Place transmission delay result in packet's */
    /* reserved transmission data attribute.       */
    op_td_set_dbl (pkptr, OPC_TDA_RA_TX_DELAY, tx_delay);

    FOUT
    }
```

Default Link Closure Model (dra_closure)

This section describes the functionality and implementation of the default model occupying stage two of the Radio Transceiver Pipeline. This model is named dra_closure. The source code is provided in the file `dra_closure.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The Radio Transceiver Pipeline determines the connectivity between a radio transmitter and a radio receiver by invoking the closure model immediately after completion of the transmission delay model. The simulation time at invocation is the start time for the packet transmission. The closure model allows dynamic enabling and disabling of radio links between radio transceivers. In other words, all pairs of transmitter channels and receiver channels that are evaluated positively by the receiver group model will be used in separate invocations of the closure model. This allows the pipeline to account for effects such as transmission range limitations, or dynamic changes to the attributes of the transmitters and receivers.

The result of the closure model invocation for each packet is a Boolean (OPC_TRUE or OPC_FALSE) value which represents the ability of the radio transmitter to reach the radio receiver of interest at the time of transmission. The Simulation Kernel expects this result to be placed in the packet's OPC_TDA_RA_CLOSURE transmission data attribute. If the closure model returns a value of OPC_FALSE, indicating no connectivity between the transmitter and receiver, the Simulation Kernel will not execute the remainder of the pipeline stages for this particular packet transmission.

Terrain Modeling

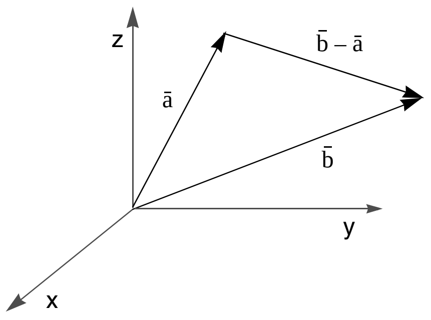
The default link closure model includes support for terrain modeling using the Terrain Modeling Module (TMM). With terrain modeling, closure is based on criteria set by the propagation model being used.

Closure Computation

When terrain modeling is not used, link closure is computed for all transmitted packets based on a *ray-tracing* line-of-sight model. The algorithm tests the line segment joining the transmitter and receiver for intersections with the earth's surface (the earth is modeled as a sphere). If an intersection exists, the receiver cannot be reached and the remainder of the pipeline stages will not be executed for the packet of interest. This algorithm does not account for any wave-bending.

The ray-tracing line-of-sight algorithm uses the transmitter vector, receiver vector and difference vector (from the transmitter to the receiver) in performing calculations. These vectors are defined within a coordinate system whose origin is at the earth's center. Vectors are denoted by ordered triples, $\bar{v} = (x, y, z)$, and can be represented as directed line segments beginning at an arbitrary point in space (x', y', z') and ending at a point in space $(x' + x, y' + y, z' + z)$. Vector addition can be performed by the pairwise addition of vector coordinates. The difference between two vectors $\bar{a} - \bar{b}$ is the vector which if added to $\bar{b}$ would result in $\bar{a}$. These relationships are depicted below for two vectors $\bar{a}$ and $\bar{b}$ represented at the origin.

Figure 10-21 Vector Subtraction Representation



The transmitter vector $\bar{T}$, is the vector with initial point at the origin and terminating at the point (t_x, t_y, t_z), the location of the transmitter. Similarly, the receiver vector $\bar{R}$ can be represented beginning at the origin and terminating at (r_x, r_y, r_z), the location of the receiver. The difference vector $\bar{D}$ from the transmitter to the receiver originates at the point (t_x, t_y, t_z), and terminates at the point (r_x, r_y, r_z). Thus its coordinates are $\bar{D} = (r_x - t_x, r_y - yt, r_z - t_x)$.

dra_closure uses a line-of-sight algorithm which tests the difference vector from the transmitter to the receiver for intersections with the earth's surface. The model makes use of the fact that an intersection will not occur, and therefore line-of-sight will exist between the transmitter and receiver in any of the three following cases:

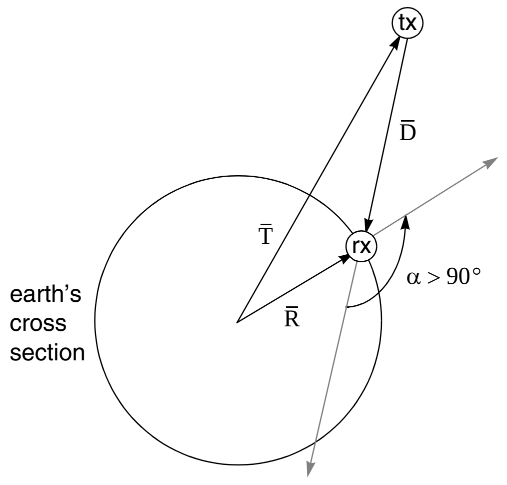
1) the angle between the receiver vector $\bar{R}$ and the difference vector $\bar{D}$ has a magnitude greater than 90 degrees.

2) the angle between the receiver vector $\bar{R}$, and difference vector $\bar{D}$, is less than or equal to 90 degrees *and* the angle between the transmitter vector $\bar{T}$ and $\bar{D}$ is less than 90 degrees.

3) the angle between the receiver vector $\bar{R}$ and difference vector $\bar{D}$ is less than or equal to 90 degrees, the angle between the transmitter vector $\bar{T}$, and $\bar{D}$ is greater than or equal to 90 degrees, *and* the shortest distance from the center of the earth to the line segment joining the transmitter and the receiver is greater than the radius of the earth.

Examples of each of these three cases are illustrated in Figure 10-22 and Figure 10-23. The locations of the transmitter and receiver are depicted by the encircled labels tx and rx, respectively.

Figure 10-22 Conditions for Link Closure - Case 1



Example 1:

The angle a between the difference vector and the receiver vector is greater than 90 degrees.

Figure 10-23 Conditions for Link Closure - Case 2

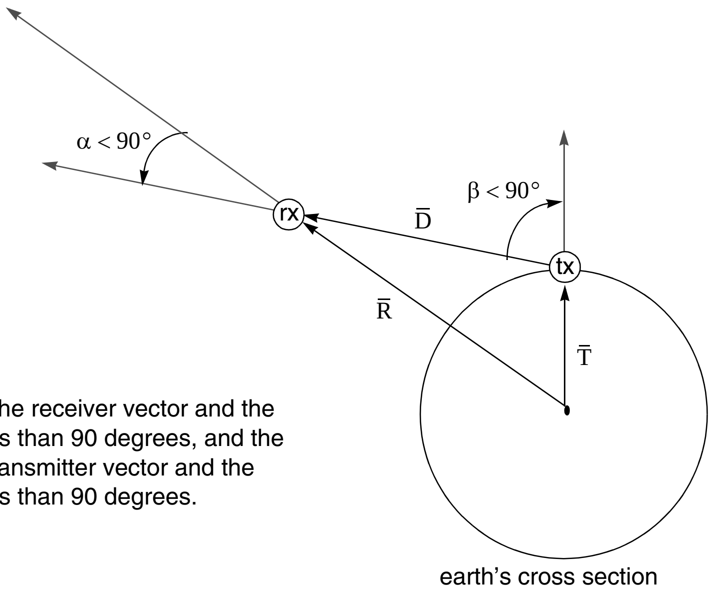
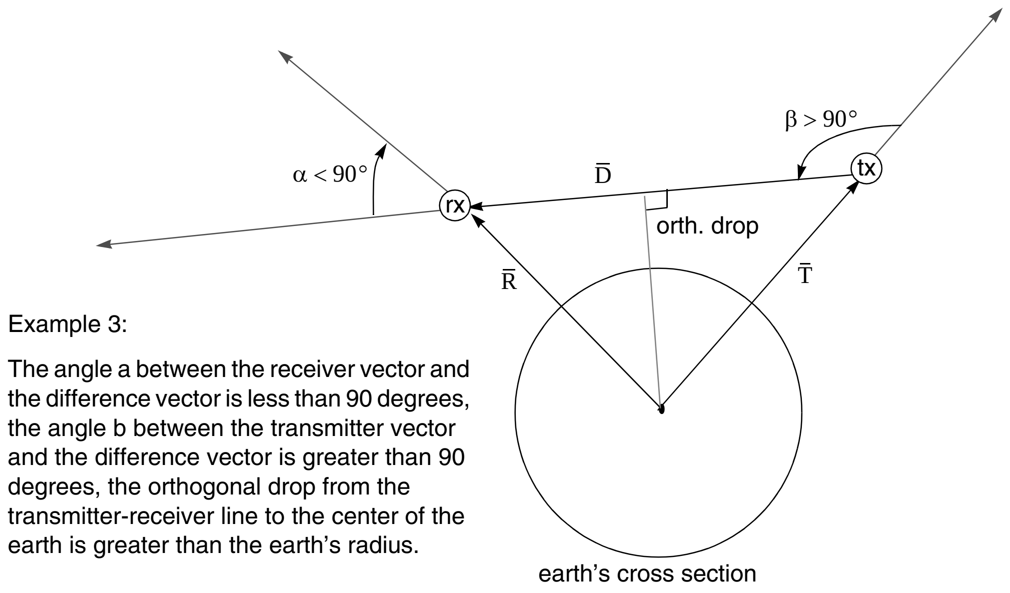

α < 90°
β < 90°
rx
tx
D̄
R̄
T̄
Example 2:
The angle a between the receiver vector and the difference vector is less than 90 degrees, and the angle b between the transmitter vector and the difference vector is less than 90 degrees.
earth's cross section

Figure 10-24 Conditions for Link Closure - Case 3


β > 90°
α < 90°
D̄
rx
tx
orth. drop
R̄
T̄
Example 3:
The angle a between the receiver vector and the difference vector is less than 90 degrees, the angle b between the transmitter vector and the difference vector is greater than 90 degrees, the orthogonal drop from the transmitter-receiver line to the center of the earth is greater than the earth's radius.
earth's cross section

The Closure Model computes the dot product of $\bar{R}$ and $\bar{D}$ to determine the size of the angle between the two vectors. The dot product can be given by:

$$a \bullet b = \|a\|\|b\|\cos\theta$$

Therefore,

- $\bar{a} \bullet \bar{b} = 0$ when cos θ = 0 (i.e., when θ = 90 or 270 degrees)

- $\bar{a} \bullet \bar{b} > 0$ when cos $\theta > 0$ (i.e., when $\theta < 90$ or $\theta > 270$ degrees), and
- $\bar{a} \bullet \bar{b} < 0$ when cos $\theta < 0$ (i.e., when $90 < \theta < 270$ degrees).

First, dra_closure obtains the cartesian-geocentric coordinates of the transmitter and receiver from the packet's OPC_TDA_RA_TX_GEO_X, OPC_TDA_RA_TX_GEO_Y, OPC_TDA_RA_TX_GEO_Z, OPC_TDA_RA_RX_GEO_X, OPC_TDA_RA_RX_GEO_Y, and OPC_TDA_RA_RX_GEO_Z transmission data attributes. These values are held in the automatic variables tx_x, tx_y, tx_z and rx_x, rx_y, rx_z. These variables represent the components of the transmitter vector and receiver vector. dra_closure uses these vectors to calculate the difference vector from the transmitter to the receiver, and assigns the difference vector components to the automatic variables dif_x, dif_y, and dif_z. The source code for these operations is shown below.

Figure 10-25 Source Code for dra_closure Model (Part 1)

```
/* dra_closure.ps.c                                                    */
/* TMM enabled closure model. This stage determines if communication is   */
/* possiblebetween sites. When executing during a TMM simulation, this    */
/* stage will utilize a TMM propagation model to determine closure and    */
/* calculate a path loss value.                                           */
/* The path loss value will be used later on in "power" pipeline stage.   */

/****************************************/
/*    Copyright (c) 1993-2002          */
/*      by OPNET Technologies, Inc.     */
/*      (A Delaware Corporation)        */
/*  7255 Woodmont Av., Suite 250        */
/*     Bethesda, MD 20814, U.S.A.       */
/*          All Rights Reserved.        */
/****************************************/

#include <math.h>
#include <string.h>
#include "opnet.h"

/* This stage will operate in three basic modes:                      */
/*      mode #1 The tranmission path will never be occluded. In this   */
/*              mode OPC_TDA_RA_CLOSURE will always be set to OPC_TRUE.*/
/*              This is the default mode for Wireless LAN model when   */
/*              TMM is not enabled.                                    */
/*      mode #2 basic free-space propagation closure is Line-of-Sight  */
/*              with a spherical earth model.                          */
/*              for alltransmissions.                                  */
/*      mode #3 Utilize a TMM propagation model.                       */

/****** enum types definition.******/
typedef enum DraT_Closure_Method
    {
    DraC_Line_Of_Sight_Never_Occluded,
    DraC_Earth_Line_Of_Sight,
    DraC_Terrain_Modeling
    } DraT_Closure_Method;

/***** Global variables*****/
/* Variables are initialized once and then used throughout the        */
/* simulation duration.                                               */

/* This variable will contain the closure method used during the whole */
/* simulation. Its possible values correspond to those defined under   */
/* DraT_Closure_Method type. It is initialized to Earth_Line_Of_Sight, */
/* however its final value will be set by tmm_closure_init().          */
static DraT_Closure_MethodDraS_Active_Closure_Method = DraC_Earth_Line_Of_Sight;
```

```
/* When we are in mode #3, this is the TMM propagation model used for  */
/* calculating path loss.                                               */
static TmmT_Propagation_Model *DraS_Closure_Prop_Model_Ptr = OPC_NIL;

/* When in mode #3, this variable controls whether detailed diagnostic */
/* message are logged in the simulation log for every packet           */
/* transmission.                                                        */
static int                  DraS_Closure_Tmm_Verbose_Mode;

/* When in mode #3, this variable is the name of the propagation model */
/* being used.                                                          */
static const char * DraS_Closure_Tmm_Prop_Model_Name;

/* Messages detailing packet transmission are logged when the          */
/* tmm_verbose mode is requested. (either by the environment attribute,*/
/* or via a call to tmm_verbose_set (1)).                               */
Log_Handle                  DraS_TMM_Verbose_Log_H;

/***** Function prototypes.*****/
static void tmm_closure_init (OP_SIM_CONTEXT_ARG_OPT);
static void dra_non_tmm_closure_method_set (OP_SIM_CONTEXT_ARG_OPT);
static void simple_earth_LOS_closure (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr);
static void tmm_model_closure_calc (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr);

/***** constants *****/
/* Threshold value in dB. Path loss beyond this threshold will be      */
/* considered to be so weak as to not have closure with the receiver.  */
#define LOSS_CUTOFF_THRESHOLD_DB-140.0

/* Name of the global attribute used by the Wireless LAN model to      */
/* select between the two non-TMM closure methods.This global          */
/* attribute is defined at wlan_mac.pr.m.                               */
#define CLOSURE_METHOD_GLOBAL_ATTRIBUTE_NAME"Closure Method (non-TMM)"

/***** pipeline procedure *****/

#if defined (__cplusplus)
extern "C"
#endif
void
dra_closure_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    static int  closure_initialized = 0;

    FIN_MT (dra_closure (pkptr));

    /* When using a parallel kernel, several threads of computation     */
    /* might reach that point at the same time. All we need is to       */
    /* ensure that tmm_closure_init is invoked only once.               */
    /* This simple requirement gives us some flexibility.               */
    /* closure_initialized might be set to 1 while another thread       */
    /* is reading the value.  That other thread will get some value     */
    /* (0, 1, or some garbled bit pattern).                             */
    /* If 1 or the garbled pattern is read, then the test will fail     */
    /* but since this results from the initialization already done      */
    /* then it is fine.                                                 */
    /* If 0 is read, then we try to create a mutex and lock it.         */
    /* There might be several calls to op_prg_mt_mutex_create, but      */
    /* all of them will return the same mutex pointer.  Writing the     */
    /* same value in a global variable should survive the               */
    /* non-atomicity.                                                   */
    if (!closure_initialized)
        {
        op_prg_mt_global_lock ();

        /* Check the variable again.  If another thread also "created"  */
        /* the mutex and already did the initialization, then the value */
        /* is going to be 1 and the test will fail.                     */
        if (!closure_initialized)
            {
            /* This function will determine the behavior for closure    */
            /* by setting static varaibles                              */
            /* (DraS_Active_Closure_Method and                          */
            /* DraS_Closure_Prop_Model_Ptr).                            */
```

```
                tmm_closure_init (OP_SIM_CONTEXT_PTR_OPT);

                /* Only once the intialization has been done do we reset    */
                /* the flag to make sure another thread is not going to     */
                /* rush ahead and assume the initialization is done.        */
                /* In the worst case, every thread will have "created" the  */
                /* same mutex and will be serialized on the mutex lock.     */
                closure_initialized = 1;
                }

            op_prg_mt_global_unlock ();
            }

        /* This stage has three modes of operation. (See the comments      */
        /* above, near the definition of DraT_Closure_Method type).        */
        switch (DraS_Active_Closure_Method)
            {
            case DraC_Line_Of_Sight_Never_Occluded:
                /* mode #1 The tranmission path will never be occluded.    */
                /* Set OPC_TDA_RA_CLOSURE to OPC_TRUE for all              */
                /* transmissions.                                          */
                op_td_set_int (pkptr, OPC_TDA_RA_CLOSURE, OPC_TRUE);
            break;
            case DraC_Earth_Line_Of_Sight:
                /* mode #2: basic spherical earth closure.                 */
                simple_earth_LOS_closure (OP_SIM_CONTEXT_PTR_OPT_COMMA pkptr);
            break;
            case DraC_Terrain_Modeling:
                /* mode #3: Utilize TMM propagation model.                 */
                tmm_model_closure_calc (OP_SIM_CONTEXT_PTR_OPT_COMMA pkptr);
            break;
            }
        FOUT
        }

    /***** Support functions *****/

    static void
    tmm_closure_init (OP_SIM_CONTEXT_ARG_OPT)
        {
        int             using_tmm;
        Log_Handle      tmm_problem_log_handle;
        int             load_successful;
        char            line0_buf [512];
        char            line1_buf [512];

        /** This function is invoked once at the start of the simulation.      **/
        /** Options that will be used throughout the simulation duration are   **/
        /** established in this function.                                      **/
        FIN_MT (tmm_closure_init ());

        /* Define a log handler for all messages related to the                */
        /* initialization.                                                     */
        tmm_problem_log_handle = op_prg_log_handle_create (
            OpC_Log_Category_Configuration,
            "TMM", "closure stage loading of propagation model", 20);

        /* If the nodes are out of valid computation areas                     */
        /* such as for Longley-Rice, a very large number of log                */
        /* entries could be generated.  Stop recording after 500               */
        DraS_TMM_Verbose_Log_H = op_prg_log_handle_create (
            OpC_Log_Category_Lowlevel,
            "TMM", "path loss calculation", 500);

        /* Determine if we are executing as a simulation that wants to use TMM */
        if (prg_env_attr_value_get (PrgC_Env_Attr_Boolean, TMMC_ENV_SIMULATE,
                &using_tmm) == PrgC_Compcode_Failure)
            using_tmm = OPC_FALSE;

        /* Check is TMM is active.                                             */
        if (using_tmm == OPC_FALSE)
            {
            /* Simulation is not using TMM for path loss calculations or       */
            /* closure. Set the non-TMM closure method that will be used.      */
            dra_non_tmm_closure_method_set (OP_SIM_CONTEXT_PTR_OPT);
            }
        else
            {
            /** Simulation is using TMM for propagation. **/

            /* Initialize a flag to check for a successful load of the TMM     */
            /* module.                                                         */
            load_successful = OPC_FALSE;

            /* Attempt to load the default TMM propagation model.              */
            DraS_Closure_Tmm_Prop_Model_Name = tmm_default_propagation_model_get ();
            if (strcmp ("NONE", DraS_Closure_Tmm_Prop_Model_Name) == 0)
                {

                /* Propagation model called NONE, is special and indicates     */
                /* that no default propagation model is set. Use free space.   */
                load_successful = OPC_FALSE;

                /* Simulation is not using TMM for path loss calculations      */
                /* or closure. Set the non-TMM closure method that will be     */
                /* used.                                                       */
                dra_non_tmm_closure_method_set (OP_SIM_CONTEXT_PTR_OPT);

                FOUT
                }

            /* Read the TMM propagation model.                                 */
            DraS_Closure_Prop_Model_Ptr = tmm_propagation_model_get
                (DraS_Closure_Tmm_Prop_Model_Name);

            /* Check if the TMM propagation model was not successfully read.   */
            if (DraS_Closure_Prop_Model_Ptr == OPC_NIL)
                {
                /* Failed to load the requested model. Most likely because     */
                /* is missing model from mod_dirs.                             */
                load_successful = OPC_FALSE;

                /* Prepare a simulation message.                               */
                sprintf (line0_buf, "TMM: unable to load propagation model (%s). ",
                    DraS_Closure_Tmm_Prop_Model_Name);
                strcpy (line1_buf, "Using default closure instead.");

                /* Print simulation message.                                   */
                op_sim_message (line0_buf, line1_buf);

                /* Write a log message.                                        */
                op_prg_log_entry_write (tmm_problem_log_handle,
                    "Pipeline stage 'closure' during initialization for\n"
                    "%s\n"
                    "%s\n"
                    "\n"
                    "Check that your mod_dirs contains all 3 of the needed\n"
                    "propagation model files (.prop.d, .prop.p and .prop.so).\n",
                    line0_buf,
                    line1_buf);
                }
            else if (DraS_Closure_Prop_Model_Ptr->initialized_ok_flag == OPC_FALSE)
                {
                /* Propagation model's init function has set the flag          */
                /* "initialized_ok_flag" to indicate some problem inside       */
                /* of the propagation model.                                   */
                load_successful = OPC_FALSE;

                /* Prepare a simulation message.                               */
                sprintf (line0_buf,
                    "TMM propagation model (%s) reported an initialization problem.",
                    DraS_Closure_Tmm_Prop_Model_Name);
                strcpy (line1_buf, "Using default closure instead.");

                /* Print simulation message.                                   */
                op_sim_message (line0_buf, line1_buf);
```

```
                /* Write a log message.                                     */
                op_prg_log_entry_write (tmm_problem_log_handle,
                    "Pipeline stage 'closure' during initialization for TMM:\n"
                    "%s\n"
                    "%s\n",
                    line0_buf,
                    line1_buf);
                }
            else
                {
                /* We've successfully obtained the propagation model. We       */
                /* are going to use the path loss function from the model      */
                /* we have loaded.                                             */
                load_successful = OPC_TRUE;

                /* Indicate that the TMM closure method will be used.          */
                DraS_Active_Closure_Method = DraC_Terrain_Modeling;

                /* If verbose, write a log message.                            */
                if (tmm_verbose_get ())
                    {
                    sprintf (line0_buf,
                        "TMM initialization: successfully loaded the propagation model
                            (%s)",
                        DraS_Closure_Tmm_Prop_Model_Name);
                    op_prg_log_entry_write (tmm_problem_log_handle,
                        line0_buf);
                    }
                }

            /* Check if the TMM module was not successfully loaded.            */
            if (load_successful == OPC_FALSE)
                {
                /* Simulation is not using TMM for path loss calculations      */
                /* or closure. Set the non-TMM closure method that will be     */
                /* used.                                                       */
                dra_non_tmm_closure_method_set (OP_SIM_CONTEXT_PTR_OPT);
                }
            }

        FOUT
        }

    static void
    dra_non_tmm_closure_method_set (OP_SIM_CONTEXT_ARG_OPT)
        {
        int             attr_value;

        /** This function is called to get the closure method that the         **/
        /** simulation will use if TMM is not active. There are two more        **/
        /** available closure methods (besides TMM) that may be used:           **/
        /** -The transmission path will never be occluded.                      **/
        /**     -Line-of-sight with a spherical earth model                     **/
        /**                                                                     **/
        /** There are two cases on how the non-TMM closure method will be       **/
        /** selected:                                                           **/
        /**                                                                     **/
        /** A) For Wireless LAN standard models: The closure method is          **/
        /** selected based on the value set on the global attribute:            **/
        /**   CLOSURE_METHOD_GLOBAL_ATTRIBUTE_NAME                              **/
        /**                                                                     **/
        /** B) Any other custom wireless models: If the global                  **/
        /** attribute is not defined in the simulation, then the Earth          **/
        /** Line-of-Sight method will always be selected.                       **/

        FIN_MT(dra_non_tmm_closure_method_set (void));

        /* Check if the global attribute exists. If it does, apply the case    */
        /* for Wireless LAN (A), otherwise this indicates that custom          */
        /* wireless models are being used (apply case B).                      */
        if (op_ima_sim_attr_exists (CLOSURE_METHOD_GLOBAL_ATTRIBUTE_NAME))
            {
            if (op_ima_sim_attr_get_int32 (CLOSURE_METHOD_GLOBAL_ATTRIBUTE_NAME,
    &attr_value) == OPC_COMPCODE_SUCCESS)
                {
                /* Set the closure method based on the global attribute.       */
                DraS_Active_Closure_Method = (DraT_Closure_Method) attr_value;
                }
            else
                {
                /* For non-Wireless LAN models always use Earth Line-of-Sight. */
                DraS_Active_Closure_Method = DraC_Line_Of_Sight_Never_Occluded;
                }
            }
        else
            {
            /* For non-Wireless LAN models always use Earth Line-of-Sight.     */
            DraS_Active_Closure_Method = DraC_Earth_Line_Of_Sight;
            }
        FOUT;
        }

    static void
    tmm_model_closure_calc (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
        {
        TmmT_Positiontx_position;
        TmmT_Positionrx_position;
        void *      pipeline_invocation_state_ptr;
        double      tx_base_freq;
        double      tx_bandwidth;
        double      tx_center_freq;
        int         verbose_active;
        int         trace_active;
        int         str_index;
        char *      msg_str_ptrs [TMMC_LOSS_MESSAGE_BUF_NUM_STRS];
        char        log_str_buf [16 * TMMC_LOSS_MESSAGE_BUF_STR_SIZE];

        double      tmm_model_path_loss_dB;
        TmmT_Loss_Statustmm_model_loss_status;
        char        tmm_model_msg_buffer_v [TMMC_LOSS_MESSAGE_BUF_NUM_STRS]
    [TMMC_LOSS_MESSAGE_BUF_STR_SIZE];
        char        msg_buf0 [256];
        char        msg_buf1 [256];
        char        msg_buf2 [256];
        char        msg_buf3 [256];
        char        msg_buf4 [256];

        /** Call the propagation model's path loss calculation method.         **/
        /** The function can report:                                           **/
        /**     a signal loss value                                            **/
        /**     a error condition (e.g. invalid elevation)                     **/
        /**     a lack of link closure - so no communication is                **/
        /**     possible.                                                      **/
        FIN_MT (tmm_model_closure_calc (pkptr));


    /* Get transmission frequency in Hz. */
        tx_base_freq = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_FREQ);
        tx_bandwidth = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_BW);
        tx_center_freq = tx_base_freq + (tx_bandwidth / 2.0);

        /* Get transmitter's location. */
        tx_position.latitude = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_LAT);
        tx_position.longitude = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_LONG);
        tx_position.elevation = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_ALT);

        /* Get receiver's location. */
        rx_position.latitude = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_LAT);
        rx_position.longitude = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_LONG);
        rx_position.elevation = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_ALT);

        /* Propagation models shipped by OPNET don't use any extra state       */
        /* so just for clarity set the local state pointer to null.            */
        /* User created propagation models can pass any value they wish.       */
        pipeline_invocation_state_ptr = OPC_NIL;

        /* Determine if TMM verbose is requested.                              */
        /* This can be set using the tmm_verbose environment attribute        */
        trace_active = op_prg_odb_trace_active ()
            || op_prg_odb_pktrace_active (pkptr);
        verbose_active = tmm_verbose_get ();

        /*** Call the propagation model's path_loss_calc_method.              ***/

        /* All TMM propagation models have a path loss function prototype      */
        /* like this (from tmm.h):                                             */
        /*
            double TmmT_Path_Loss_Calc_Method (
                    TmmT_Propagation_Model *model_ptr,
                    void *                  tda_state_ptr,
                    TmmT_Position *         tx_position_ptr,
                    TmmT_Position *         rx_position_ptr,
                    double                  center_frequency,
                    double                  bandwidth,
                    int                     verbose,
                    TmmT_Loss_Status *      status_ptr,
                    char                    message_buffer_v
                                                [TMMC_LOSS_MESSAGE_BUF_NUM_STRS]
                                                [TMMC_LOSS_MESSAGE_BUF_STR_SIZE]);
        */
        for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS; str_index++)
            tmm_model_msg_buffer_v [str_index] [0] = '\0';
        tmm_model_path_loss_dB =
            DraS_Closure_Prop_Model_Ptr->path_loss_calc_method (
                DraS_Closure_Prop_Model_Ptr,
                pipeline_invocation_state_ptr,
                &tx_position,
                &rx_position,
                tx_center_freq,
                tx_bandwidth,
                verbose_active || trace_active,
                &tmm_model_loss_status,
                tmm_model_msg_buffer_v );

        if (tmm_model_loss_status == TmmC_Loss_No_Closure)
            {
            /* The propagation model has reported that signal                  */
            /* from this packet will not be observable by the                  */
            /* receiver. This packet fails closure.                            */

            if (trace_active || verbose_active)
                {
                sprintf (msg_buf0, "Transmission Closure: Propagation Model %s:",
                    DraS_Closure_Tmm_Prop_Model_Name );
                sprintf (msg_buf1,
                    "Path loss between transmitter (ID %d) and receiver (ID %d)",
                    op_td_get_int (pkptr, OPC_TDA_RA_TX_OBJID),
                    op_td_get_int (pkptr, OPC_TDA_RA_RX_OBJID));
                strcpy (msg_buf2,
                    "fails closure. Packet can not be received by the receiver");
                strcpy (msg_buf3,
                    "Additional messages reported by propagation model follow:");
                for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS;
                    str_index++)
                    {
                    if (tmm_model_msg_buffer_v [str_index] [0] == '\0')
                        {
                        /* A null pointer to op_prg_odb_print will signify end of*/
                        /* arguments                                             */
                        msg_str_ptrs [str_index] = OPC_NIL;
                        break;
                        }
                    else
                        {
                        msg_str_ptrs [str_index] = &tmm_model_msg_buffer_v [str_index]
                            [0];
                        }
                    }
                if (msg_str_ptrs [0] == OPC_NIL)
                    {
```

```
            strcpy (msg_buf3, "<no messages reported by the propagation model>");
            }
        }
    if (trace_active)
        {
        /* Print message to ODB */
        op_prg_odb_print_major (msg_buf0, msg_buf1, msg_buf2, msg_buf3,
            msg_str_ptrs [0], msg_str_ptrs [1], msg_str_ptrs [2],
            msg_str_ptrs [3], msg_str_ptrs [4], OPC_NIL);
        }
    if (verbose_active)
        {
        /* Log a simulation-log message becasue tmm_verbose is true. */

        /* Entries in the log are added via 1 long format string */
        log_str_buf [0] = '\0';
        strcat (log_str_buf, msg_buf0);
        strcat (log_str_buf, "\n");

        strcat (log_str_buf, "  ");
        strcat (log_str_buf, msg_buf1);
        strcat (log_str_buf, "\n");

        strcat (log_str_buf, "  ");
        strcat (log_str_buf, msg_buf2);
        strcat (log_str_buf, "\n");

        strcat (log_str_buf, "  ");
        strcat (log_str_buf, msg_buf3);
        strcat (log_str_buf, "\n");

        for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS;
            str_index++)
            {
            if (tmm_model_msg_buffer_v [str_index] [0] != '\0')
                {
                strcat (log_str_buf, "    ");
                strcat (log_str_buf, tmm_model_msg_buffer_v [str_index]);
                strcat (log_str_buf, "\n");
                }
            else
                {
                /* Empty string, so no more string in message buffer from  */
                /* propagation model                                       */
                break;
                }
            }

        /* Log the message */
        op_prg_log_entry_write (DraS_TMM_Verbose_Log_H,
            log_str_buf);
        }
    op_td_set_int (pkptr, OPC_TDA_RA_CLOSURE, OPC_FALSE);
    }
else if (tmm_model_loss_status == TmmC_Loss_Error)
    {
    /* An error condition was reported by the propagation model.            */
    /* We will fall back to simple freespace closure.                       */

    sprintf (msg_buf0, "Transmission Closure: Propagation Model %s:",
        DraS_Closure_Tmm_Prop_Model_Name );
    sprintf (msg_buf1, "Model reported error in computing path between
                        transmitter (ID %d) and receiver (ID %d)",
        op_td_get_int (pkptr, OPC_TDA_RA_TX_OBJID),
        op_td_get_int (pkptr, OPC_TDA_RA_RX_OBJID));
    strcpy (msg_buf2, "Using simple-earth closure model for this transmission");
    strcpy (msg_buf3, "Messages reported by propagation model follow:");
    for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS; str_index++)
        {
        if (tmm_model_msg_buffer_v [str_index] [0] == '\0')
            {
            /* A null pointer to op_prg_odb_print will signify */
            /* end of arguments                                */
            msg_str_ptrs [str_index] = OPC_NIL;
            break;
            }
        else
            {
            msg_str_ptrs [str_index] = &tmm_model_msg_buffer_v [str_index] [0];
            }
        }
    if (msg_str_ptrs [0] == OPC_NIL)
        {
        strcpy (msg_buf3, "<no messages reported by the propagation model>");
        }
    if (trace_active)
        {
        /* Print message to ODB */
        op_prg_odb_print_major (msg_buf0, msg_buf1, msg_buf2, msg_buf3,
            msg_str_ptrs [0], msg_str_ptrs [1], msg_str_ptrs [2],
            msg_str_ptrs [3], msg_str_ptrs [4], OPC_NIL);
        }

    /* Entries in the log are added via 1 long format string*/
    log_str_buf [0] = '\0';
    strcat (log_str_buf, msg_buf0);
    strcat (log_str_buf, "\n");

    strcat (log_str_buf, "  ");
    strcat (log_str_buf, msg_buf1);
    strcat (log_str_buf, "\n");

    strcat (log_str_buf, "  ");
    strcat (log_str_buf, msg_buf2);
    strcat (log_str_buf, "\n");

    strcat (log_str_buf, "  ");
    strcat (log_str_buf, msg_buf3);
    strcat (log_str_buf, "\n");

    for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS; str_index++)
        {
        if (tmm_model_msg_buffer_v [str_index] [0] != '\0')
            {
            strcat (log_str_buf, "    ");
            strcat (log_str_buf, tmm_model_msg_buffer_v [str_index]);
            strcat (log_str_buf, "\n");
            }
        else
            {
            /* Empty string, so no more string in message buffer   */
            /* from propagation model                              */
            break;
            }
        }

    /* Log message indicating we are reverting to simple freespace model   */
    /* for this packet transmission.                                       */
    op_prg_log_entry_write (DraS_TMM_Verbose_Log_H,
        log_str_buf);

    /* Fallback to the simple earth line-of-sight closure model           */
    simple_earth_LOS_closure (OP_SIM_CONTEXT_PTR_OPT_COMMA pkptr);
    }
else
    {
    /* The propagation model was able to calculate a signal value.        */

    int message_from_prop = OPC_FALSE;
    if (tmm_model_path_loss_dB > LOSS_CUTOFF_THRESHOLD_DB)
        {
        /* This packet's signal is strong enough. The amount of loss is less*/
        /* then our cutoff threshold (defined by LOSS_CUTOFF_THRESHOLD_DB). */

        /* The path loss is within our threshold for reception.            */
        /* We have link closure.                                           */
        op_td_set_int (pkptr, OPC_TDA_RA_CLOSURE, OPC_TRUE);
        /* The power pipeline stage will calculate the final received  */
        /* signal power (included antenna, tx power, etc.).            */
        /* For now, we store just the propagation loss in the TD for   */
        /* received power.                                             */

        /* The default power pipeline stage will use a convention of   */
        /* interpreting this path loss in raw form (not in dB).        */
        op_td_set_dbl (pkptr, OPC_TDA_RA_RCVD_POWER,
            pow (10.0, tmm_model_path_loss_dB / 10.0));

        /* Write an ODB trace message if enabled.                      */
        if (trace_active)
            {
            op_prg_odb_print_major
                ("Successful path loss computation reported by TMM.", OPC_NIL);

            /* Also append the additional messages from the propagation*/
            /* model if any.                                           */
            for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS;
                str_index++)
                {
                if (tmm_model_msg_buffer_v [str_index][0] != '\0')
                    {
                    op_prg_odb_print_minor (tmm_model_msg_buffer_v [str_index],
                        OPC_NIL);
                    }
                else
                    {
                    /* Empty string indicates end of messages.         */
                    /* Terminate the loop.                             */
                    break;
                    }
                }
            }
        }
    else
        {
        /* The propagation loss is so great, that the received signal  */
        /* is below our threshold. We choose to regard this packet as  */
        /* failing closure.                                            */
        /* See the constant value define for this threshold at the top */
        /* of the file. By using this threshold, we can greatly improve*/
        /* simulation performance by scheduling fewer pipeline events. */

        if (trace_active || verbose_active)
            {
            sprintf (msg_buf0, "Transmission Closure: Propagation Model %s:",
                DraS_Closure_Tmm_Prop_Model_Name );
            sprintf (msg_buf1,
                "Path loss between transmitter (ID %d) and receiver (ID %d)",
                op_td_get_int (pkptr, OPC_TDA_RA_TX_OBJID),
                op_td_get_int (pkptr, OPC_TDA_RA_RX_OBJID));
            sprintf (msg_buf2, "is beyond threshold of %f dB.",
                LOSS_CUTOFF_THRESHOLD_DB);
            strcpy (msg_buf3, "Transmission is considered to fail closure");
            strcpy (msg_buf4,
                "Additional messages reported by propagation model follow:");
            for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS;
                str_index++)
                {
                if (tmm_model_msg_buffer_v [str_index] [0] == '\0')
                    {
                    /* A null pointer to op_prg_odb_print_will signify */
                    /* end of arguments                                */
                    msg_str_ptrs [str_index] = OPC_NIL;
                    break;
                    }
                else
                    {
                    msg_str_ptrs [str_index] = & tmm_model_msg_buffer_v
                        [str_index] [0];
                    }
                }
                if (msg_str_ptrs [0] == OPC_NIL)
                    {
                    strcpy (msg_buf4,
                        "<no messages reported by the propagation model>");
                    }
                }

            if (trace_active)
                {
                /* Print message to ODB */
                op_prg_odb_print_major (msg_buf0, msg_buf1, msg_buf2, msg_buf3,
                    msg_buf4, msg_str_ptrs [0], msg_str_ptrs [1], msg_str_ptrs [2],
                    msg_str_ptrs [3], msg_str_ptrs [4], OPC_NIL);
                }
            if (verbose_active)
                {
                /* Log a simulation-log message because tmm_verbose is true.  */

                /* Entries in the log are added via 1 long format string      */
                log_str_buf [0] = '\0';
                strcat (log_str_buf, msg_buf0);
                strcat (log_str_buf, "\n");

                strcat (log_str_buf, "  ");
                strcat (log_str_buf, msg_buf1);
                strcat (log_str_buf, "\n");

                strcat (log_str_buf, "  ");
                strcat (log_str_buf, msg_buf2);
                strcat (log_str_buf, "\n");

                strcat (log_str_buf, "  ");
                strcat (log_str_buf, msg_buf3);
                strcat (log_str_buf, "\n");

                strcat (log_str_buf, "  ");
                strcat (log_str_buf, msg_buf4);
                strcat (log_str_buf, "\n");

                for (str_index = 0; str_index < TMMC_LOSS_MESSAGE_BUF_NUM_STRS;
                    str_index++)
                    {
                    if (tmm_model_msg_buffer_v [str_index] [0] != '\0')
                        {
                        strcat (log_str_buf, "    ");
                        strcat (log_str_buf, tmm_model_msg_buffer_v [str_index]);
                        strcat (log_str_buf, "\n");
                        }
                    else
                        {
                        /* Empty string, so no more string in message buffer   */
                        /* from propagation model                              */
                        break;
                        }
                    }

                /* Log the message */
                op_prg_log_entry_write (DraS_TMM_Verbose_Log_H,
                    log_str_buf);
                }
            op_td_set_int (pkptr, OPC_TDA_RA_CLOSURE, OPC_FALSE);
            }
        }

    FOUT
    }

static void
simple_earth_LOS_closure (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    int     occlude;
    double  tx_x, tx_y, tx_z, rx_x, rx_y, rx_z;
    double  dif_x, dif_y, dif_z, dot_rx_dif;
    double  dot_tx_dif, rx_mag, dif_mag, cos_rx_dif, sin_rx_dif;
    double  orth_drop;
    double  tx_alt, rx_alt;

    /** Compute whether or not the packet's transmitter                **/
    /** can reach the receiver of interest.                            **/

    /** Determinate for closure is direct line of sight                **/
    /** between transmitterand receiver. The earth is                  **/
    /** modeled as simple sphere.**/
    FIN_MT (simple_earth_LOS_closure (pkptr));

    /* dra_closure implements a 'ray-tracing' closure model for radio  */
    /* transmissions, testing the line segment joining tx and rx for   */
    /* intersection with the earth (modeled as a sphere).  This default*/
    /* model does not account for any wave bending.                    */

    /* Obtain the cartesian-geocentric coordinates for both nodes.     */
    tx_x = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GEO_X);
    tx_y = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GEO_Y);
    tx_z = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GEO_Z);

    rx_x = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GEO_X);
    rx_y = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GEO_Y);
    rx_z = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GEO_Z);
```

If the dot product of $\bar{D}$ and $\bar{R}$ is less than or equal to 0 (i.e., the angle between the receiver vector and the difference vector is greater than 90 degrees), there is line of sight between the transmitter and the receiver. In this case, the automatic variable occlude is set to OPC_FALSE to indicate that there is no occlusion between the receiver and transmitter, as shown below.

Figure 10-26 Source Code for dra_closure Model (Part 2)

```
/* Calculate difference vector (transmitter to receiver). */
dif_x = rx_x - tx_x;
dif_y = rx_y - tx_y;
dif_z = rx_z - tx_z;

/* Calculate dot product of (rx) and (dif) vectors. */
dot_rx_dif = rx_x*dif_x + rx_y*dif_y + rx_z*dif_z;

/* If angle (rx, dif) > 90 deg., there is no occlusion. */
if (dot_rx_dif <= 0.0)
    occlude = OPC_FALSE;
```

If the first test fails to determine line of sight (i.e., the angle between $\bar{R}$ vector and $\bar{D}$ is less than or equal to 90 degrees), dra_closure calculates the dot product of $\bar{T}$ and $\bar{D}$. If this dot product is greater than or equal to 0 (i.e., the angle between these vectors is less than 90 degrees), there is line of sight. The automatic variable occlude is set to OPC_FALSE to indicate that there is no occlusion between the transmitter and receiver.

Figure 10-27 Source Code for dra_closure Model (Part 3)

```
else
    {
    /* Calculate dot product of (tx) and (dif) vectors. */
    dot_tx_dif = tx_x*dif_x + tx_y*dif_y + tx_z*dif_z;
    /* If angle (tx, dif) < 90) there is no occlusion. */
    if (dot_tx_dif >= 0.0)
        {
        tx_alt = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_ALT);
        rx_alt = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_ALT);
        if ((tx_alt < 0) || (rx_alt < 0))
            {
            occlude = OPC_TRUE;
            }
        else
            {
            occlude = OPC_FALSE;
            }
        }
```

If both of these tests fail to determine line of sight, dra_closure calculates the length of the orthogonal drop from the transmitter-receiver line to the center of the earth. If the length of the orthogonal drop is less than the earth's radius, there is no line of sight and the automatic variable occlude is set to OPC_TRUE to indicate that there is occlusion between the transmitter and receiver. If instead the length of the orthogonal vector is greater than or equal to the earth's radius, there is line of sight and the automatic variable occlude is set to OPC_FALSE to indicate that there is no occlusion between the transmitter and receiver.

Figure 10-28 Source Code for dra_closure Model (Part 4)

```
        else
            {
            /* Calculate magnitude of (rx) and (dif) vectors. */
            rx_mag = sqrt (rx_x*rx_x + rx_y*rx_y + rx_z*rx_z);
            dif_mag = sqrt (dif_x*dif_x + dif_y*dif_y + dif_z*dif_z);

            /* Calculate sin (rx, dif). */
            cos_rx_dif = dot_rx_dif / (rx_mag * dif_mag);
            sin_rx_dif = sqrt (1.0 - (cos_rx_dif * cos_rx_dif));

            /* Calculate length of orthogonal drop */
            /* from (dif) to earth center.         */
            orth_drop = sin_rx_dif * rx_mag;

            /* The satellites are occluded if this distance is less than   */
            /* the earth's radius. Depending on the characteristics of your*/
            /* radio model, you may wish to scale this radius by 4/3 to    */
            /* account for atmospheric refraction.                         */
            if (orth_drop < VOSC_EARTH_RADIUS_METERS)
                occlude = OPC_TRUE;
            else
                occlude = OPC_FALSE;
            }
        }

    /* Place closure status in packet transmission data block. */
    op_td_set_int (pkptr, OPC_TDA_RA_CLOSURE,
        (occlude == OPC_FALSE) ? OPC_TRUE : OPC_FALSE);

    FOUT;
    }
```

Default Channel Match Model (dra_chanmatch)

This section describes the functionality and implementation of the default model occupying stage three of the Radio Transceiver Pipeline. This model is named dra_chanmatch. The source code is provided in `dra_chanmatch.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The Radio Transceiver Pipeline determines the compatibility between a radio transmitter channel and a radio receiver channel by invoking the channel match model immediately after completion of the closure model, provided that the latter determined that there was link closure. The simulation time at invocation is the start time for the packet transmission. This model is used to classify transmissions into three categories with respect to the receiver channel of interest. Packets can be considered as *valid* transmissions which the receiver is capable of processing, *interference* which cannot be decoded or used by the receiver but which affects other transmissions, or *ignored* in which case the pipeline sequence is not continued for the packet transmission.

The result of the channel match model for each packet is an integer flag indicating that the packet falls into one of the three categories. The three symbolic constants corresponding to these categories are OPC_TDA_RA_MATCH_VALID, OPC_TDA_RA_MATCH_NOISE, and OPC_TDA_RA_MATCH_IGNORE, respectively. The Simulation Kernel expects this result to be placed in the packet's OPC_TDA_RA_MATCH_STATUS transmission data attribute.

Compatibility Evaluation

The default pipeline model, dra_chanmatch, considers a transmitter channel and a receiver channel to be compatible only if all of their common channel characteristics match. The characteristics which are analyzed by dra_chanmatch include frequency, bandwidth, data rate, spreading code, and modulation. These are given by the channel attributes min freq, bandwidth, data rate, spreading code, and by the modulation attribute of the transmitter and receiver. If these attributes are identical between the transmitter and the receiver, the packet is considered to be *valid*.

If the band of transmission has no overlap with the band of the receiver channel, the packet is considered to be an invalid packet which is unable to affect the receiver in any significant manner. These packets are labeled as *ignored*. For such packets, the remainder of the pipeline stages need not be executed.

If the band of transmission overlaps the band of the receiver channel, but any of the channel attributes do not match, the packet is considered to be *interference.* Interference packets are modeled only while they might affect the proper reception of valid packets. Thus, certain stages later in the pipeline will not be invoked to process interference packets.

Figure 10-29   Source Code for dra_chanmatch Model

```
/* dra_chanmatch.ps.c */
/* Default channel match model for radio link Transceiver Pipeline */

/****************************************/
/*       Copyright (c) 1993-2002       */
/*      by OPNET Technologies, Inc.     */
/*      (A Delaware Corporation)        */
/*  7255 Woodmont Av., Suite 250        */
/*     Bethesda, MD 20814, U.S.A.       */
/*          All Rights Reserved.        */
/****************************************/

#include "opnet.h"


#if defined (__cplusplus)
extern "C"
#endif
void
dra_chanmatch_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double  tx_freq, tx_bw, tx_drate, tx_code;
    double  rx_freq, rx_bw, rx_drate, rx_code;
    Vartype tx_mod;
    Vartype rx_mod;

    /** Determine the compatibility between transmitter and receiver channels. **/
    FIN_MT (dra_chanmatch (pkptr));

    /* Obtain transmitting channel attributes. */
    tx_freq = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_FREQ);
    tx_bw   = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_BW);
    tx_drate= op_td_get_dbl (pkptr, OPC_TDA_RA_TX_DRATE);
    tx_code = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_CODE);
    tx_mod  = op_td_get_ptr (pkptr, OPC_TDA_RA_TX_MOD);

    /* Obtain receiving channel attributes. */
    rx_freq = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_FREQ);
    rx_bw   = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_BW);
    rx_drate= op_td_get_dbl (pkptr, OPC_TDA_RA_RX_DRATE);
    rx_code = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_CODE);
    rx_mod  = op_td_get_ptr (pkptr, OPC_TDA_RA_RX_MOD);

    /* For non-overlapping bands, the packet has no */
    /* effect; such packets are ignored entirely.   */
    if ((tx_freq > rx_freq + rx_bw) || (tx_freq + tx_bw < rx_freq))
        {
        op_td_set_int (pkptr, OPC_TDA_RA_MATCH_STATUS, OPC_TDA_RA_MATCH_IGNORE);
        FOUT
        }

    /* Otherwise check for channel attribute mismatches which would */
    /* cause the in-band packet to be considered as noise.          */
    if ((tx_freq != rx_freq) || (tx_bw != rx_bw) ||
        (tx_drate != rx_drate) || (tx_code != rx_code) || (tx_mod != rx_mod))
        {
        op_td_set_int (pkptr, OPC_TDA_RA_MATCH_STATUS, OPC_TDA_RA_MATCH_NOISE);
        FOUT
        }

    /* Otherwise the packet is considered a valid transmission which */
    /* could eventually be accepted at the error correction stage.   */
    op_td_set_int (pkptr, OPC_TDA_RA_MATCH_STATUS, OPC_TDA_RA_MATCH_VALID);

    FOUT
    }
```

Default Transmitter Antenna Gain Model (dra_tagain)

This section describes the functionality and implementation of the default model occupying stage four of the Radio Transceiver Pipeline. This model is named dra_tagain. The source code is provided in the file `dra_tagain.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The Radio Transceiver Pipeline determines the gain associated with a packet's transmitting antenna by invoking the transmitter antenna gain model immediately after completion of the channel match model, provided that the packet was not determined to be *ignored*. The simulation time at invocation is the start time for the packet transmission.

The result of the transmitter antenna gain model for each packet is a floating point value specifying the gain of the transmitter's antenna in the direction of the receiver of interest. This result is expressed in decibels (dB) and is expected by the Simulation Kernel in the packet's OPC_TDA_RA_TX_GAIN transmission data attribute.

Transmitter Antenna Gain Computation

For all transmitted packets, the default transmitter antenna gain model (dra_tagain) computes gain based on the transmitter-to-receiver direction and the pointing direction and pattern of the transmitter's associated antenna. A special case is made for isotropic (omnidirectional) antennas, for which no computations are needed because the gain will necessarily be 0 dB.

Figure 10-30 Source Code for dra_tagain Model (Part 1)

```
/* dra_tagain.ps.c */
/* Default transmitter antenna gain model for radio link Transceiver Pipeline */

/****************************************/
/*        Copyright (c) 1993-2007       */
/*      by OPNET Technologies, Inc.     */
/*       (A Delaware Corporation)       */
/*     7255 Woodmont Av., Suite 250     */
/*      Bethesda, MD 20814, U.S.A.      */
/*         All Rights Reserved.         */
/****************************************/

#include "opnet.h"
#include <math.h>


#if defined (__cplusplus)
extern "C"
#endif
void
dra_tagain_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double      tx_x, tx_y, tx_z;
    double      rx_x, rx_y, rx_z;
    double      dif_x, dif_y, dif_z, dist_xy;
    double      rot1_x, rot1_y, rot1_z;
    double      rot2_x, rot2_y, rot2_z;
    double      rot3_x, rot3_y, rot3_z;
    double      rx_phi, rx_theta; point_phi, point_theta;
    double      bore_phi, bore_theta, lookup_phi, lookup_theta, gain;
```

```
Vartype     pattern_table;

/** Compute the gain associated with the transmitter's antenna. **/
FIN_MT (dra_tagain (pkptr));

/* Obtain handle on receiving antenna's gain. */
pattern_table = op_td_get_ptr (pkptr, OPC_TDA_RA_TX_PATTERN);

/* Special case: By convention a nil table address indicates an  */
/* isotropic antenna pattern. Thus no calculations are necessary.*/
if (pattern_table == OPC_NIL)
    {
    /* Assign zero dB gain regardless of transmission direction. */
    op_td_set_dbl (pkptr, OPC_TDA_RA_TX_GAIN, 0.0);
    FOUT;
    }
```

For the general case, dra_tagain first computes the vector from the transmitter to the receiver. If the transmitter and receiver are collocated, the gain is set to 0 dB.

Figure 10-31 Source Code for dra_tagain Model (Part 2)

```
/* Obtain the geocentric coordinates of the transmitter. */
tx_x = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GEO_X);
tx_y = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GEO_Y);
tx_z = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GEO_Z);

/* Obtain the geocentric coordinates of the receiver. */
rx_x = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GEO_X);
rx_y = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GEO_Y);
rx_z = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GEO_Z);

/* Compute the vector from the transmitter to the receiver. */
dif_x = rx_x - tx_x;
dif_y = rx_y - tx_y;
dif_z = rx_z - tx_z;

/* Special case: If transmitter and receiver are the same  */
/* then calculations are unnecessary.  We set gain = 0 */
if ((dif_x == 0) && (dif_y == 0) && (dif_z == 0))
    {
    op_td_set_dbl(pkptr, OPC_TDA_RA_TX_GAIN, 0.0);
    FOUT;
    }
```

Figure 10-32 Source Code for dra_tagain Model (Part 3)

```
/* Determine phi, theta pointing directions for antenna.       */
/* These are computed based on the target point of the antenna  */
/* module and the position of the transmitter.                 */
point_phi = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_PHI_POINT);
point_theta = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_THETA_POINT);

/* Determine antenna pointing reference direction              */
/* (usually boresight cell of pattern).                        */
/* Note that the difference in selected coordinate systems     */
/* between the antenna definiton and the geocentric axes,      */
/* is accomodated for here by modifying the given phi value.   */
bore_phi = 90.0 - op_td_get_dbl (pkptr, OPC_TDA_RA_TX_BORESIGHT_PHI);
bore_theta = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_BORESIGHT_THETA);

{
// Setup a new coordinate system originating at the antenna location
// where x axis is pointing at the antenna target
// and z axis is pointing to the "sky"
// Deterministic z-axis definition is required to make rotation
// of pattern assymetrical around the boresigh independent of the antenna location
// [more strictly, "sky" means that tangental projection of Z-axis to the surface
// tangetal to Earth at the antenna location is covered by the tangental
// projection of the X-axis to the same surface]
double cos_pt_th = cos (VOSC_NA_DEG_TO_RAD * point_theta);
double sin_pt_th = sin (VOSC_NA_DEG_TO_RAD * point_theta);
double cos_pt_ph = cos (VOSC_NA_DEG_TO_RAD * point_phi);
double sin_pt_ph = sin (VOSC_NA_DEG_TO_RAD * point_phi);

// rotate about z axis by -point_theta
double rot_x = dif_x * cos_pt_th + dif_y * sin_pt_th;
double rot_y = -dif_x * sin_pt_th + dif_y * cos_pt_th;
double rot_z = dif_z;
// rotate about y axis by -point_phi
rot1_x = rot_x * cos_pt_ph + rot_z * sin_pt_ph;
rot1_y = rot_y;
rot1_z = rot_z * cos_pt_ph - rot_x * sin_pt_ph;
}
{
// now roll around x axis to make z axis point to the "sky"
double r = sqrt (tx_x * tx_x + tx_y * tx_y + tx_z * tx_z);
double sin_lat = tx_z / r;
double cos_lat = sqrt (1 - sin_lat * sin_lat);

rot2_x = rot1_x;
rot2_y = rot1_y * sin_lat + rot1_z * cos_lat;
rot2_z = rot1_z * sin_lat - rot1_y * cos_lat;
}
{
// now rotate vector by the pattern's boresight angles
double cos_b_th = cos (VOSC_NA_DEG_TO_RAD * bore_theta);
double cos_b_ph = cos (VOSC_NA_DEG_TO_RAD * bore_phi);
double sin_b_th = sin (VOSC_NA_DEG_TO_RAD * bore_theta);
double sin_b_ph = sin (VOSC_NA_DEG_TO_RAD * bore_phi);

// first by +boresigh_phi about y axis
double rot_x = rot2_x * cos_b_ph - rot2_z * sin_b_ph;
double rot_y = rot2_y;
double rot_z = rot2_x * sin_b_ph + rot2_z * cos_b_ph;
// then by +boresigh_theta about the z axis
rot3_x = rot_x * cos_b_th - rot_y * sin_b_th;
rot3_y = rot_x * sin_b_th + rot_y * cos_b_th;
rot3_z = rot_z;
}

/* Determine x-y projected distance. */
dist_xy = sqrt (rot3_x * rot3_x + rot3_y * rot3_y);

/* For the vector to the receiver, determine phi-deflection from    */
/* the x-y plane (in degrees) and determine theta deflection from   */
/* the positive x axis.                                             */
if (dist_xy == 0.0)
    {
    if (rot3_z < 0.0)
        rx_phi = -90.0;
    else
        rx_phi = 90.0;
    rx_theta = 0.0;
    }
else
    {
    rx_phi = VOSC_NA_RAD_TO_DEG * atan (rot3_z / dist_xy);
    rx_theta = VOSC_NA_RAD_TO_DEG * atan2 (rot3_y, rot3_x);
    }

/* Setup the angles at which to lookup gain.                 */
/* In the rotated coordinate system, these are really        */
/* just the angles of the transmission vector. However,      */
/* note that here again the difference in the coordinate     */
/* systems of the antenna and the geocentric axes is         */
/* accomodated for by modiftying the phi angle.              */
lookup_phi = 90.0 - rx_phi;
lookup_theta = rx_theta;

/* Obtain gain of antenna pattern at given angles. */
```

```
gain = op_tbl_pat_gain (pattern_table, lookup_phi, lookup_theta);

/* Set the tx antenna gain in the packet's transmission data attribute. */
op_td_set_dbl (pkptr, OPC_TDA_RA_TX_GAIN, gain);

FOUT;
}
```

Default Propagation Delay Model (dra_propdel)

This section describes the functionality and implementation of the default model occupying stage five of the Radio Transceiver Pipeline. This model is named dra_propdel. The source code is provided in the file `dra_propdel.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The computation of propagation delay occurs independently for each packet transmission which successfully completes the closure and channel match stages of the pipeline. The propagation delay for the packet is the time separating the arrival of the first bit at the receiver from the transmission of the first bit at the transmitter. The propagation delay model is invoked by the Simulation Kernel to perform this computation immediately upon the completion of the transmitter antenna gain model. The simulation time at invocation is the starting time of the packet transmission.

Because the distance of propagation can vary during the packet transmission due to the possible mobility of nodes, the propagation delay model must actually compute two results. These results are the propagation delay at the beginning of packet transmission and the propagation delay at the end of packet transmission. The Simulation Kernel expects these results to be placed in the packet's OPC_TDA_RA_START_PROPDEL and OPC_TDA_RA_END_PROPDEL transmission data attributes.

Propagation Delay Computation

The default propagation delay model for radio links, dra_propdel, calculates delay based on the distance separating the transmitter and receiver, and the propagation velocity of radio waves.

dra_propdel obtains the distances separating the transmitter from the receiver at both the start and end of transmission, which are provided by the Simulation Kernel in the OPC_TDA_RA_START_DIST and OPC_TDA_RA_END_DIST transmission data attributes. Both required results can be obtained by dividing these distances by the propagation velocity of the radio signal which is represented by the locally defined symbolic constant PROP_VELOCITY, as shown in the following listing.

Figure 10-33 Source Code for dra_propdel Model

```
/* dra_propdel.ps.c */
/* Default propagation delay model for radio link Transceiver Pipeline */

/****************************************/
```

```
/*       Copyright (c) 1993-2002        */
/*     by OPNET Technologies, Inc.      */
/*      (A Delaware Corporation)        */
/*  7255 Woodmont Av., Suite 250        */
/*     Bethesda, MD 20814, U.S.A.       */
/*          All Rights Reserved.        */
/****************************************/

#include "opnet.h"


/***** constants *****/

/* propagation velocity of radio signal (m/s) */
#define PROP_VELOCITY    3.0E+08


/***** pipeline procedure *****/

#if defined (__cplusplus)
extern "C"
#endif
void
dra_propdel_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double    start_prop_delay, end_prop_delay;
    double    start_prop_distance, end_prop_distance;

    /** Compute the propagation delay separating the  **/
    /** radio transmitter from the radio receiver.    **/
    FIN_MT (dra_propdel (pkptr));

    /* Get the start distance between transmitter and receiver. */
    start_prop_distance = op_td_get_dbl (pkptr, OPC_TDA_RA_START_DIST);

    /* Get the end distance between transmitter and receiver. */
    end_prop_distance = op_td_get_dbl (pkptr, OPC_TDA_RA_END_DIST);

    /* Compute propagation delay to start of reception. */
    start_prop_delay = start_prop_distance / PROP_VELOCITY;

    /* Compute propagation delay to end of reception. */
    end_prop_delay = end_prop_distance / PROP_VELOCITY;

    /* Place both propagation delays in packet transmission data attributes. */
    op_td_set_dbl (pkptr, OPC_TDA_RA_START_PROPDEL, start_prop_delay);
    op_td_set_dbl (pkptr, OPC_TDA_RA_END_PROPDEL, end_prop_delay);

    FOUT
    }
```

Default Receiver Antenna Gain Model (dra_ragain)

This section describes the functionality and implementation of the default model occupying stage six of the Radio Transceiver Pipeline. This model is named dra_ragain. The source code is provided in the file `dra_ragain.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The Radio Transceiver Pipeline determines the gain associated with a packet's receiving antenna by invoking the receiver antenna gain model. This invocation logically follows the invocation of the propagation delay model, but these two stages are separated in time by a simulated duration equal to the propagation delay. The simulation time at invocation is the start time for the packet reception.

The result of the receiver antenna gain model for each packet is a floating point value specifying the gain of the receiver's antenna in the direction of the transmitter of interest. This result is expressed in decibels (dB) and is expected by the Simulation Kernel in the packet's OPC_TDA_RA_RX_GAIN transmission data attribute.

Receiver Antenna Gain Computation

Receiver antenna gain is computed using the same techniques found in the transmitter antenna gain model. The implementations of these two stages differ only in that receiver-related attributes are accessed in the packet rather than transmitter-related ones (e.g., OPC_TDA_RA_RX_PATTERN is used instead of OPC_TDA_RA_TX_PATTERN). In addition, the calculations are performed by placing the receiver in a coordinate system where it is at the origin, rather than the transmitter. Because the implementations are otherwise identical, the earlier description of the dra_tagain model serves as documentation for dra_ragain as well. Additional information is available in the comments of `dra_ragain.ps.c`.

Default Received Power Model (dra_power)

This section describes the functionality and implementation of the default model occupying stage seven of the Radio Transceiver Pipeline. This model is named dra_power. The source code is provided in the file `dra_power.ps.c`, which resides in the `<reldir>/models/std/wireless` directory. An alternate version of this stage, for networks not using receiver channel state information, is in the file `dra_power_no_rxstate.ps.c`.

Invocation Context

The fundamental performance measure computed by the default Radio Transceiver Pipeline is the average power level of signals received by radio receiver channels. By computing this value for every relevant signal arriving at each radio receiver channel, the Received Power Model, which is the seventh stage of the pipeline, enables later stages to compute signal-to-noise ratio (SNR) and then derive bit error rate (BER).

The computation of received power occurs independently for each packet that is able to reach and affect the radio receiver channel (this determination is made by the Link Closure Model and the Channel Match Model, which occupy stages two and three of the pipeline). The Received Power Model is invoked by the Simulation Kernel to perform this computation immediately upon the completion of the Receiver Antenna Gain Model which occupies stage six of the pipeline. This invocation occurs at the simulation time where the leading edge of the packet arrives at the receiver channel. This time is given by the sum of the transmission start time and the propagation delay, and is provided in the packet's OPC_TDA_RA_START_RX transmission data attribute.

The result of the Received Power Model invocation for each packet is a single double precision floating point value which represents the received power level for the packet. The Simulation Kernel expects this result to be placed in the packet's OPC_TDA_RA_RCVD_POWER transmission data attribute.

Reception Exclusivity

In the default construction of the Radio Transceiver Pipeline, a signal lock value associated with the radio receiver channel object is used to prevent the simultaneous correct reception of multiple packets. While the pipeline could support the multiplexing of concurrent receptions onto a single outgoing packet stream of the radio receiver module, this is considered an uncommon case and is prevented by special logic incorporated into two stages of the pipeline. These are the Received Power Model and the Error Correction Model, which occupy stages seven and thirteen, respectively. The function of providing reception exclusivity at each radio receiver channel is not the primary objective of either of these stages, but has been incorporated within them for timing reasons.

The signal lock field of the receiver channel state information is used as a boolean to represent the busy or idle state of the receiver channel. The value OPC_TRUE is used to indicate that the channel is currently processing a valid packet which has the potential to eventually be received correctly. The value OPC_FALSE represents either a completely inactive channel with no incoming packets, or one at which all currently incoming packets do not have the potential to be received correctly. The determination of the potential for correct reception is initially made for each packet by the Channel Match Model which occupies stage three of the pipeline. Packets which might be received correctly are termed *valid* packets.

The Received Power Model dra_power is used to implement verification of the channel's status, because it is invoked at the exact time of packet reception. Only valid packets which arrive at an idle receiver channel can cause the channel to become busy. For arriving invalid packets, the value of signal lock is not relevant; however, their power calculation is still performed because their noise contribution might affect the correct reception of other packets. The special processing of valid packets occurs in the first part of the *dra_power()* function, shown below.

In this segment, valid and invalid packets are distinguished by testing the value of their OPC_TDA_RA_MATCH_STATUS transmission data attribute in the opening `if` statement. For valid packets, this attribute will have been assigned the symbolic constant OPC_TDA_RA_MATCH_VALID by the Channel Match Model which occupies stage three of the pipeline. Within the `if` statement, the *op_td_get_int()* Kernel Procedure is used to obtain the object ID for the radio receiver channel at which the packet is arriving. This object ID is provided by the Simulation Kernel in the packet's OPC_TDA_RA_RX_CH_OBJID transmission data attribute. This object ID is used to access the channel's state information to check the signal lock value.

If signal lock is non-zero, indicating that the receiver channel is already busy acquiring a valid packet, the status of the newly arriving packet is overridden so that it is now considered as an invalid packet. This is done by using the *op_td_set_int()* Kernel Procedure to reassign its OPC_TDA_RA_MATCH_STATUS transmission data attribute to the symbolic constant OPC_TDA_RA_MATCH_NOISE, thus preventing the packet from later being correctly received and processed within the destination node.

If, on the other hand, signal lock is equal to 0 (OPC_FALSE), the channel is captured by the newly arriving packet, whose match status remains unchanged (the packet remains valid). The channel is then locked by assigning the signal lock attribute to OPC_TRUE.

Figure 10-34 Source Code for dra_power Model (Part 1)

```
/* dra_power.ps.c */
/* Default received power model for radio link Transceiver   */
/* Pipeline. This model uses the receiver channel state      */
/* information to check and update the signal lock status    */
/* of the channel. It relies on the rxgroup stage model for  */
/* the creation and initialization of the channel state      */
/* information.                                               */

/****************************************/
/*      Copyright (c) 1993-2003       */
/*     by OPNET Technologies, Inc.     */
/*      (A Delaware Corporation)       */
/*    7255 Woodmont Av., Suite 250     */
/*     Bethesda, MD 20814, U.S.A.      */
/*        All Rights Reserved.         */
/****************************************/

#include "opnet.h"
#include "dra.h"
#include <math.h>


/***** constants *****/

#define C               3.0E+08     /* speed of light (m/s) */
#define SIXTEEN_PI_SQ   157.91367   /* 16 times pi-squared */


/***** pipeline procedure *****/

#if defined (__cplusplus)
extern "C"
#endif

void
dra_power_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double                  prop_distance, rcvd_power, path_loss;
    double                  tx_power, tx_base_freq, tx_bandwidth, tx_center_freq;
    double                  lambda, rx_ant_gain, tx_ant_gain;
    Objid                   rx_ch_obid;
    double                  in_band_tx_power, band_max, band_min;
    double                  rx_base_freq, rx_bandwidth;
    DraT_Rxch_State_Info*   rxch_state_ptr;

    /** Compute the average power in Watts of the     **/
    /** signal associated with a transmitted packet.  **/
    FIN_MT (dra_power (pkptr));

    /* If the incoming packet is 'valid', it may cause the receiver to  */
    /* lock onto it. However, if the receiving node is disabled, then   */
    /* the channel match should be set to noise.                        */
```

```
if (op_td_get_int (pkptr, OPC_TDA_RA_MATCH_STATUS) == OPC_TDA_RA_MATCH_VALID)
    {
    if (op_td_is_set (pkptr, OPC_TDA_RA_ND_FAIL))
        {
        /* The receiving node is disabled.  Change  */
        /* the channel match status to noise.       */
        op_td_set_int (pkptr, OPC_TDA_RA_MATCH_STATUS, OPC_TDA_RA_MATCH_NOISE);
        }
    else
        {
        /* The receiving node is enabled.  Get    */
        /* the address of the receiver channel.   */
        rx_ch_obid = op_td_get_int (pkptr, OPC_TDA_RA_RX_CH_OBJID);

        /* Access receiver channels state information. */
        rxch_state_ptr =
            (DraT_Rxch_State_Info *) op_ima_obj_state_get (rx_ch_obid);

        /* If the receiver channel is already locked,        */
        /* the packet will now be considered to be noise.    */
        /* This prevents simultaneous reception of multiple  */
        /* valid packets on any given radio channel.         */
        if (rxch_state_ptr->signal_lock)
            op_td_set_int (pkptr, OPC_TDA_RA_MATCH_STATUS,
                OPC_TDA_RA_MATCH_NOISE);
        else
            {
            /* Otherwise, the receiver channel will become   */
            /* locked until the packet reception ends.       */
            rxch_state_ptr->signal_lock = OPC_TRUE;
            }
        }
    }
```

Received Power Computation

For all arriving packets, whether valid or invalid, the average power level of the received signal is computed in the remainder of the *dra_power()* function. This computation is a link budget which takes into account the initial transmitted power, the path loss, and receiver and transmitter antenna gains.

The power (in units of watts) allocated to the transmission is obtained from the packet's OPC_TDA_RA_TX_POWER transmission data attribute by using the *op_ima_td_get_dbl()* Kernel Procedure, and assigning the result to the automatic variable tx_power. The same Kernel Procedure is used to obtain the base frequency of transmission and the bandwidth of transmission from the transmission data attributes OPC_TDA_RA_TX_FREQ, and OPC_TDA_RA_TX_BW. These are assigned to the automatic variables tx_base_freq and tx_bandwidth, respectively. The values of these two variables are then used to compute the center frequency of the transmission which is held in the variable tx_center_freq. The wavelength lambda, of the packet transmission is given by the propagation velocity of light, C, divided by the center frequency, tx_center_freq. The source code for these operations is shown below.

Figure 10-35 Source Code for dra_power Model (Part 2)

```
/* Get power allotted to transmitter channel. */
tx_power = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_POWER);
```

```
/* Get transmission frequency in Hz. */
tx_base_freq = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_FREQ);
tx_bandwidth = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_BW);
tx_center_freq = tx_base_freq + (tx_bandwidth / 2.0);

/* Caclculate wavelength (in meters). */
lambda = C / tx_center_freq;
```

The propagation distance (in meters) for the packet transmission is obtained from the packet's OPC_TDA_RA_START_DIST transmission data attribute by using the *op_ima_td_get_dbl()* Kernel Procedure, and the result is assigned to the automatic variable prop_distance. If you are using terrain modeling, the value for propagation loss calculated by it is already set. Otherwise, the free space propagation loss is computed as a function of wavelength and propagation distance with the relation given in Figure 10-36.

Figure 10-36 Calculation of Free Space Propagation Loss in dra_power

$$L_p = \left(\frac{\lambda}{4\pi D}\right)^2$$

The result of this computation is placed in the automatic variable path_loss. In the case where the propagation distance is 0 (i.e., the transmitter and antenna are collocated), there is no loss and the path_loss variable is set to 1. The computation of the path_loss variable is shown in source code form below.

Figure 10-37 Source Code for dra_power Model (Part 3)

```
/* Get distance between transmitter and receiver (in meters).  */
prop_distance = op_td_get_dbl (pkptr, OPC_TDA_RA_START_DIST);

/* When using TMM, the TDA OPC_TDA_RA_RCVD_POWER will already  */
/* have a raw value for the path loss.                         */
if (op_td_is_set (pkptr, OPC_TDA_RA_RCVD_POWER))
    {
    path_loss = op_td_get_dbl (pkptr, OPC_TDA_RA_RCVD_POWER);
    }
else
    {
    /* Compute the path loss for this distance and wavelength.  */
    if (prop_distance > 0.0)
        {
        path_loss = (lambda * lambda) /
            (SIXTEEN_PI_SQ * prop_distance * prop_distance);
        }
    else
        path_loss = 1.0;
    }
```

As a final stage of the computation of receiver carrier power, the transmitter and receiver antenna gains are extracted from the packet's transmission data attributes OPC_TDA_RA_TX_GAIN and OPC_TDA_RA_RX_GAIN, respectively. The antenna gain values are obtained first in decibels (dB) and converted to non-logarithmic form so that they can be multiplied with the other variables of the link budget. The variable rcvd_power is then computed as the product of tx_power, tx_ant_gain, path_loss, and rx_ant_gain. This result is assigned to the packet's OPC_TDA_RA_RCVD_POWER transmission data attribute for use by later pipeline stages and the Simulation Kernel. These operations are shown in source code form below.

Figure 10-38 Source Code for dra_power Model (Part 4)

```
/* Determine the receiver bandwidth and base frequency. */
rx_base_freq = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_FREQ);
rx_bandwidth = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_BW);

/* Use these values to determine the band overlap with the transmitter.   */
/* Note that if there were no overlap at all, the packet would already    */
/* have been filtered by the channel match stage.                         */

/* The base of the overlap band is the highest base frequency. */
if (rx_base_freq > tx_base_freq)
    band_min = rx_base_freq;
else
    band_min = tx_base_freq;

/* The top of the overlap band is the lowest end frequency. */
if (rx_base_freq + rx_bandwidth > tx_base_freq + tx_bandwidth)
    band_max = tx_base_freq + tx_bandwidth;
else
    band_max = rx_base_freq + rx_bandwidth;

/* Compute the amount of in-band transmitter power. */
in_band_tx_power = tx_power * (band_max - band_min) / tx_bandwidth;

/* Get antenna gains (raw form, not in dB). */
tx_ant_gain = pow (10.0, op_td_get_dbl (pkptr, OPC_TDA_RA_TX_GAIN) / 10.0);
rx_ant_gain = pow (10.0, op_td_get_dbl (pkptr, OPC_TDA_RA_RX_GAIN) / 10.0);

/* Calculate received power level. */
rcvd_power = in_band_tx_power * tx_ant_gain * path_loss * rx_ant_gain;

/* Assign the received power level (in Watts) */
/* to the packet transmission data attribute. */
op_td_set_dbl (pkptr, OPC_TDA_RA_RCVD_POWER, rcvd_power);

FOUT;
}
```

Default Interference Noise Model (dra_inoise)

This section describes the functionality and implementation of the default model occupying stage eight of the Radio Transceiver Pipeline. This model is named dra_inoise. The source code is provided in the file `dra_inoise.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The Radio Transceiver Pipeline can model the interference that concurrent transmissions might impose on each other. The actual effect of interference depends on the properties of the transmitting and receiving devices and the timing of the transmissions. Stage eight of the pipeline is devoted to analyzing these parameters and quantifying the mutual interference of concurrent transmissions.

The computation of interference noise occurs only when two packets are simultaneously present at the same radio receiver. This stage will not necessarily be invoked during execution of the Radio Transceiver Pipeline if a packet experiences no collisions. On the other hand, it might be called several times if the packet of interest should collide with more than one other packet. Invocation of the interference noise model always coincides with the arrival of a packet at the receiver. This might be the arrival of the packet of interest if another packet was already in reception, or the arrival of another packet during the reception of the packet of interest. The earlier arriving packet is passed to the pipeline procedure as the first argument; the second argument is the newly arriving packet which causes the collision. The simulation time at invocation is the start of reception time for the later arriving packet.

The results of the Interference Noise Model invocation for a packet are two double precision floating point values which represent the interference noise power contributed by each packet involved in the collision towards the other. The Simulation Kernel expects that these results will be added into the respective packets' OPC_TDA_RA_NOISE_ACCUM transmission data attributes.

Interference Noise Computation

The default Interference Noise Model, called dra_inoise, filters out zero-length packet overlaps by testing the end of reception time for the earlier arriving packet against the current time. Packets that are ending at the current time can be considered to escape the collision, because in fact the overlap is really a result of the arbitrary order given to events that occur at the same simulation time.

If a non-zero width collision has occurred, dra_inoise increments the number of collisions for each packet, determines if the packets are interference packets or valid packets, and retrieves the received power levels of both packets. Computations of noise contribution are only necessary for affected valid packets, because invalid packets are not considered for reception, and their signal quality need not be determined. For valid packets, noise contributions from other packets are based on the interfering packets' power level. These computations are shown in the following listing.

Figure 10-39 Source Code for dra_inoise Model

```
/* dra_inoise.ps.c */
/* Default interference noise model for radio link Transceiver Pipeline */

/****************************************/
/*       Copyright (c) 1993-2002       */
/*     by OPNET Technologies, Inc.     */
/*      (A Delaware Corporation)       */
/*  7255 Woodmont Av., Suite 250       */
/*     Bethesda, MD 20814, U.S.A.      */
/*         All Rights Reserved.        */
/****************************************/

#include "opnet.h"

#if defined (__cplusplus)
extern "C"
#endif
void
dra_inoise_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr_prev,
    Packet * pkptr_arriv)
    {
    int         arriv_match, prev_match;
    double      prev_rcvd_power, arriv_rcvd_power;

    /** Evaluate a collision due to arrival of 'pkptr_arriv'  **/
    /** where 'pkptr_prev' is the packet that is currently    **/
    /** being received.                                       **/
    FIN_MT (dra_inoise (pkptr_prev, pkptr_arriv));

    /* If the previous packet ends just as the new one begins, this is not   */
    /* a collision (just a near miss, or perhaps back-to-back packets).      */
    if (op_td_get_dbl (pkptr_prev, OPC_TDA_RA_END_RX) != op_sim_time ())
        {
        /* Increment the number of collisions in previous packet. */
        op_td_increment_int (pkptr_prev, OPC_TDA_RA_NUM_COLLS, 1);

        /* Increment number of collisions in arriving packet. */
        op_td_increment_int (pkptr_arriv, OPC_TDA_RA_NUM_COLLS, 1);

        /* Determine if previous packet is valid or noise. */
        prev_match = op_td_get_int (pkptr_prev, OPC_TDA_RA_MATCH_STATUS);

        /* Determine if arriving packet is valid or noise. */
        arriv_match = op_td_get_int (pkptr_arriv, OPC_TDA_RA_MATCH_STATUS);

        /* If the arriving packet is valid, calculate        */
        /* interference of previous packet on arriving one.  */
        if (arriv_match == OPC_TDA_RA_MATCH_VALID)
            {
            prev_rcvd_power = op_td_get_dbl (pkptr_prev, OPC_TDA_RA_RCVD_POWER);
            op_td_increment_dbl (pkptr_arriv, OPC_TDA_RA_NOISE_ACCUM,
                prev_rcvd_power);
            }

        /* And vice-versa. */
        if (prev_match == OPC_TDA_RA_MATCH_VALID)
            {
            arriv_rcvd_power = op_td_get_dbl (pkptr_arriv, OPC_TDA_RA_RCVD_POWER);
            op_td_increment_dbl (pkptr_prev, OPC_TDA_RA_NOISE_ACCUM,
                arriv_rcvd_power);
            }
        }

    FOUT
    }
```

The Simulation Kernel automatically subtracts the average received power from the accumulated noise attribute as interfering packets complete transmission. The net effect of this process is a time-varying interference noise level, and therefore a time-varying signal-to-noise ratio (SNR) and bit error rate (BER). These values are calculated in stages ten and eleven.

Default Background Noise Model (dra_bkgnoise)

This section describes the functions and implementation of the default model that occupies stage nine of the Radio Transceiver Pipeline. This model is named dra_bkgnoise. The source code is provided in `dra_bkgnoise.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The background noise model of the pipeline can be used to account for noise sources other than explicitly modeled interferers (i.e., sources which are not other transmitter modules). These sources can be, for example, galactic, thermal, or urban noise, depending on the environment in which the receiver is operating.

The result produced by the background noise model is a double precision floating point value that represents the accumulated power of these noise sources. This value will be factored in to computing the signal to noise ratio (in the SNR pipeline stage) for the recovered signal. The Simulation Kernel expects this result to be placed in the OPC_TDA_RA_BKGNOISE transmission data attribute.

Background Noise Computation

The default procedure provided for the background noise model is dra_bkgnoise, which is a simple model accounting for a constant ambient noise level, a constant source of background noise, and a constant thermal noise at the receiver.

The ambient noise power spectral density can be used to represent sources such as urban noise in the frequency band of interest. In dra_bkgnoise, it is represented as a negligible value given by the symbolic constant AMB_NOISE_LEVEL. This value can be adjusted according to the modeled situation.

Aggregate background noise is characterized by an effective background temperature which is added to the effective device temperature of the receiver. The background temperature is represented by the symbolic constant BKG_TEMP. The noise figure of the receiver is obtained to calculate the effective device temperature, assuming an operating temperature of 290K. The sum of these temperatures, each accounting for a separate source of noise, is multiplied by the bandwidth of the receiver channel and Boltzmann’s constant to obtain the added noise contributed by the receiver to the processed signal.

This noise is added to the ambient noise to model the overall effect of inexplicitly modeled noise sources. Note that receiver amplifier gain which would apply to these noise components, is not directly modeled here. The source code for dra_bkgnoise is given in the following listing.

Figure 10-40 Source Code for dra_bkgnoise Model

```
/* dra_bkgnoise.ps.c */
/* Default background noise model for radio link Transceiver Pipeline */

/***************************************/
/*       Copyright (c) 1993-2002       */
/*      by OPNET Technologies, Inc.    */
/*      (A Delaware Corporation)       */
/*  7255 Woodmont Av., Suite 250       */
/*     Bethesda, MD 20814, U.S.A.      */
/*         All Rights Reserved.        */
/***************************************/

#include "opnet.h"

/***** constants *****/

#define BOLTZMANN       1.379E-23
#define BKG_TEMP        290.0
#define AMB_NOISE_LEVEL 1.0E-26

/***** procedure *****/

#if defined (__cplusplus)
extern "C"
#endif
void
dra_bkgnoise_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double      rx_noisefig, rx_temp, rx_bw;
    double      bkg_temp, bkg_noise, amb_noise;

    /** Compute noise sources other than transmission interference. **/
    FIN_MT (dra_bkgnoise (pkptr));

    /* Get receiver noise figure. */
    rx_noisefig = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_NOISEFIG);

    /* Calculate effective receiver temperature. */
    rx_temp = (rx_noisefig - 1.0) * 290.0;

    /* Set the effective background temperature. */
    bkg_temp = BKG_TEMP;

    /* Get receiver channel bandwidth (in Hz). */
    rx_bw = op_td_get_dbl (pkptr, OPC_TDA_RA_RX_BW);

    /* Calculate in-band noise from both background and thermal sources. */
    bkg_noise = (rx_temp + bkg_temp) * rx_bw * BOLTZMANN;

    /* Calculate in-band ambient noise. */
    amb_noise = rx_bw * AMB_NOISE_LEVEL;

    /* Put the sum of both noise sources in the packet transmission data attr. */
    op_td_set_dbl (pkptr, OPC_TDA_RA_BKGNOISE, (amb_noise + bkg_noise));

    FOUT
    }
```

Default SNR Model (dra_snr)

This section describes the functionality and implementation of the default model occupying stage ten of the Radio Transceiver Pipeline. This model is named dra_snr. The source code is provided in the file `dra_snr.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

The computation of SNR depends on the computation of background noise and that of interference noise from other transmissions. While background noise is evaluated only once for each packet transmission (at the time where reception starts), new interference sources can become active or inactive many times during the packet's reception. Thus, SNR might need to be reevaluated multiple times for a given packet. The portion of the packet which arrives at the receiver between SNR updates is called a *packet segment*. During any given packet segment, SNR holds a constant value.

The result of the SNR Model invocation for each packet segment is two double precision floating point values which represent the current average power signal to noise ratio for the received packet in decibels (dB), and the time at which the SNR was calculated. The Simulation Kernel expects these results to be placed in the packet's OPC_TDA_RA_SNR and OPC_TDA_RA_SNR_CALC_TIME transmission data attributes, respectively.

SNR Computation

The SNR computed in this stage is the ratio of the average power of the information signal to the accumulated average power of all background and interference noise sources. These values have been computed by earlier pipeline stages and are therefore available in the transmission data attributes of the packet. The default model, dra_snr, simply computes this ratio by extracting the background and interference noise average power levels from the appropriate transmission data attributes of the packet, and uses the sum of these to divide the received power level of the information signal. The result is returned in dB as required by the Simulation Kernel. These computations are shown in the following listing.

Figure 10-41   Source Code for dra_snr Model

```
/* dra_snr.ps.c */
/* Default Signal-to-Noise-Ratio (SNR) model for radio link Transceiver Pipeline */

/***************************************/
/*        Copyright (c) 1993-2002      */
/*       by OPNET Technologies, Inc.   */
/*       (A Delaware Corporation)      */
/*  7255 Woodmont Av., Suite 250       */
/*      Bethesda, MD 20814, U.S.A.     */
/*           All Rights Reserved.      */
/***************************************/

#include "opnet.h"
#include <math.h>
```

```
#if defined (__cplusplus)
extern "C"
#endif
void
dra_snr_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double      bkg_noise, accum_noise, rcvd_power;

    /** Compute the signal-to-noise ratio for the given packet. **/
    FIN_MT (dra_snr (pkptr));

    /* Get the packet's received power level. */
    rcvd_power = op_td_get_dbl (pkptr, OPC_TDA_RA_RCVD_POWER);

    /* Get the packet's accumulated noise levels calculated by the */
    /* interference and background noise stages.                   */
    accum_noise = op_td_get_dbl (pkptr, OPC_TDA_RA_NOISE_ACCUM);
    bkg_noise = op_td_get_dbl (pkptr, OPC_TDA_RA_BKGNOISE);

    /* Assign the SNR in dB. */
    op_td_set_dbl (pkptr, OPC_TDA_RA_SNR,
        10.0 * log10 (rcvd_power / (accum_noise + bkg_noise)));

    /* Set field indicating the time at which SNR was calculated. */
    op_td_set_dbl (pkptr, OPC_TDA_RA_SNR_CALC_TIME, op_sim_time ());

    FOUT
    }
```

Default BER Model (dra_ber)

This section describes the functionality and implementation of the default model occupying stage eleven of the Radio Transceiver Pipeline. This model is named dra_ber. The source code is provided in the file `dra_ber.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

Bit error rate is typically a function of signal-to-noise ratio (SNR). Thus, bit error rate is evaluated not on a packet by packet basis, but instead in each packet segment, where packet segments are defined by the points at which SNR changes. Bit error rate is evaluated retroactively at the time where a packet segment ends, because it is not usually possible to predict the timing of SNR changes. In addition, bit error rate is measured for the last segment of each packet at the time where it completes reception.

The result of the bit error rate model invocation is a double precision floating point value which represents the expected bit error rate (BER) for the received packet over the previous packet segment. The Simulation Kernel expects this result to be placed in the packet's OPC_TDA_RA_BER transmission data attribute.

BER Computation

The default model for the BER pipeline stage is called dra_ber. This model evaluates BER based on the previously computed average power SNR and also accounts for processing gain at the receiver. The SNR in dB is obtained from the OPC_TDA_RA_SNR transmission data attribute of the packet. This value is added to the processing gain (also in dB) to obtain an effective SNR. The effective SNR can be converted from log scale and expressed as E_b / N_0, where E_b is the received energy per bit (in Joules) and N_0 is the noise power spectral density (in watts/hertz). The bit error rate is derived from the effective SNR based on a modulation curve assigned to the receiver. This function, which associates a unique BER with each value of effective SNR, can be specified in the Antenna Curve Editor or via the External Model Access package. Refer to Chapter 7 Modulation Curve Editor on page WM-7-1 chapter or EMA Support on page WM-6-5 for more information about using these features.

The source code for dra_ber is shown in the following listing.

Figure 10-42 Source Code for dra_ber Model

```
/* dra_ber.ps.c */
/* Default Bit-Error-Rate (BER) model for radio link Transceiver Pipeline */

/***************************************/
/*       Copyright (c) 1993-2002      */
/*     by OPNET Technologies, Inc.    */
/*      (A Delaware Corporation)      */
/*  7255 Woodmont Av., Suite 250      */
/*     Bethesda, MD 20814, U.S.A.     */
/*         All Rights Reserved.       */
/***************************************/

#include "opnet.h"

#if defined (__cplusplus)
extern "C"
#endif
void
dra_ber_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
    {
    double      ber, snr, proc_gain, eff_snr;
    Vartype     modulation_table;

    /** Calculate the average bit error rate affecting given packet. **/
    FIN_MT (dra_ber (pkptr));

    /* Determine current value of Signal-to-Noise-Ratio (SNR). */
    snr = op_td_get_dbl (pkptr, OPC_TDA_RA_SNR);

    /* Determine address of modulation table. */
    modulation_table = op_td_get_ptr (pkptr, OPC_TDA_RA_RX_MOD);

    /* Determine processing gain on channel. */
    proc_gain = op_td_get_dbl (pkptr, OPC_TDA_RA_PROC_GAIN);

    /* Calculate effective SNR incorporating processing gain. */
    eff_snr = snr + proc_gain;

    /* Derive expected BER from effective SNR. */
    ber = op_tbl_mod_ber (modulation_table, eff_snr);

    /* Place the BER in the packet's transmission data. */
    op_td_set_dbl (pkptr, OPC_TDA_RA_BER, ber);

    FOUT
    }
```

Default Error Allocation Model (dra_error)

This section describes the functionality and implementation of the default model occupying stage twelve of the Radio Transceiver Pipeline. This model is named dra_error. The source code is provided in the file `dra_error.ps.c`, which resides in the `<reldir>/models/std/wireless` directory.

Invocation Context

Packets transmitted over radio links can be affected by both interference and background noise. These noise sources affect the quality of reception which can be characterized by an average bit error rate. This bit error rate is computed by the previously invoked BER model which occupies stage eleven of the pipeline. The task of the error allocation model is to translate the given bit error rate into an actual set of bit errors for each valid packet which is received.

Because bit error rate varies with SNR, as previously explained, the error allocation model is called to operate on packet segments defined as portions of the packet where SNR remains constant. Thus the error allocation model might be called several times to operate on the same packet as it traverses different phases of SNR. The Simulation Kernel expects this stage to compute the number of bit errors affecting each packet segment and add this value to the number of bit errors already affecting the packet as a whole. This error accumulator is located in the OPC_TDA_RA_NUM_ERRORS transmission data attribute of each packet. In addition, the Simulation Kernel expects the error allocation model to compute the actual (instantaneous as opposed to average) bit error rate affecting the packet segment and place this result in the OPC_TDA_RA_ACTUAL_BER transmission data attribute.

Error Allocation Algorithm

The dra_error error allocation model is identical in form to the dbu_error model described in Pipeline Stages/Bus Link on page GM-3-1 of *General Models* manual, though the names of the TDA fields used to access information in the transmitted packet differ slightly. The error allocation algorithm is not repeated here; see Pipeline Stages/Bus Link on page GM-3-1 for a detailed explanation.

Default Error Correction Model (dra_ecc)

This section describes the functionality and implementation of the default model occupying stage thirteen of the Radio Transceiver Pipeline. This model is named dra_ecc. The source code is provided in the file `dra_ecc.ps.c`, which resides in the `<reldir>/models/std/wireless` directory. An alternate version of this stage, for networks not using receiver channel state information, is in the file `dra_ecc_no_rxstate.ps.c`.

Invocation Context

The error correction model is the final stage of the Radio Transceiver Pipeline, and is called after the allocation of errors for the last packet segment. The simulation time at invocation corresponds to the end of packet reception (i.e., the time at which the last bit of the packet has been received).

The result of the error correction model for each received packet is a single Boolean value (i.e., OPC_TRUE or OPC_FALSE) which represents the acceptability of the packet in the destination node. A result of OPC_FALSE indicates to the Simulation Kernel that the packet should be dropped, while a result of OPC_TRUE indicates that the packet should be accepted. The Simulation Kernel expects this result to be placed in the packet's OPC_TDA_RA_PK_ACCEPT transmission data attribute.

Acceptability Test

dra_ecc determines whether a packet can be accepted by the receiver based on the application of two criteria. The first test has to do with packets that might not have been transmitted in full due to a decision at the source to abort the transmission. Truncated packets, or *runts* as they are sometimes called, are filtered out as unacceptable. The second test compares the proportion of bit errors occurring in the packet and the error correction threshold of the receiver. The source code for dra_ecc is shown in the following figure.

Figure 10-43 Source Code for dra_ecc Model

```
/* dra_ecc.ps.c */
/* Default error correction model for radio link           */
/* Transceiver Pipeline. This model uses the receiver      */
/* channel state information to update the signal lock     */
/* status of the channel. It relies on the rxgroup stage   */
/* model for the creation and initialization of the channel */
/* state information.                                      */

/****************************************/
/*       Copyright (c) 1993-2003        */
/*      by OPNET Technologies, Inc.     */
/*       (A Delaware Corporation)       */
/*     7255 Woodmont Av., Suite 250     */
/*      Bethesda, MD 20814, U.S.A.      */
/*         All Rights Reserved.         */
/****************************************/

#include "opnet.h"
#include "dra.h"

#if defined (__cplusplus)
extern "C"
#endif

void
dra_ecc_mt (OP_SIM_CONTEXT_ARG_OPT_COMMA Packet * pkptr)
{
    int                     num_errs, accept;
    OpT_Packet_Size         pklen;
    Objid                   rx_ch_obid;
    double                  ecc_thresh;
    DraT_Rxch_State_Info*   rxch_state_ptr;

    /** Determine acceptability of given packet at receiver. **/
    FIN_MT (dra_ecc (pkptr));
/* Do not accept packets that were received  */
/* when the node was disabled.               */
if (op_td_is_set (pkptr, OPC_TDA_RA_ND_FAIL))
    accept = OPC_FALSE;
else
    {
    /* Obtain the error correction threshold of the receiver. */
    ecc_thresh = op_td_get_dbl (pkptr, OPC_TDA_RA_ECC_THRESH);

    /* Obtain length of packet. */
    pklen = op_pk_total_size_get (pkptr);

    /* Obtain number of errors in packet. */
    num_errs = op_td_get_int (pkptr, OPC_TDA_RA_NUM_ERRORS);

    /* Test if bit errors exceed threshold. */
    if (pklen == 0)
        accept = OPC_TRUE;
    else
        accept = ((((double) num_errs) / pklen) <= ecc_thresh) ? OPC_TRUE :
            OPC_FALSE;
    }

/* Place flag indicating accept/reject in transmission data block. */
op_td_set_int (pkptr, OPC_TDA_RA_PK_ACCEPT, accept);

/* In either case the receiver channel is no longer locked. */
rxch_state_ptr = (DraT_Rxch_State_Info *) op_ima_obj_state_get
    (op_td_get_int (pkptr, OPC_TDA_RA_RX_CH_OBJID));
rxch_state_ptr->signal_lock = OPC_FALSE;

FOUT
}
```