

Behavioral Modeling of RF Systems With VHDL

by

Anil Sama

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Electrical Engineering

APPROVED:


Dr. James R. Armstrong, Chairman


Dr. Walling R. Cyre


Dr. Joseph G. Tront

May, 1991

Blacksburg, Virginia

LD

5655

V 855

1991

S262

C.2

Behavioral Modeling of RF Systems With VHDL

by

Anil Sama

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Behavioral modeling of RF systems with VHDL is considered and a modeling methodology is developed for modeling the I/O response of these systems. A Pulsed Doppler radar system is chosen as a representative system, and a VHDL model for this system is presented. The modeling approach and the working of the model are explained, and some example runs are provided. Some problems that are posed by VHDL in attempting to model the *behavior* of RF systems are discussed, along with the solutions that we adopted.

A fault diagnosis methodology for systems of this type that uses information about the *behavior* of the system (extracted from a VHDL model of the system) is discussed, and an example is presented.

Acknowledgements

I would like to thank Dr. James Armstrong for his support and guidance throughout this project. I would also like to thank Dr. Walling Cyre and Dr. Joseph Tront for serving as members of my committee.

In addition, I would like to acknowledge the General Dynamics Corporation for their support and funding of this project.

Table of Contents

Chapter 1. Introduction. 1

 1.1. Motivation 1

 1.2. Features of VHDL 2

Chapter 2. An Example RF System 5

 2.1. Radar - An RF System 5

 2.2. Overview of Radar 6

 2.3. Specifics of the Radar System 11

Chapter 3. The Radar System Model 13

 3.1. The Top Level Entity 13

 3.2. System Model Operation 23

 3.3. An Example Run 27

Chapter 4. Modeling Methodology 32

 4.1. Modeling Methodology 32

 4.2. The Package RADAR 34

Chapter5. The Entities of the Radar System Model	46
5.1. The Entity Descriptions	46
 Chapter 6. Some Problems Posed by VHDL and VHDL Tools	 74
6.1. Type Conversions	75
6.2. The Range Restriction Problem	76
6.3. Problems Posed by VHDL Tools	79
 Chapter 7. A Fault Diagnosis Methodology	 84
7.1. Introduction	84
7.2. Hierarchy of Paths of Interaction	91
7.3. Discrepancy Detection & Constraint Suspension	91
7.4. A Diagnosis Example	97
 Chapter 8 : Conclusions	 106
8.1. Conclusions	106
 Bibliography	 107
 Appendix A. The Package Body	 108
 Appendix B. Constraints for the Diagnosis Example	 116
 Appendix C. Pascal Code for the Noise File	 120
 Appendix D. Pascal Code for the Targets File.	 121

Appendix E. Some More Test Simulations	124
---	------------

Vita	134
-------------	------------

List of Illustrations

Figure 1. The Basic Elements of a Radar System 7

Figure 2. Operation of Pulsed Doppler Radar 8

Figure 3. Observed Doppler Shift 10

Figure 4. Block Diagram Representation of the System Model 14

Figure 5. The Diagnosis Example 96

Chapter 1. Introduction.

1.1. Motivation

Hardware description languages have traditionally been used to model digital circuits of varying sizes and complexity. These languages have been used for modeling at varied levels of abstraction; from the transistor or switch level up to the system level.

One such hardware description language is the VHSIC Hardware Description Language (VHDL) [4]. VHDL has proven to be a very powerful hardware description language and judging from the events during the past few years, it seems to be fast becoming the industry standard.

Up until recently, the power of VHDL has been demonstrated by modeling a wide range of digital circuits and systems. Work is being successfully done in using VHDL not only for chip level and system level design validation, testing, and documentation, but also as a very powerful tool for synthesis from behavioral descriptions, as was recently demonstrated at the VHDL 1991 Spring Users Group Conference [10]. However, little work

exists in the literature to date as far as the behavioral modeling of analog or mixed (digital and analog) systems is concerned. This is a growing area of interest and it is hoped that VHDL can prove to be a powerful tool in this area as well.

The prime objectives of this thesis have been to :

1. Assess the capability of VHDL as a tool to model the behavior of analog and mixed systems. By modeling the behavior of analog systems, we mean the modeling of the I/O response, and not the detailed electrical response of these systems.
2. To determine if these behavioral models could be used for system level fault diagnosis, and to suggest a fault diagnosis methodology for them.

This thesis concentrates on the first objective in considerable detail, in attempting to establish a modeling methodology for RF systems at the behavioral level, and takes a cursory look at the second objective, i.e., suggests a fault diagnosis methodology.

Modeling of analog systems in VHDL is a very young area of research, but one of growing interest, and it is hoped that this research provides some insight and ideas for future efforts.

1.2. Features of VHDL

VHDL has a few important unique features that make it suitable for attempting to model analog behavior. In particular, four features of VHDL that distinguish it from

other hardware description languages, and that make it suitable to model the behavior of analog systems are :

1. The capability of performing real number arithmetic. This capability is combined with an algorithmic approach not much unlike that of a high level programming language. Aside from performing the basic arithmetic operations (addition, subtraction, multiplication, division), this gives users the flexibility to define their own procedures and functions and expand the arithmetic capability of VHDL.
2. The definition and use of abstract data types. Apart from the basic pre-defined types like BIT, INTEGER, REAL, TIME, etc. VHDL allows users to define their own data types. Abstract data types can be defined as desired, and their units and scope can also be specified. VHDL also allows the definition of signals as a record of abstract data types. This allows basic analog types to be defined, and then an analog signal can be defined as a record of these basic analog types. Each field of this record then specifies some property of the analog signal.
3. The use of the WAIT statement. The WAIT statement is a VHDL construct that allows for realistic modeling. Timing can be incorporated into the model using the WAIT statement, which allows processes to be suspended till some condition is met. It also lets the user incorporate delay into the model, so as to model real hardware more accurately.
4. The use of File I/O. VHDL File I/O (and TEXTIO) lets the user input and output data into and from the simulation under simulation control. Analog signal data (for example random signal data generated by an external program and written to a file) can thus be generated outside of VHDL and inputted to the model by File I/O. This

is specially useful when simulations are required to be repeatable; for example in diagnosis or testing areas. Similarly, output data from the model can be written out to a file for further processing.

These features, are unique to VHDL and as will be seen in later chapters, are instrumental in allowing us to model analog behavior, and make VHDL a suitable language for modeling analog systems.

Chapter 2. An Example RF System

2.1. Radar - An RF System

In order to develop a methodology for the behavioral modeling of RF systems, we need a representative RF system. A RADAR system proves to be an excellent example of an RF system for this purpose. Radar systems are widely used and are fairly complex. They include many of the basic analog entities like amplifiers, mixers, transmitters, receivers, etc. Moreover, these systems are good examples of mixed type systems and contain both analog and digital sub-systems. Certain aspects of radar systems like the representation of radar targets, antenna movement, and the search for targets are challenging to model using VHDL. Thus, these aspects represent an interesting application of the language.

A pulsed Doppler radar system was chosen as the RF system to model. Pulsed Doppler radar systems are the most common types of radar systems encountered and are used in all commercial and military aircraft. A modeling methodology was developed, and a

model was written for this system. The model that was written represents the behavior of a generic pulsed Doppler radar system.

In order to understand the modeling process, and the modeling methodology that was developed, it is necessary to first gain a brief background of the operation of pulsed Doppler radar.

2.2. Overview of Radar

A brief overview of the operation and working of a pulsed Doppler radar system is presented below. [7,8,9]

Refer to Figure 1 on page 7 which shows the very basic elements of a radar system. The system essentially consists of an RF transmitter that transmits a very high power pulse of RF energy (typically a megawatt at 8-12 GHz for X band operation [8]) for a very short period of time. This pulse of RF energy is radiated out into the environment through a bi-directional antenna system (capable of transmitting as well as receiving). This antenna concentrates the energy into a small beam (typically 2 degrees). The RF energy that is radiated is an electromagnetic wave that travels at the speed of light. If there exists a target in the beam, it scatters this energy in all directions and part of it is radiated back towards the antenna, where it is received by the antenna during the receive cycle, and passed on to the receiver section. If the time between transmission and reception is known, then the range of the target can be found. As opposed to continuous wave radar systems, where the system transmits and receives concurrently, the operation of a pulsed radar system involves distinct non-overlapping transmit and receive cycles.

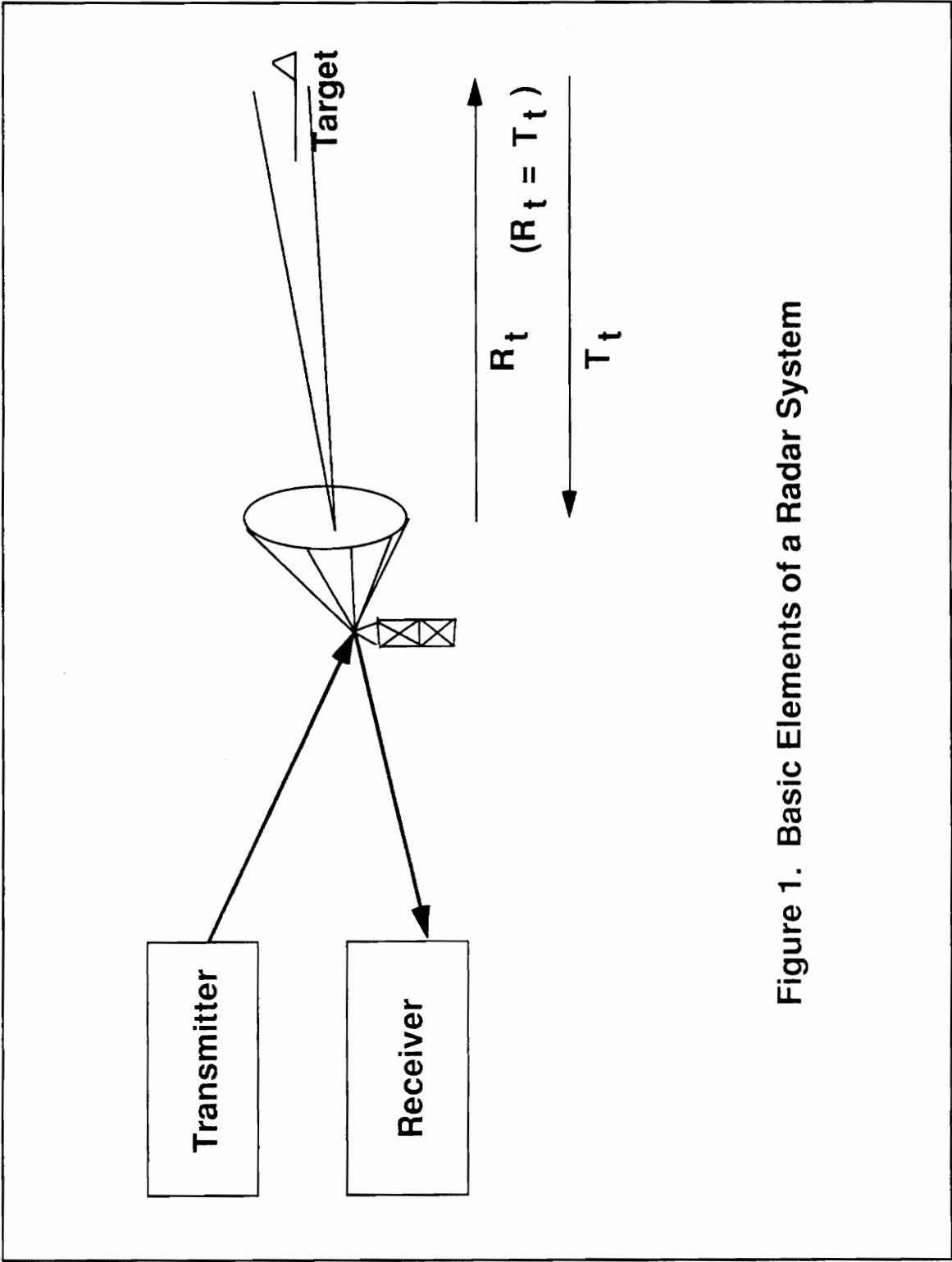


Figure 1. Basic Elements of a Radar System

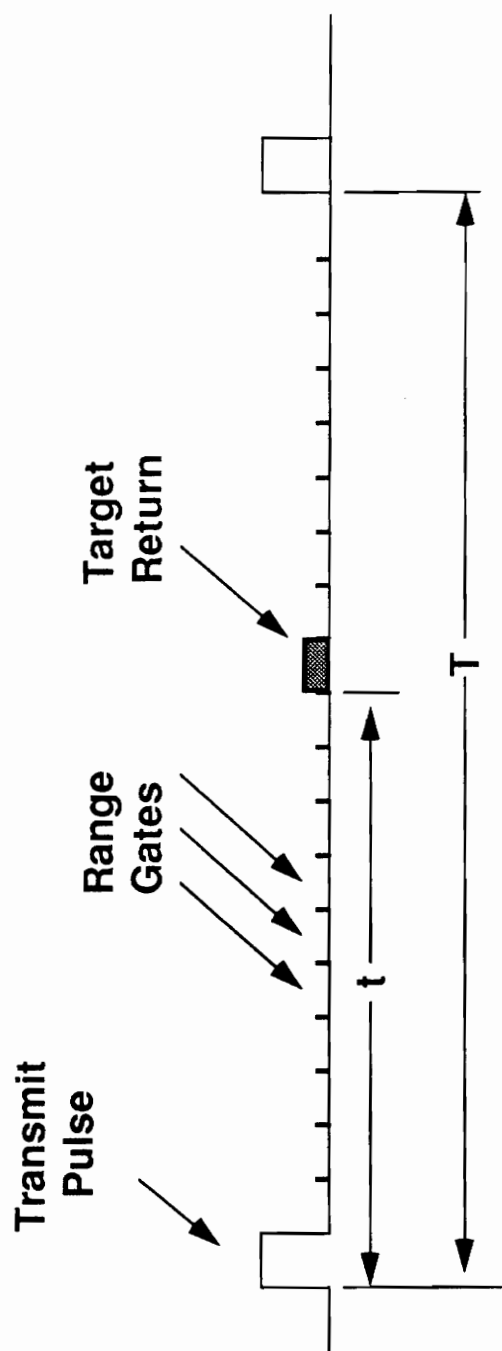


Figure 2. Pulse Radar Range Measurement

Refer to Figure 2 on page 8 which illustrates the operation of a pulsed radar system. The figure shows one complete transmit and receive cycle. The total duration of the transmit and receive cycles is 'T'. The transmit cycle consists of a pulse of RF energy. This is a short pulse that occurs at the start of the cycle. The receive cycle, wherein the radar system listens for echoes of the transmitted pulse off targets, immediately follows the transmit cycle. The receive cycle is typically many times longer than the transmit cycle. The receive cycle can be viewed as being divided into several small distinct intervals of time called "range gates" or "range bins". These can be viewed as being sequentially numbered up from zero to some maximum. If these intervals of time are counted, and the count is incremented at every interval, then a target's range can be told by the range bin count. For example, if the target return is received after a time duration 't', then the "range bin it falls into" gives an indication as to the range of the target. In actual hardware, this range gating mechanism corresponds to a sequential memory system, where the output of the receiver is dumped for analysis of the returns. The address to this memory location is provided by the range bin count.

Refer to Figure 3 on page 10 which illustrates the change in frequency of the received signal due to the Doppler effect [9]. This arises because the target has a finite velocity with respect to the line of sight of the radar system. If the wavelength of operation of the radar system is ' λ ', and the velocity of the target along the line of sight of the radar is ' v ', then the frequency of the received signal is changed by a factor $2v/\lambda$. Note that the velocity may be positive or negative, depending on whether the target is approaching the radar or travelling away from it.

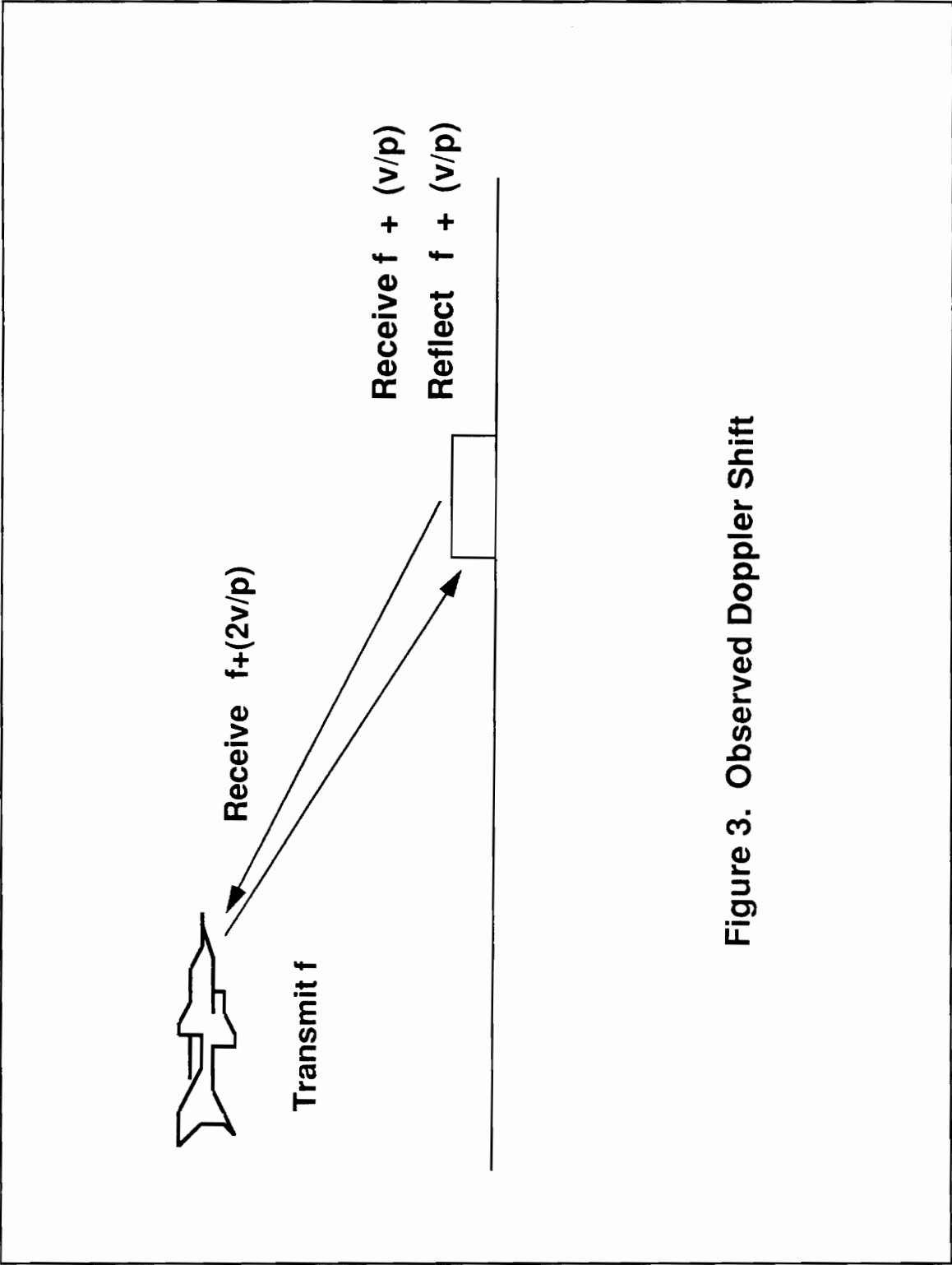


Figure 3. Observed Doppler Shift

2.3. Specifics of the Radar System

The model that was written represents a generic pulsed Doppler radar system, but it was based on one of General Dynamics Corporation's fighter aircraft RADAR systems [3]. This RADAR is a multimode pulse Doppler radar system. It consists of four major LRU's (Line Replaceable Units). These are :

1. MLPRF (Modular Low Power RF)
2. DMT (Dual Mode Transmitter)
3. PSP (Programmable Signal Processor)
4. ANTENNA

A brief description of the function of each of these units follows :

The MLPRF generates and processes the low power RF signals that are involved with the RADAR process. The STALO (Stable Local Oscillator) section of the MLPRF is responsible for generating and mixing the signals that are used to form the transmission signal at the frequency of operation, and at the required pulse width. The RCVR section of the MLPRF provides amplification, down conversion, range-gate forming, and digital conversion of the RF returns; a major part of the receive process.

The DMT section of the radar system provides the high power amplification of the radar system. It consists of a dual mode TWT amplifier. It accepts a low power X-band sig-

nal from the MLPRF and provides gating and amplification to deliver pulsed high power RF to the ANTENNA unit.

The data from the RCVR section of the MLPRF is collected by the PSP in digital form, and processed to determine target detection, range, target velocity, etc. A part of the PSP section is also responsible for the timing and control portion of the RADAR system.

The ANTENNA section receives commands from the PSP and rotates the ANTENNA in both azimuth and elevation to point the ANTENNA in a certain direction, as required by the operation. It radiates the high power X-band RF signal received from the Dual Mode Transmitter, listens for RF echoes, and delivers them to the MLPRF. The ANTENNA can be gimbaled in both directions, and can scan ± 60 degrees in either azimuth or elevation.

Chapter 3. The Radar System Model

3.1. The Top Level Entity

In order to introduce the radar system model, we first start with the top level entity (called RADAR_SYSTEM), describe its structure, and discuss the working of the model as seen from this top level. An example run is also presented, so as to illustrate what the model accomplishes.

Refer to Figure 4 on page 14 which shows a block diagram representation of the system model. This is the structure of the top level entity RADAR_SYSTEM, and is a structural composition of eight entities. These eight entities are very briefly introduced below, and their relation to the four basic LRUs introduced in chapter 2 are identified. The eight entities are :

1. The STALO entity.
2. The RCVR entity.

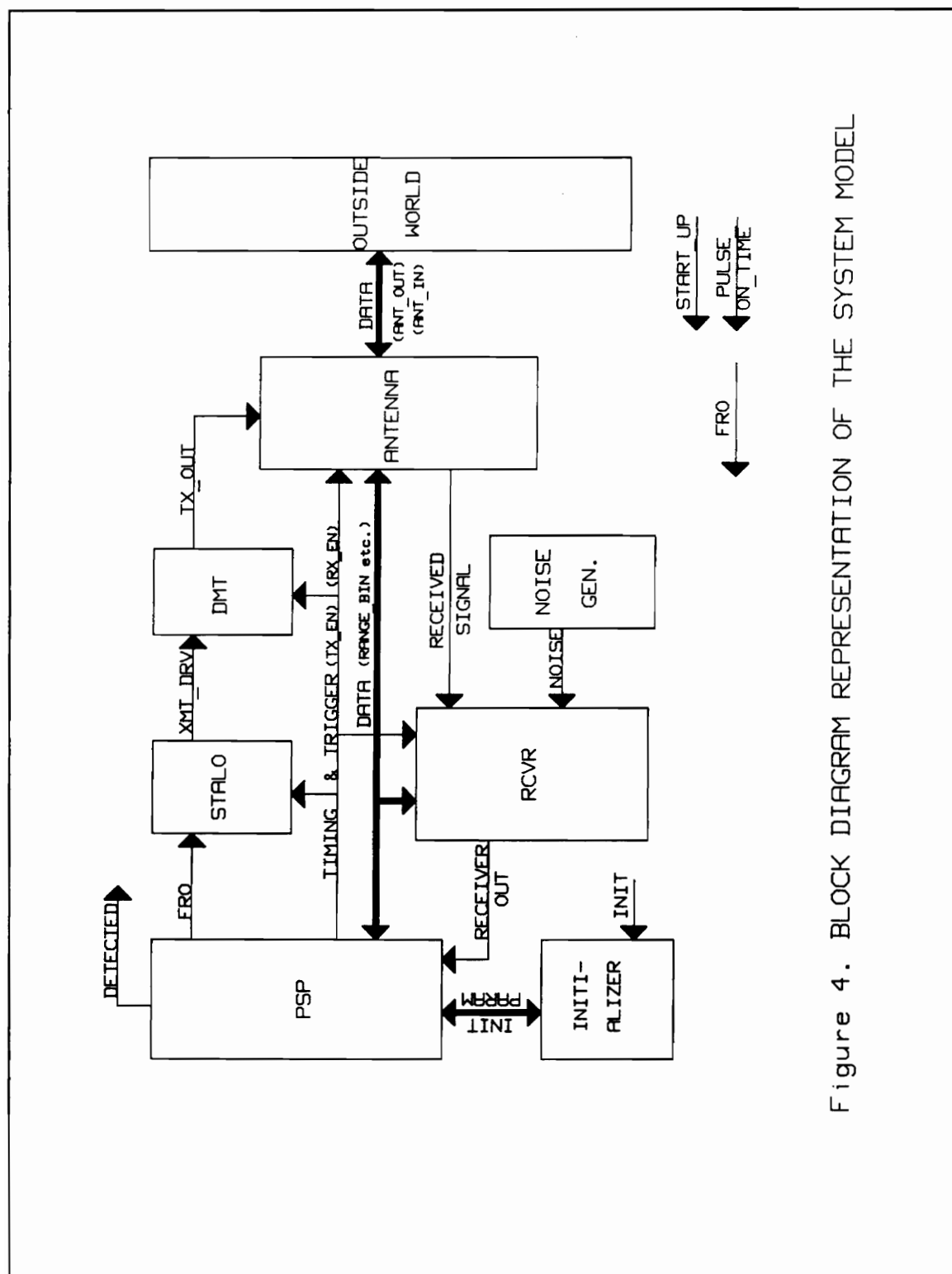


Figure 4. BLOCK DIAGRAM REPRESENTATION OF THE SYSTEM MODEL

(The STALO and RCVR entities together model the MLPRF LRU).

3. The DMT entity.

(The DMT entity models the DMT LRU)

4. The ANTENNA entity.

(The ANTENNA entity models the ANTENNA LRU)

5. The PSP entity.

(The PSP entity models the PSP LRU)

6. OUTSIDE_WORLD. This entity reads in target information from an external file at system START_UP. It is used by the ANTENNA entity to scan for radar targets. It models the target environment.

7. INITIALIZER. This entity initializes some of the signal values that will be used during the radar process. It initializes Antenna scan range, maximum detection range, etc. This can be viewed as the entity that acts as the human element in radar operation.

8. NOISE_GENERATOR. This entity produces gaussian distributed random noise in the receiver, which is amplified along with the received signal. It was introduced to more accurately model the radar process, and to model for false alarms, and missed detections.

Each of these eight entities were modeled as behavioral entities. The detailed description of each of these entities will be described in the following chapter.

The entity declaration and architecture body of the top level entity `RADAR_SYSTEM` appear below. The signals that are internal to the `RADAR_SYSTEM` as a whole are first declared. These include all the I/O ports of the eight entities. After the signal declaration section, templates are made for each of the components that make up this `RADAR_SYSTEM`. In this case, the components are the eight entities. In the main body of the architecture declaration, the components are instantiated and the ports are mapped to the signals declared above. Configuration statements are used in the declarative section of the architecture body to specify the entity and architecture to be used for the component being instantiated.

ENTITY `RADAR_SYSTEM` :

```
-----
use work.all, work.radar.all;
use STD.TEXTIO.ALL;
entity RADAR_SYSTEM is
end RADAR_SYSTEM;

use work.all, work.radar.all;
use std.TEXTIO.all;
architecture STRUCTURAL of RADAR_SYSTEM is
signal FRO, LO1, LO2, LO3, OP_FREQ : HIGH_FREQUENCY := 0 MHz;
signal TARGET_DOPPLER : LOW_FREQUENCY := 0 Hz;
signal TX_EN, RX_EN, START_UP, DETECTED, INIT : BIT := '0';
signal XMT_DRIVE, XMT_OUT, RCVD_SIG, AMP1_SIG, IF1,
         IF2, AMP2_SIG, RCVR_OUT, ANT_IN , ANT_OUT : RADAR_SIGNAL :=
         (0 MHz, 0 Hz, 0 mW, 0 pW);
signal MAX_DET_RANGE, RCVR_NOISE, AMPLIFIED_RCVR_NOISE :
```



```

        REAL := 0.0;

signal DETECTION_THRESHOLD : LOW_POWER := 0 pW;
signal AZIM_SCAN_RANGE, ELEV_SCAN_RANGE, ANGLE_ELEV,
        ANGLE_AZIM : ANGLE := 0 degrees;
signal PULSE_ON_TIME : TIME := 10 ns;
signal RANGE_BIN, NUMBER_TARGETS : NATURAL := 1;
signal RANGE_BIN_LIMIT : NATURAL;
signal FLAG : NATURAL := 0;
signal TARGET_INFO : DETECTIONS;
signal TEMP_TARGET : TARGET;
signal TARGET_MAP : TARGET_ENVIRONMENT;
signal POINTER : POSITIVE;
signal RANDOM_NOISE : GAUSSIAN_REAL;

component STALO_TEMPLATE
port (FRO : in HIGH_FREQUENCY := 0 MHz;
        TX_EN : in BIT;
        LO1, LO2, LO3 : inout HIGH_FREQUENCY := 0 MHz;
        OP_FREQ : out HIGH_FREQUENCY := 0 MHz;
        XMT_DRIVE : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW));
end component;

for L1 : STALO_TEMPLATE use entity STALO(BEHAVIOR);

component RCVR_TEMPLATE
port (RCVD_SIG : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
        RCVR_NOISE : in REAL := 0.0;

```

```

AMPLIFIED_RCVR_NOISE : out REAL := 0.0;
AMP1_SIG, IF1, IF2, AMP2_SIG : inout RADAR_SIGNAL
    := (0 MHz, 0 Hz, 0 mW, 0 pW);
RCVR_OUT : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
RX_EN, START_UP : in BIT := '0';
LO1, LO2, LO3 : in HIGH_FREQUENCY := 0 MHz);
end component;

for L2 : RCVR_TEMPLATE use entity RCVR(BEHAVIOR);

component PSP_TEMPLATE
port (START_UP, INIT : in BIT := '0';
    RCVR_OUT : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
    MAX_DET_RANGE : in REAL := 0.0;    -- Max around 160 miles.
    DETECTION_THRESHOLD : in LOW_POWER := 0 pW;
    AZIM_SCAN_RANGE : in ANGLE := 0 degrees;
    ELEV_SCAN_RANGE : in ANGLE := 0 degrees;
    FRO : in HIGH_FREQUENCY := 0 MHz;
    OP_FREQ : in HIGH_FREQUENCY := 0 MHz;
    AMPLIFIED_RCVR_NOISE : in REAL := 0.0;
    PULSE_ON_TIME : in TIME;
    RANGE_BIN : inout NATURAL;
    RANGE_BIN_LIMIT : in NATURAL;
    RX_EN, TX_EN : out BIT := '0';
    ANGLE_ELEV, ANGLE_AZIM : in ANGLE := 0 degrees;
    DETECTED : inout BIT := '0';
    TARGET_INFO : out DETECTIONS;

```

```

    TARGET_DOPPLER : inout LOW_FREQUENCY := 0 Hz);
end component;

for L3 : PSP_TEMPLATE use entity PSP(BEHAVIOR);

component DMT_TEMPLATE
port (XMT_DRIVE : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
    TX_EN : in BIT;
    XMT_OUT : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW));
end component;

for L4 : DMT_TEMPLATE use entity DMT(BEHAVIOR);

component ANTENNA_TEMPLATE
port (ANGLE_ELEV, ANGLE_AZIM : inout ANGLE := 0 degrees;
    ELEV_SCAN_RANGE, AZIM_SCAN_RANGE : in ANGLE := 0 degrees;
    XMT_IN : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
    ANT_IN : inout RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
    RANGE_BIN : in NATURAL;
    OP_FREQ : in HIGH_FREQUENCY := 0 MHz;
    START_UP, TX_EN, RX_EN, INIT : in BIT;
    RCVD_SIG, ANT_OUT : out RADAR_SIGNAL :=
        (0 MHz, 0 Hz, 0 mW, 0 pW);
    NUMBER_TARGETS : in NATURAL;
    FLAG : inout NATURAL := 1;
    TEMP_TARGET : inout TARGET;
    PULSE_ON_TIME : in TIME;

```

```

    TARGET_MAP : in TARGET_ENVIRONMENT);
end component;

for L5 : ANTENNA_TEMPLATE use entity ANTENNA(BEHAVIOR);

component OUTSIDE_WORLD_TEMPLATE
port (TARGET_MAP : out TARGET_ENVIRONMENT; START_UP : in BIT;
      NUMBER_TARGETS : out NATURAL);
end component;

for L6 : OUTSIDE_WORLD_TEMPLATE use entity
      OUTSIDE_WORLD(BEHAVIOR);

component INITIALIZER_TEMPLATE
port (INIT : in BIT;
      AZIM_SCAN_RANGE, ELEV_SCAN_RANGE : inout ANGLE := 0 degrees;
      DETECTION_THRESHOLD : out LOW_POWER := 0 pW;
      MAX_DET_RANGE : inout REAL := 0.0;
      RANGE_BIN_LIMIT : out NATURAL;
      PULSE_ON_TIME : in TIME);
end component;

for L7 : INITIALIZER_TEMPLATE use entity INITIALIZER(BEHAVIOR);

component NOISE_GENERATOR_TEMPLATE
port (POINTER : inout POSITIVE;
      RANGE_BIN : in NATURAL;
      RANDOM_NOISE : inout GAUSSIAN_REAL;
      RCVR_NOISE : inout REAL := 0.0;

```

```

    INIT : in BIT);

end component;

for L8 : NOISE_GENERATOR_TEMPLATE use entity
    NOISE_GENERATOR (BEHAVIOR);

begin

L1 : STALO_TEMPLATE
    port map(FRO, TX_EN, LO1, LO2, LO3, OP_FREQ, XMT_DRIVE);

L2 : RCVR_TEMPLATE
    port map(RCVD_SIG, RCVR_NOISE, AMPLIFIED_RCVR_NOISE,
        AMP1_SIG, IF1, IF2, AMP2_SIG, RCVR_OUT, RX_EN,
        START_UP, LO1, LO2, LO3);

L3 : PSP_TEMPLATE
    port map(START_UP, INIT, RCVR_OUT, MAX_DET_RANGE,
        DETECTION_THRESHOLD,
        AZIM_SCAN_RANGE, ELEV_SCAN_RANGE, FRO, OP_FREQ,
        AMPLIFIED_RCVR_NOISE, PULSE_ON_TIME, RANGE_BIN,
        RANGE_BIN_LIMIT, RX_EN, TX_EN, ANGLE_ELEV,
        ANGLE_AZIM, DETECTED, TARGET_INFO,
        TARGET_DOPPLER);

L4 : DMT_TEMPLATE
    port map(XMT_DRIVE, TX_EN, XMT_OUT);

```

L5 : ANTENNA_TEMPLATE

```
port map(ANGLE_ELEV, ANGLE_AZIM, ELEV_SCAN_RANGE,  
        AZIM_SCAN_RANGE, XMT_OUT, ANT_IN, RANGE_BIN,  
        OP_FREQ, START_UP, TX_EN, RX_EN, INIT, RCVD_SIG,  
        ANT_OUT, NUMBER_TARGETS, FLAG, TEMP_TARGET,  
        PULSE_ON_TIME, TARGET_MAP);
```

L6 : OUTSIDE_WORLD_TEMPLATE

```
port map(TARGET_MAP, START_UP, NUMBER_TARGETS);
```

L7 : INITIALIZER_TEMPLATE

```
port map (INIT, AZIM_SCAN_RANGE, ELEV_SCAN_RANGE,  
        DETECTION_THRESHOLD, MAX_DET_RANGE,  
        RANGE_BIN_LIMIT, PULSE_ON_TIME);
```

L8 : NOISE_GENERATOR_TEMPLATE

```
port map (POINTER, RANGE_BIN, RANDOM_NOISE, RCVR_NOISE, INIT);
```

```
PULSE_ON_TIME <= transport 10 us;
```

```
FRO <= transport 158 MHz after 1 ns;
```

```
INIT <= transport '1' after 2 ns;
```

```
START_UP <= transport '1' after 3 ns;
```

```
end STRUCTURAL;
```

3.2. System Model Operation

A brief description of the system model operation as a whole is presented here. Many of the signal names and procedure names used in this section are described in more detail in the next chapter. Only a brief description is presented here in order to follow the flow of the model.

The package RADAR that is pointed to in the entity declaration and architecture body of the top level entity is a package that contains all the analog type definitions and the procedures and functions that were defined in order to model the system. These analog types and procedures and functions are treated in detail in the next chapter in illustrating the modeling methodology that was developed.

There are four signals that are input through the top-level entity. These are :

1. Signal PULSE_ON_TIME. (On time of transmit pulse)
2. Signal FRO (Frequency of the Stable Oscillator).
3. Signal INIT.
4. Signal START_UP.

PULSE_ON_TIME is the time during which the RF energy is transmitted from the radar system in every transmit/receive cycle. It is used to control the timing and gating of the transmitted pulse, to determine the range resolution, and the number of range bins

that will be needed in order to satisfy the requirement of the desired range that the radar should operate upto. It is inputted as soon as the simulation starts.

FRO is the frequency of the stable master oscillator that is used in the STALO portion of the MLPRF. This is used to determine the frequency of operation of the radar system, and also the Local Oscillator frequencies. Since the frequency of operation is input at the top level, it is dynamically changeable. It is also inputted as soon as simulation starts.

Signal INIT is asserted after 1 ns. This initializes scan volume (+/- 60 degrees azimuth and elevation) that the antenna goes through, maximum detectable range (100 statute miles), and detection threshold (10 uW). This signal behaves like a button that a human operator would control to load in new values of the above-mentioned system parameters. If the simulation needs to be run with a different set of system parameters, the required changes need to be made in the architecture body of entity INITIALIZER. These could be defined as generic parameters or actual values could be input at simulation start if it is desired to change these values frequently.

Signal START_UP triggers the process of radar transmission and reception. When signal START_UP goes to '1', ANGLE_AZIM is at -60 degrees (60 degrees left) and ANGLE_ELEV is +60 degrees (60 degrees up), RANGE_BIN is at 0.

Shortly after START_UP is asserted (1 delta time later), TX_EN goes to '1'. TX_EN (Transmitter Enable) is the signal that, when asserted, causes the DMT and STALO sections to output an RF signal. During this time, RX_EN remains at '0'. RX_EN (Receiver Enable) is used to enable the receive process.

During this time when TX_EN is asserted, the DMT outputs the high power RF signal that is generated in the STALO section of the MLPRF to the ANTENNA.

After one PULSE_ON_TIME (10 us in this case), RX_EN goes to '1', and TX_EN goes to '0'. This stops the transmit process, and causes the receive process to start.

As soon as the receive process starts, RANGE_BIN is incremented to value 1 (up from 0). Throughout the receive process, RANGE_BIN is incremented at intervals equal to the PULSE_ON_TIME. When RANGE_BIN goes to 1, and also each time RANGE_BIN changes to a non-zero value, a LOOK_FOR_TARGET procedure in entity ANTENNA is executed. This procedure checks to see if a target is found in the beam, and if it falls in the current range bin (it will be described in detail in the next chapter). At the same time, a new value for average noise power level is picked from the array RANDOM_NOISE (this is an array of gaussian distributed noise power levels), and assigned to the input of the receiver. This is done to model false alarms or missed detections due to noise in the receiver. A false alarm is a false target detection caused by excessive noise in the receiver. A missed detection is caused by the attenuation of the otherwise detectable signal due to noise.

If there exists a target in the beam whose return would fall into the current range bin (as determined by procedure LOOK_FOR_TARGET), then ANT_IN is assigned an RF signal that corresponds to the return from that target. If there does not exist a target in the beam whose return would fall into the current range_bin, then ANT_IN is updated to a value that represents no return, i.e. zero frequency, and zero power levels. This implies that only noise is present at the input of the receiver.

As soon as ANT_IN is updated, processes in entity RCVR start to execute. The RF signal is passed through the RECEIVER_PROTECTOR stage (in the RCVR). This stage checks to see if the power_level of the returned signal is excessive. If so, the receiver section would be damaged and an assertion error occurs if the error condition is met. After the signal passes the Receiver_Protector, it is amplified in the FET_AMP stage. The output is the input signal with the power_level boosted by 30 dB.

The output of the FET_AMP is then passed to the MIXER1 stage. This is the mixer stage where the incoming radar signal is down converted from RF to IF.

The output signal from the first IF MIXER stage is then passed through an AMPLIFIER stage. The signal power_level is further boosted by 27 dB.

The signal passes through another mixer stage, MIXER2, and is further down converted. It is again down converted by MIXER3 to a video signal, RCVR_OUT. This signal is then passed to the DETECTOR in the PSP.

When RCVR_OUT is updated, process CHECK_FOR_DETECTION in the PSP (this procedure checks to see if the power level of the output of the receiver is high enough to be detectable) is executed. For this purpose the signal power levels and noise power levels (after amplification through the receiver) are added. If the power level of the resultant signal is above detectable limits, signal DETECTED is asserted.

If a target is DETECTED, procedure WRITE_TARGET (elaborated upon in the next chapter) is called, and information about the target is written to the output file. If not, the process of searching for another target continues.

The value of RANGE_BIN is incremented every PULSE_ON_TIME ns, and after RANGE_BIN_LIMIT is reached, the value of RANGE_BIN is returned to zero. At this point, procedure SCAN_ADVANCE (used to advance the antenna) is called, the antenna is advanced further, and the whole process as outlined above is repeated. This process continues until the antenna completes one entire scan of the environment.

3.3. An Example Run

Section 3.1 presented some of the basic aspects of the operation of the system model. Presented in this section is an example run of the model (a simulation) which will give an indication as to what the model accomplishes.

A file of targets (we use text files to input target information into the system) that was used in a simulation run appears below. Following that is the output file that was created by the VHDL model. Several other runs with different target files are provided in the appendix.

In the input file, the targets are listed in order by increasing angles of azimuth. This was done to reduce the time spent in looking for the target each time the value of the RANGE_BIN changed. Every five lines represents one target. The information that is provided for every target is :

- LINE 1 : Azimuth angle of the target.
- LINE 2 : Elevation angle of the target.

- LINE 3 : Time Away (An indication of the round trip range of the target)
- LINE 4 : Doppler shift (Shift in frequency that the transmitted signal undergoes after reflecting off a moving target).
- LINE 5 : Attenuation (Round Trip Attenuation that indicates the attenuation the signal underwent from the time it left the transmitter till the time it returned).

The following file is the targets file "TARGETS." that is read in at the start of simulation. This file was generated by a program written in Pascal. The Pascal code for the program appears in Appendix D. After one complete cycle, the results are output to the file DETECTED.OUT.

File "TARGETS.IN":

```
-60
59
650 us
4.5E+11
221
-57
51
490 us
1.5E+12
400
60
```

13

1203 us

4.2E + 19

338

149

-43

945 us

4.7E + 06

315

110

34

330 us

1.6E + 06

288

-103

34

592 us

2.9E + 06

298

164

58

52 us

7.6E + 5

145

27

-49

686 us

3.4E+06

222

-120

-32

377 us

1.9E+06

168

41

-5

530 us

2.6E+06

239

162

-11

846 us

4.2E+06

365

-101

16

729 us

3.6E+06

113

-159

29

910 us

4.5E + 06

249

The results of the simulation were output to the file "DETECTED.OUT". The contents of the file appear below :

TARGET DETECTED AT A DISTANCE OF:

60.47 MILES WITH A RELATIVE VELOCITY OF:

5.82 METERS PER SEC. CLOSING. IT'S POSITION IS:

60 DEGREES ELEVATION,

-60 DEGREES AZIMUTH

TARGET DETECTED AT A DISTANCE OF:

45.47 MILES WITH A RELATIVE VELOCITY OF:

10.53 METERS PER SEC. CLOSING. IT'S POSITION IS:

51 DEGREES ELEVATION,

-57 DEGREES AZIMUTH

As seen in the output of the file "DETECTED.OUT", only two targets were detected. Even though the third target in the input file was in the beam, it was not detected, as it is at a large range, and provides a much larger attenuation. All the other targets were not in the scan volume, and were not detected.

Chapter 4. Modeling Methodology

4.1. Modeling Methodology

In this chapter, some basic modeling methodology for modeling RF systems at the behavioral level is first presented.

We need to represent the behavior of an analog entity. That is, we need some way to model the relation between an analog entity's inputs and outputs. The following three points bring out the essential aspects of the methodology that was developed, as will be seen often in the model that is later presented.

1. Use of real number arithmetic.

We make use of real number arithmetic to model the relation between the analog input(s) and analog output(s) of an entity. For example, for an amplifier one can have the power level of the output as some real gain factor times the power level of the input. Generic functions and procedures can be written for analog behavior and

these can form part of a package. These functions and procedures can be called by the model.

2. Use of abstract data types.

We use abstract data types to define basic analog types that will be needed, and then define analog signals as a record of these types. After analog signals have been defined in this manner, one can refer to the fields as and when needed.

e.g., type POWER is range 0 to 1E9

units pW;

nW = 1000 pW;

uW = 1000 nW;

end units;

type FREQUENCY is range 0 to 1E9

units Hz;

KHz = 1000 Hz;

MHz = 1000 KHz;

end units;

type ANALOG_SIGNAL is

record

POWER_LEVEL : POWER;

FREQ : FREQUENCY;

end record;

3. Use of File I/O.

We make use of VHDL File I/O and TEXTIO to input data (target information and noise information) into the model and to output data (detections) from the model.

4.2. The Package RADAR

In order to see how the above methodology was applied to the radar system that was modeled, the VHDL code for the package that was defined in order to model the radar system is presented below. Following that package is a brief description of the types that were defined and the functions and procedures that were written.

```
-----  
use WORK.all, STD.TEXTIO.all;  
package RADAR is  
  
    constant PI : REAL := 3.142;  -- Value of Pi.  
  
    constant C : REAL := 3.0E8; -- Speed Of Light in meters per  
    -- second.  
  
    type LOW_FREQUENCY is range -2E9 to 2E9  
        units Hz;  
        KHz = 1000 Hz;  
    end units;
```

```
type HIGH_FREQUENCY is range -1e9 to 1E9
```

```
  units MHz;
```

```
    GHz = 1000 MHz;
```

```
  end units;
```

```
type ANGLE is range -360 to 360
```

```
  units degrees;
```

```
  end units;
```

```
type HIGH_POWER is range 0 to 2e9
```

```
  units mW;
```

```
    W = 1000 mW;
```

```
    KW = 1000 W;
```

```
  end units;
```

```
type LOW_POWER is range 0 to 1e9
```

```
  units pW;
```

```
    nW = 1000 pW;
```

```
    uW = 1000 nW;
```

```
  end units;
```

```
type RADAR_SIGNAL is
```

```
  record
```

```
    HIFREQ : HIGH_FREQUENCY;
```

```
LOFREQ : LOW_FREQUENCY;  
HIPOWER_LEVEL : HIGH_POWER;  
LOPOWER_LEVEL : LOW_POWER;  
end record;
```

type GAUSSIAN_REAL is array (INTEGER range 1 to 100) of REAL;

type TARGET is

```
record  
    AZIMUTH : ANGLE;  
    ELEVATION : ANGLE;  
    TIME_AWAY : TIME; -- in microseconds.  
    TARGET_DOPPLER : LOW_FREQUENCY; -- in Hertz  
    ATTENUATION : REAL;  
end record;
```

type TARGET_FILE is file of TARGET;

type DIRECTION is (OPENING, CLOSING);

type DETECTIONS is

```
record  
    TARGET_RANGE : REAL; -- in miles;  
    REL_VEL : REAL;  
    VEL_DIR : DIRECTION;  
    TARGET_ELEVATION : ANGLE;
```

```

    TARGET_AZIMUTH : ANGLE;
end record;

type DETECTIONS_FILE is file of DETECTIONS;

type TARGET_ENVIRONMENT is array (INTEGER range 0 to 20) of
    TARGET;

file I : TEXT is in "file.name";

file O : TEXT is out "DETECTED.OUT";

function MAX_RANGE_BIN (PULSE_ON_TIME : TIME;
                        MAX_DET_RANGE : REAL)
    return NATURAL;

function TIME_TO_REAL_IN_NS (A : TIME) return REAL;

function HIFREQ_TO_REAL_IN_MHz (A : HIGH_FREQUENCY) return REAL;

function LOFREQ_TO_REAL_IN_Hz (A : LOW_FREQUENCY) return REAL;

function ANGLE_TO_REAL_IN_DEG (A : ANGLE) return REAL;

function BIN_DISTANCE (A : TIME) return REAL;

```

```

procedure SCAN_ADVANCE (signal AZIM, ELEV : in ANGLE;
    signal ELEV_RANGE, AZIM_RANGE : in ANGLE;
    signal AZIM_1, ELEV_1 : out ANGLE);

procedure INCREMENT_RANGE_BIN (signal RANGE_BIN : in NATURAL;
    signal RANGE_BIN_2 : out NATURAL;
    signal RANGE_BIN_LIMIT : in NATURAL);

procedure READ_TARGET_ENVIRONMENT (signal TARGET_MAP : out
    TARGET_ENVIRONMENT; signal NUMBER_TARGETS : out INTEGER);

procedure WRITE_TARGET (signal TARGET_DOPPLER : in LOW_FREQUENCY;
    signal ANGLE_ELEV, ANGLE_AZIM : in ANGLE;
    signal PULSE_ON_TIME : in TIME;
    signal RANGE_BIN : in NATURAL;
    signal OP_FREQ : in HIGH_FREQUENCY;
    signal TARGET_INFO : out DETECTIONS;
    signal DETECTED : out BIT);

procedure LOOK_FOR_TARGET (signal ANGLE_ELEV,
    ANGLE_AZIM : in ANGLE;
    signal RANGE_BIN : in NATURAL;
    signal TARGET_MAP : in TARGET_ENVIRONMENT;
    signal NUMBER_TARGETS : in INTEGER;
    signal FLAG : inout NATURAL;
    signal PULSE_ON_TIME : in TIME);

```

```

procedure POTENTIAL_TARGET_INFO
    (signal TARGET_MAP_FLAG : in TARGET;
     signal ANT_OUT : out RADAR_SIGNAL;
     signal OP_FREQ : in HIGH_FREQUENCY;
     signal FLAG : out NATURAL);

procedure AMPLIFY_BY_K (variable K : in REAL;
    signal AMPLIFIER_IN : in RADAR_SIGNAL;
    signal AMPLIFIER_OUT : out RADAR_SIGNAL);

procedure CHECK_FOR_DETECTION (signal RCVR_OUT :
    in RADAR_SIGNAL;
    signal AMPLIFIED_RCVR_NOISE : in REAL;
    signal DETECTION_THRESHOLD : in LOW_POWER;
    signal DETECTED_1 : out BIT);

procedure READ_GAUSSIAN_NOISE
    (signal RANDOM_NOISE : out GAUSSIAN_REAL);

end RADAR;

```

The basic analog types (physical types): LOW_FREQUENCY, HIGH_FREQUENCY, LOW_POWER, and HIGH_POWER are defined first. Their scope and units are also defined. These are used as fields of a data type (a record - RADAR_SIGNAL) that will represent all radar signals used in the model. Type ANGLE is defined for target placing

and antenna positioning. Its range limits are from -180 to 180 degrees. This is sufficient to specify any position for the target or the antenna.

A special abstract data type TARGET is defined to represent all targets that will be seen by the radar system. It is a record of five fields, and the information contained in the fields is:

- Target positioning i.e. azimuth and elevation angles (fields 1 and 2)
- The time it takes for a target echo to return to the radar (which is a representation of its distance from the radar - field 3).
- The Doppler shift - frequency shift that comes about due to the relative velocity of the target with respect to the radar (field 4).
- The total attenuation that the radar signal undergoes from the time it leaves the transmitter to the time it reaches back to the receiver (field 5).

Type DIRECTION is defined in order to identify the direction of the target's velocity with respect to the radar. "OPENING" implies that the target's velocity has a direction that enables it to distance itself from the radar, and "CLOSING" implies just the opposite.

Type DETECTIONS is a data type that is used to represent the information about a detected target. It is a record of five fields and the information in the fields is :

- Target Range (field 1)
- Target velocity (field 2)
- Velocity direction (field 3)
- Target elevation and azimuth angles (fields 4, 5)

The data type `TARGET_ENVIRONMENT` is defined in order to represent all the targets that can possibly exist and can be detected around and about the radar system. It is a restricted array of type `TARGET`. Note that a maximum of twenty targets can be represented, since the size of the array has been constrained to that value.

Type `GAUSSIAN_REAL` is an array of type `real` that holds a string of gaussian distributed real numbers, which represents gaussian noise at the inputs of the receiver.

`MAX_RANGE_BIN` is a function that uses the pulse width of the transmitted signal, and the maximum desired detectable range, and outputs an integer value that corresponds to the maximum value that the `range_bin_counter` must count up to.

`TIME_TO_REAL_IN_NS` is a function that was defined in order to convert a signal/variable of type `time` `TIME` to one of type `REAL`. Since VHDL is very strongly typed, and does not have any pre-defined functions for conversion of physical types to real types for the purposes of calculation (since this situation never arises in digital circuit modeling), these functions have to be defined in this package. Type `TIME` is converted to a `REAL` number (relative to 1 ns) which is returned by the function.

Similarly, `HIFREQ_TO_REAL_IN_MHZ`, `LOFREQ_TO_REAL_IN_HZ`, and `ANGLE_TO_REAL_IN_DEG` convert types `HIFREQ`, `LOFREQ`, and `ANGLE` respectively to type `REAL`.

`BIN_DISTANCE` is a simple function that takes the value of type `TIME` as input, and returns a `REAL` value corresponding to the round trip range in miles that a signal would cover, if it returns to the radar in that time.

`SCAN_ADVANCE` is a procedure that takes as input the present position of the antenna in azimuth and elevation, and also the limits on the angles of azimuth and elevation which represent the maximum scan range that the antenna goes through. Every time this function is called, (provided of course that the entire scan has not been completed) it advances the antenna one position to the right in azimuth. If the azimuth limit has been reached, it advances the antenna in elevation, and returns the azimuth to its least value. The scanning of the antenna continues till an entire scan of the target environment is complete.

`INCREMENT_RANGE_BIN` is a procedure that takes the current value of the `RANGE_BIN` and increments it if the value of the range bin limit has not been reached. If the value of the limit has been reached, then `RANGE_BIN` is assigned 0.

`READ_TARGET_ENVIRONMENT` is a procedure that reads information about all the possible targets (ranging from 1 to 20 in number) that are randomly positioned anywhere about the radar system. These are read into a signal that is an array of type `TARGET` that represents target information. The file that contains the information is randomly generated by a Pascal program that generates anywhere between one and

twenty targets, positions them randomly at various azimuth and elevation angles, and assigns a random value of range, target Doppler, and attenuation to each target.

WRITE_TARGET is a procedure that takes as its input information regarding the detected target. This procedure is called whenever a target return is found to have a signal strength strong enough to be detected. The information that is passed to the procedure includes target Doppler, angles of elevation and azimuth, the range bin value of the counter at the time the target is detected, the value of the pulse width of the transmitted signal, and the operating frequency of the radar. After the necessary calculations in order to determine the range, velocity, etc., the information about the target is written out to a text file "DETECTED.OUT". The information about the target includes its approximate range, its relative velocity (to the line of sight of the radar), direction of velocity, and the position of the target (i.e. approximate azimuth and elevation angles).

LOOK_FOR_TARGET is a procedure that is called each time the value of the RANGE_BIN changes. Each time this occurs, the target map (signal that represents the target environment) is scanned to see if the current angle of azimuth and elevation that the antenna is pointing in, match with those of any of the targets (within the beamwidth of course). If they do, the procedure checks to see if target's range allows the return to fall within the current value of the range bin. If it does not fall within the current range bin, then the process continues scanning the other targets to see if they are in the beam and satisfy this condition. It does this till all the targets have been scanned. If a target is in the beam and does fall within the current range bin, then the procedure assigns to signal FLAG (integer), the array index of the possibly detectable target. This target is still only potentially detectable since it is still to be determined if this target returns a signal strong enough to be detected. At the end of the procedure, signal FLAG either

contains a non-zero value or a zero value depending on whether any target is potentially detectable.

POTENTIAL_TARGET_INFO is a procedure that is called each time a target is in the beam and in detectable range. (determined by procedure LOOK_FOR_TARGET). If a target is in the beam and its return falls within the current range bin, then the received signal ANT_IN is assigned a signal (RADAR_SIGNAL) whose power level is that of the transmitted signal divided by the value of the attenuation. Its frequency is that of the transmitted signal with the target Doppler added to it. At the same time that this is done, FLAG is reset to zero.

AMPLIFY_BY_K is a generic amplification procedure that is called from an amplifier entity. It is passed a RADAR_SIGNAL, and a generic amplification factor K as its input. It returns a RADAR_SIGNAL as its output after amplification.

Procedure CHECK_FOR_DETECTION is a procedure that takes as input a RADAR_SIGNAL (output from the receiver) and a noise signal, AMPLIFIED_RCVR_NOISE, sums them and determines if the power level of the resulting RADAR_SIGNAL is strong enough to be detected above the DETECTION_THRESHOLD. If yes, then it asserts signal DETECTED.

Procedure READ_GAUSSIAN_NOISE is a procedure that is executed at START_UP. It reads an external file of gaussian distributed real numbers and assigns them to a signal RANDOM_NOISE, which is an array of REAL and represents noise in the receiver. This external file is created externally by a Pascal program. Its code can be found in the Appendix.

The body of the package, i.e., the part of the package where all the functions and procedures are expanded upon, appears in Appendix A.

Chapter5. The Entities of the Radar System Model

5.1. The Entity Descriptions

Presented below is the main text of the VHDL code for the entity declarations and the corresponding architecture bodies of the eight behavioral entities that make up the system. At the end of each entity and its corresponding architecture body, appears a description of the functioning of the entity.

Entity STALO :

```
use work.all, work.RADAR.all;
```

```
entity STALO is
```

```
port (FRO : in HIGH_FREQUENCY := 0 MHz;
```

```
      TX_EN : in BIT;
```

```
      LO1, LO2, LO3 : inout HIGH_FREQUENCY := 0 MHz;
```

```

    OP_FREQ : out HIGH_FREQUENCY := 0 MHz;
    XMT_DRIVE : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW));
end STALO;

architecture BEHAVIOR of STALO is
begin

    GEN_LO_FREQ:
    Process (FRO)
    Begin

        LO1 <= 48 * FRO;
        LO2 <= 8 * FRO;
        LO3 <= FRO;

    end process;

    OUTPUT_SIGNAL :
    Process (TX_EN)
    begin
        If TX_EN = '1' then
            XMT_DRIVE.HIFREQ <= LO1 + LO2 + LO3;
            XMT_DRIVE.LOFREQ <= 0 Hz;
            XMT_DRIVE.HIPOWER_LEVEL <= 150 mW;
            XMT_DRIVE.LOPOWER_LEVEL <= 0 pW;
            OP_FREQ <= LO1 + LO2 + LO3;
        end if;
    end process;
end architecture;

```

```

else
    XMT_DRIVE.LOFREQ <= 0 Hz;
    XMT_DRIVE.HIFREQ <= 0 MHz;
    XMT_DRIVE.HIPOWER_LEVEL <= 0 mW;
    XMT_DRIVE.LOPOWER_LEVEL <= 0 pW;
end if;
end process;
end BEHAVIOR;

```

The entity STALO is part of the MLPRF. It generates the Local Oscillator signals and provides transmitter drive, when required. (i.e. at the given PRF and pulse width) This timing is initiated by the TX_EN signal which is generated by the PSP. The local oscillator signals are multiples of the FRO frequency which is the stable oscillator reference. LO1, LO2, LO3 are mixed (added) to form the output signal or operating frequency of the radar system. When TX_EN goes to '1', the transmitter drive signal takes on the value of the radar signal, whose power level is 150 mW (22dbm), and whose frequency is the frequency of operation. When TX_EN goes to '0', all fields of the radar signal that form the transmitter drive go to zero.

ENTITY RCVR :

```

use work.all, work.RADAR.all;

entity RCVR is
port (RCVD_SIG : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
    RCVR_NOISE : in REAL := 0.0;

```



```

AMPLIFIED_RCVR_NOISE : out REAL := 0.0;
AMP1_SIG, IF1, IF2, AMP2_SIG : inout RADAR_SIGNAL :=
    (0 MHz, 0 Hz, 0 mW, 0 pW);
RCVR_OUT : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);
RX_EN, START_UP : in BIT;
LO1, LO2, LO3 : in HIGH_FREQUENCY := 0 MHz);
end RCVR;

architecture BEHAVIOR of RCVR is
begin
-- RECEIVER_PROTECTOR :
process (RCVD_SIG)
begin
    assert not (RCVD_SIG.POWER_LEVEL > 10 mW)
    report "RECEIVED SIGNAL POWER EXCEEDED SAFE LIMIT"
    severity note;
end process;

-- FET_AMP :
process (RCVD_SIG)
variable K : REAL := 1000.0;
begin
    AMPLIFY_BY_K (K, RCVD_SIG, AMP1_SIG);
end process;

```

```

-- MIXER1 :
process(AMP1_SIG)
begin
    if AMP1_SIG.HIFREQ /= 0 MHz then
        IF1.HIFREQ <= AMP1_SIG.HIFREQ - LO1;
    else IF1.HIFREQ <= 0 MHz;
    end if;

    IF1.LOFREQ <= AMP1_SIG.LOFREQ;
    IF1.HIPOWER_LEVEL <= AMP1_SIG.HIPOWER_LEVEL;
    IF1.LOPOWER_LEVEL <= AMP1_SIG.LOPOWER_LEVEL;
end process;

-- AMPLIFIER :
process (IF1)
variable K : REAL := 500.0;
begin
    AMPLIFY_BY_K (K, IF1, AMP2_SIG);
end process;

-- MIXER2 :
process (AMP2_SIG)
begin
    If AMP2_SIG.HIFREQ /= 0 MHz then
        IF2.HIFREQ <= AMP2_SIG.HIFREQ - LO2;
    else IF2.HIFREQ <= 0 MHz;
    end if;

```

```

    IF2.LOFREQ <= AMP2_SIG.LOFREQ;
    IF2.HIPOWER_LEVEL <= AMP2_SIG.HIPOWER_LEVEL;
    IF2.LOPOWER_LEVEL <= AMP2_SIG.LOPOWER_LEVEL;
end process;

-- MIXER3 :
process (IF2)
begin
    If IF2.HIFREQ /= 0 MHz then
        RCVR_OUT.HIFREQ <= IF2.HIFREQ - LO3;--Target_doppler.
    else
        RCVR_OUT.HIFREQ <= 0 MHz;
    end if;
    RCVR_OUT.LOFREQ <= IF2.LOFREQ;
    RCVR_OUT.HIPOWER_LEVEL <= IF2.HIPOWER_LEVEL;
    RCVR_OUT.LOPOWER_LEVEL <= IF2.LOPOWER_LEVEL;
end process;

-- NOISE_AMPLIFICATION
process (RCVR_NOISE)
begin
    if START_UP = '1' then
        AMPLIFIED_RCVR_NOISE <= 5.0E5 * RCVR_NOISE;
    end if;
end process;

```

end BEHAVIOR;

Entity RCVR is also part of the MLPRF. It forms the received path of the radar signal. It consists of several processes. Receiver Protector is a process that accepts the signal RCVD_SIG which is input from the antenna and checks to see if it exceeds a certain value using an assert statement. If it does, it reports this as an assertion error in the output. This process is executed each time that the value of the signal RCVD_SIG changes.

FET_AMP is a process that is also executed each time RCVD_SIG changes. It amplifies the incoming signal to a power_level 1000 times (30 db) greater. The frequency and other fields remain unchanged. The output is called AMP1_SIG. An event on this output signals triggers another process MIXER1.

MIXER1 is a mixer stage that takes the AMP1_SIG as one of it's inputs. Local oscillator signal LO1 is the other input to this mixer stage. The frequency of the signal that comes out of the mixer stage is the first IF frequency. It is the difference of the frequency of the AMP1_SIG and the LO1 frequency. All other fields remain unchanged. The output signal is called IF1. An event on IF1 causes the process AMPLIFIER to be executed.

AMPLIFIER takes the output of the mixer stage and amplifies it to a power level 500 (27 db) times it's input. The other fields remain unchanged. The output of the AMPLIFIER is called AMP2_SIG.

An event on AMP2_SIG causes the process MIXER2 to be executed. It is another mixer stage in which the two inputs to be mixed are the AMP2_SIG and the 2nd Local Oscillator signal LO2. The frequency of the output is the difference between the frequency of the AMP2_SIG and the LO2 signal. The output is called IF2.

An event on IF2 triggers the process MIXER3. This process mixes the IF2_SIG and the LO3 signal. The frequency of the output is the difference between the frequency of the IF2_SIG and the LO3 signal. All other fields remain unchanged. The output signal RCVR_OUT is the signal coming out of the receiver. It has a low frequency and is a video signal.

NOISE_AMPLIFICATION is a process that amplifies noise at the input of the receiver by a factor equal to the amplification that the received signal goes through. The signal AMPLIFIED_RCVR_NOISE is the noise signal available at the output of the receiver. It is obtained by amplifying the noise signal RCVR_NOISE (assigned to the input of the receiver) by a factor 5.0E5, which is the same as the gain for the signal path through the receiver.

ENTITY PSP :

use work.all, work.RADAR.all;

entity PSP is

port (START_UP, INIT : in BIT;

RCVR_OUT : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);

MAX_DET_RANGE : in REAL := 0.0; -- Max around 160 miles.

DETECTION_THRESHOLD : in LOW_POWER := 0 pW;

AZIM_SCAN_RANGE : in ANGLE := 0 degrees;

```

ELEV_SCAN_RANGE : in ANGLE := 0 degrees;
FRO : in HIGH_FREQUENCY := 0 MHz;
OP_FREQ : in HIGH_FREQUENCY := 0 MHz;
AMPLIFIED_RCVR_NOISE : in REAL := 0.0;
PULSE_ON_TIME : in TIME;
RANGE_BIN : inout NATURAL;
RANGE_BIN_LIMIT : in NATURAL;
RX_EN, TX_EN : out BIT;
ANGLE_ELEV, ANGLE_AZIM : in ANGLE := 0 degrees;
DETECTED : inout BIT;
TARGET_INFO : out DETECTIONS;
TARGET_DOPPLER : inout LOW_FREQUENCY := 0 Hz);
end PSP;

```

architecture BEHAVIOR of PSP is

```

signal DETECTED_1, DETECTED_2 : BIT := '0';
signal RANGE_BIN_1, RANGE_BIN_2 : NATURAL := 0;
Begin
-- CHECK_FOR_DETECTION :
Process (AMPLIFIED_RCVR_NOISE)
begin
    if START_UP = '1' then
        CHECK_FOR_DETECTION (RCVR_OUT, AMPLIFIED_RCVR_NOISE,
                             DETECTION_THRESHOLD, DETECTED_1);
    end if;
end process;

```

```

-- DETECTION :
Process (DETECTED)
Begin
    If DETECTED = '1' and not DETECTED'STABLE then
        WRITE_TARGET (TARGET_DOPPLER, ANGLE_ELEV, ANGLE_AZIM,
            PULSE_ON_TIME, RANGE_BIN,
            OP_FREQ, TARGET_INFO, DETECTED_2);
    end if;
end process;

-- DETECTED_MUX :
DETECTED <= transport DETECTED_1 when not DETECTED_1'QUIET else
    DETECTED_2 when not DETECTED_2'QUIET else DETECTED;

-- SYNCHRONIZER :
Process (RANGE_BIN, START_UP)
Begin
    If (RANGE_BIN = 0 and START_UP = '1') then
        TX_EN <= '1';
        RX_EN <= '0';
    elsif (not (RANGE_BIN = 0) and START_UP = '1') then
        TX_EN <= '0';
        RX_EN <= '1';
    end if;
end process;

```

```

        end if;
end process;

-- RANGE_INITIALIZE:
Process (INIT)
begin
    if INIT = '1' and not INIT'STABLE then
        RANGE_BIN_1 <= 0;
    end if;
end process;

-- RANGE_INCREMENT
Process
Begin
    if START_UP = '1' then
        if(not((ANGLE_AZIM = AZIM_SCAN_RANGE) and (ANGLE_ELEV = -
            ELEV_SCAN_RANGE))
            and ((START_UP = '1' and not START_UP'STABLE) or
            (START_UP = '1' and not RANGE_BIN'STABLE))) then
            wait for PULSE_ON_TIME;
            INCREMENT_RANGE_BIN (RANGE_BIN, RANGE_BIN_2,
                RANGE_BIN_LIMIT);
        else
            wait until (not((ANGLE_AZIM = AZIM_SCAN_RANGE) and
                (ANGLE_ELEV = - ELEV_SCAN_RANGE)) and
                ((START_UP = '1' and not START_UP'STABLE) or
                (START_UP = '1' and not RANGE_BIN'STABLE)));

```



```

        end if;

        else

            wait until START_UP = '1';

            end if;

end process;


-- RANGE_BIN_MUX :
RANGE_BIN <= transport RANGE_BIN_1 when not RANGE_BIN_1'QUIET else
                RANGE_BIN_2 when not RANGE_BIN_2'QUIET else
                RANGE_BIN;


-- ASSIGN_TARGET_DOPPLER :
TARGET_DOPPLER <= RCVR_OUT.LOFREQ;


end BEHAVIOR;

```

The entity PSP is the heart of the system. It takes care of all the timing and control associated with the radar process. Most of the decision making occurs in this entity. The PSP is mostly digital. In actuality, almost all signals like AZIM_ANGLE, AZIM_SCAN_RANGE have their values digitally encoded. However, since we wish to deal with them as if they are physical types in VHDL, we have defined them as such. Some of the signals are bits. These are mostly for control purposes. For example, START_UP and INIT are signals of type bit. They are input by the user when initialization and start_up are required.

CHECK_FOR_DETECTION is a process that is triggered each time the value of RCVR_OUT (the signal out of the receiver portion), or AMPLIFIED_RCVR_NOISE changes. If the value of the power_level of the output of the receiver exceeds the detection threshold, signal DETECTED is asserted. The assertion of DETECTED causes process DETECTION to execute. This process passes the necessary information regarding the target that was detected, and some of the system parameters to procedure WRITE_TARGET (declared in package RADAR). This procedure performs the necessary calculations, and writes the target detection out to the output file. At the same time that this is done, DETECTED is de-asserted.

Process DETECTED_MUX is a process that is used so that signal DETECTED receives the value of DETECTED_1 or DETECTED_2 whichever has changed most recently. In actual hardware, this corresponds to time multiplexing.

SYNCHRONIZER is a process that is triggered each time RANGE_BIN changes value or at system START_UP. (Note that mostly all the processes in each entity will execute as required only after system START_UP as this condition has been inserted in the process control statements). For this particular process, every time RANGE_BIN changes to a 0 after system START_UP, TX_EN is asserted, and RX_EN is deasserted. These are inputs to the MLPRF. The DMT outputs a transmitter drive (non-zero) only when TX_EN is asserted; whereas the antenna and receive processes update signals ANT_IN and RCVD_SIG only when RX_EN is asserted and TX_EN is deasserted. This is the case whenever RANGE_BIN is non-zero.

RANGE_INITIALIZE simply initializes RANGE_BIN to 0 when the INIT signal is asserted.

RANGE_INCREMENT is a process that waits until START_UP = '1'. When START_UP = '1', it checks to see if RANGE_BIN or START_UP have just changed, and also if the antenna has not completed one entire scan. If these conditions are satisfied, then the process waits for one PULSE_ON_TIME and then increments the range_bin value by calling procedure INCREMENT_RANGE_BIN. If any of these conditions are not satisfied, and START_UP = '1', then it waits until all the conditions are satisfied. This is how the RANGE_BIN value is incremented every PULSE_ON_TIME.

RANGE_BIN_MUX is similar to DETECTED_MUX. It is needed as two separate processes affect the value that RANGE_BIN takes on. Since two drivers can not drive the same port at the same time, the mux function is necessary.

Lastly process ASSIGN_TARGET_DOPPLER is a process in which TARGET_DOPPLER is assigned the value of RCVR_OUT.LOFREQ each time RCVR_OUT changes.

ENTITY DMT :

```
-----
use work.all, work.RADAR.all;

entity DMT is
port (XMT_DRIVE : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW));
    TX_EN : in BIT;
    XMT_OUT : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW));
end DMT;
```

architecture BEHAVIOR of DMT is

begin

Process (TX_EN'DELAYED)

Begin

 If TX_EN'DELAYED = '1' then

 XMT_OUT.HIFREQ <= XMT_DRIVE.HIFREQ;

 XMT_OUT.LOFREQ <= XMT_DRIVE.LOFREQ;

 XMT_OUT.HIPOWER_LEVEL <= 15 kW;

 XMT_OUT.LOPOWER_LEVEL <= 0 pW;

 else

 XMT_OUT.LOFREQ <= 0 Hz;

 XMT_OUT.HIFREQ <= 0 MHz;

 XMT_OUT.LOPOWER_LEVEL <= 0 pW;

 XMT_OUT.HIPOWER_LEVEL <= 0 mW;

 end if;

end process;

end BEHAVIOR;

Entity DMT receives the transmitter drive from the MLPRF. It also receives the TX_EN signal from the PSP. When TX_EN is a '1', the output of the DMT is the transmitter drive signal amplified to a power level of 15 kW. Otherwise, the DMT does not output an RF signal. Notice that TX_EN'DELAYED is used in the sensitivity list of this process, since it takes one delta time for TX_EN to change to a '1' (in the PSP) after RANGE_BIN becomes a 0.

ENTITY ANTENNA :

```
-----  
use work.all, work.RADAR.all;  
  
entity ANTENNA is  
port (ANGLE_ELEV, ANGLE_AZIM : inout ANGLE := 0 degrees;  
      ELEV_SCAN_RANGE, AZIM_SCAN_RANGE : in ANGLE := 0 degrees;  
      XMT_IN : in RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);  
      ANT_IN : inout RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);  
      RANGE_BIN : in NATURAL;  
      OP_FREQ : in HIGH_FREQUENCY := 0 MHz;  
      START_UP, TX_EN, RX_EN, INIT : in BIT;  
      RCVD_SIG, ANT_OUT : out RADAR_SIGNAL := (0 MHz, 0 Hz, 0 mW, 0 pW);  
      NUMBER_TARGETS : in NATURAL;  
      FLAG : inout NATURAL := 1;  
      TEMP_TARGET : inout TARGET;  
      PULSE_ON_TIME : in TIME;  
      TARGET_MAP : in TARGET_ENVIRONMENT);  
end ANTENNA;  
  
use work.all, work.RADAR.all;  
use std.textio.all;  
  
architecture BEHAVIOR of ANTENNA is  
signal ANGLE_ELEV_1, ANGLE_ELEV_2, ANGLE_AZIM_1,  
       ANGLE_AZIM_2 : ANGLE;  
  
begin  
  
-- SCAN :
```

Process (RANGE_BIN, START_UP)

Begin

```
if(not((ANGLE_AZIM = AZIM_SCAN_RANGE) and (ANGLE_ELEV = -  
    ELEV_SCAN_RANGE)) and  
    (RANGE_BIN = 0 and START_UP'STABLE and START_UP = '1'))
```

then

```
SCAN_ADVANCE (ANGLE_AZIM, ANGLE_ELEV, ELEV_SCAN_RANGE,  
    AZIM_SCAN_RANGE, ANGLE_AZIM_1, ANGLE_ELEV_1);
```

end if;

end process;

-- ASSIGN_SCAN_LIMITS

Process (START_UP)

Begin

```
if START_UP = '1' and not START_UP'STABLE then  
    ANGLE_AZIM_2 <= - AZIM_SCAN_RANGE;  
    ANGLE_ELEV_2 <= ELEV_SCAN_RANGE;
```

end if;

end process;

-- ASSIGN_ANTENNA_I/O

Process (ANT_IN, XMT_IN)

Begin

```
if (not (RANGE_BIN = 0) and (RX_EN = '1')) then  
    RCVD_SIG <= ANT_IN;
```

end if;

```

    if (RANGE_BIN = 0 and TX_EN = '1') then
        ANT_OUT <= XMT_IN;
    end if;
end process;

-- ANGLE_ELEV_MUX
ANGLE_ELEV <= transport ANGLE_ELEV_1 when not ANGLE_ELEV_1'QUIET
    else ANGLE_ELEV_2 when not ANGLE_ELEV_2'QUIET else
    ANGLE_ELEV;

-- ANGLE_AZIM_MUX
ANGLE_AZIM <= transport ANGLE_AZIM_1 when not ANGLE_AZIM_1'QUIET
    else ANGLE_AZIM_2 when not ANGLE_AZIM_2'QUIET else
    ANGLE_AZIM;

-- ASSIGN_TEMP_TARGET :
Process (FLAG)
begin
    if not (FLAG'STABLE) and (START_UP = '1') and not (FLAG = 0) then
        TEMP_TARGET <= TARGET_MAP(FLAG);
    end if;
end process;

-- CHECK_POTENTIAL_TARGET :
Process (FLAG'DELAYED)
begin

```

```

if (not (FLAG'DELAYED = 0)) and (START_UP = '1') and
    not FLAG'DELAYED'STABLE then
    POTENTIAL_TARGET_INFO (TEMP_TARGET, ANT_IN, OP_FREQ,
        FLAG);
elsif START_UP = '1' and not FLAG'DELAYED'STABLE then
    ANT_IN.LOFREQ <= 0 Hz;
    ANT_IN.HIFREQ <= 0 MHz;
    ANT_IN.HIPOWER_LEVEL <= 0 mW;
    ANT_IN.LOPOWER_LEVEL <= 0 pW;
end if;
end process;

-- TARGET_SEARCH :
Process (RANGE_BIN)
begin
    if (not (RANGE_BIN = 0) and not RANGE_BIN'STABLE) then
        LOOK_FOR_TARGET (ANGLE_ELEV, ANGLE_AZIM, RANGE_BIN,
            TARGET_MAP, NUMBER_TARGETS, FLAG,
            PULSE_ON_TIME);
    end if;
end process;

end BEHAVIOR;

```

Entity ANTENNA performs antenna positioning and receives the signal returned from a target if the target is in range, and in the beam. In addition, it also outputs the high power RF signal from the output of the DMT to the exterior environment in the direction of the beam. It uses the signal TARGET_MAP to determine if any target lies within the beam at a given range. Other signals input to it include system parameters like operating frequency and range bin, Antenna Scan Limits, the high power RF signal from the DMT, the returned signal from the target, and timing signals from the PSP section.

Process SCAN is executed after START_UP when RANGE_BIN takes on the value 0 for the second time and every time thereafter, until one complete scan of the environment is done (the first time RANGE_BIN is zero is at START_UP and at this time the antenna is already stowed at the starting position, so it is only from the second time that range bin goes to zero that the antenna needs to be moved in order to scan the environment). The antenna is scanned by calling procedure SCAN_ADVANCE written in package RADAR. The parameters passed to this procedure are the angles of Azimuth and Elevation that the antenna is currently in and also the limits to the angles of Azimuth and Elevation that the Antenna should scan to. The antenna has a beam-width of two degrees, and is moved three degrees in azimuth each time the scan is advanced. If the azimuth limit is reached, the azimuth is returned to its least value, and the antenna is scanned in elevation by three degrees. This whole process repeats until one complete scan of the desired portion of the environment is completed.

Process ASSIGN_SCAN_LIMITS is executed at START_UP. This points the antenna to the starting position. The starting position is specified by the

AZIM_SCAN_RANGE and ELEV_SCAN_RANGE signals that are initialized through by the user when INIT is asserted.

ASSIGN_ANTENNA_I/O is a process that is executed each time that ANT_IN or XMT_IN changes value. XMT_IN is the high power RF input to the antenna that is provided by the output of the DMT to be output into the environment; and the ANT_IN signal is the radar_signal that is returned by a target in the beam if it is in range. So if ANT_IN changes, then RCVD_SIG is assigned ANT_IN. RCVD_SIG is the RF output from the antenna section into the receiver section. If XMT_IN changes, ANT_OUT is assigned XMT_IN. ANT_OUT is the signal that is output from the antenna during the transmit cycle.

Processes ANGLE_ELEV_MUX and ANGLE_AZIM_MUX are similar to the RANGE_BIN_MUX and DETECTED_MUX processes. They are required because output from more than one process changes the value of signals ANGLE_ELEV and ANGLE_AZIM respectively.

ASSIGN_TEMP_TARGET executes whenever FLAG changes value. When FLAG changes to a non-zero value after START_UP, TEMP_TARGET (a signal of type TARGET) is assigned that target from the TARGET_MAP array that appears to be in the beam and whose range falls in the current value of the range bin.

The TARGET_SEARCH process is executed each time RANGE_BIN changes value. If RANGE_BIN changes to a non-zero value, the LOOK_FOR_TARGET procedure written in the package RADAR is called. Parameters passed to it are the position of the antenna, the TARGET_MAP, and some system operating parameters, along with the current value of the RANGE_BIN. Procedure LOOK_FOR_TARGET scans the envi-

ronment to check to see if any targets lie in the beam and if they do, checks if their corresponding range would be such as to return a signal in the current range_bin. If not, flag remains zero, If there is such a target, then flag changes to a value that points to the target in the TARGET_MAP array.

Process CHECK_POTENTIAL_TARGET is executed a delta time after FLAG changes value (so that procedure LOOK_FOR_TARGET may be run in that one delta time), procedure POTENTIAL_TARGET_INFO declared in package RADAR is executed. To it are passed parameters like signal TEMP_TARGET which is a signal of type target and is the target that signal FLAG points to in the array TARGET_MAP. Also passed are system parameters like operating frequency. POTENTIAL_TARGET_INFO will use this information to assign signal ANT_IN with the return that is received from this target. This target is a potentially detectable target, since it is still to be determined in the PSP whether this target return will have a sufficient power_level; hence the name for this process.

FLAG_MUX is a process that was written to resolve the value of signal FLAG, since it is assigned a value from two different processes.

ENTITY OUTSIDE_WORLD :

```
-----  
use work.all, work.RADAR.all;  
entity OUTSIDE_WORLD is  
port (TARGET_MAP : out TARGET_ENVIRONMENT; START_UP : in BIT;  
      NUMBER_TARGETS : out NATURAL);  
end OUTSIDE_WORLD;
```

```

architecture BEHAVIOR of OUTSIDE_WORLD is
begin
-- POWER_UP :
process (START_UP)
begin
    If START_UP = '1' and not START_UP'STABLE then
        READ_TARGET_ENVIRONMENT (TARGET_MAP, NUMBER_TARGETS);
    end if;
end process;
end BEHAVIOR;

```

OUTSIDE_WORLD is an entity that represents the environment around the radar system. At system START_UP, the target scenario is loaded into the system through this entity. The target scenario is stored in a file. This file is read and information about the targets are assigned to a signal TARGET_MAP which is an array of type TARGET. The number of targets that are present in the file is also input into the system by means of a signal called NUMBER_TARGETS. The behavior of this entity is fairly straightforward. When signal START_UP is asserted, procedure READ_TARGET_ENVIRONMENT is called. This procedure uses package TEXTIO to read in the information from the text file and performs type conversions to assign them to an array of signals of type TARGET.

ENTITY INITIALIZER :

```
-----  
use work.all, work.RADAR.all;  
use std.TEXTIO.all;  
entity INITIALIZER is  
port (INIT : in BIT;  
      AZIM_SCAN_RANGE, ELEV_SCAN_RANGE : inout ANGLE := 0 degrees;  
      DETECTION_THRESHOLD : out LOW_POWER := 0 pW;  
      MAX_DET_RANGE : inout REAL := 0.0;  
      RANGE_BIN_LIMIT : out NATURAL;  
      PULSE_ON_TIME : in TIME);  
end INITIALIZER;
```

```
use work.all,work.RADAR.all;  
use std.TEXTIO.all;  
architecture BEHAVIOR of INITIALIZER is  
begin  
  -- POWER_UP_INITIALIZATION :  
  Process (INIT)  
  Begin  
    If (INIT = '1' and not INIT'STABLE) then  
      AZIM_SCAN_RANGE <= 60 degrees;  
      ELEV_SCAN_RANGE <= 60 degrees;  
      DETECTION_THRESHOLD <= 100 uW;  
      MAX_DET_RANGE <= 100.0; -- miles.
```

```

        end if;
end process;

Process (INIT'DELAYED)
begin
    if INIT'DELAYED = '1' then
        RANGE_BIN_LIMIT <= MAX_RANGE_BIN (PULSE_ON_TIME,
            MAX_DET_RANGE);
    end if;
end process;
end BEHAVIOR;

```

Entity INITIALIZER is used to act as the human interface between the radar system and the user. It initializes certain signals before the radar system starts functioning. In effect, it gives the radar system some information as to what the maximum detectable range should be, what detection threshold should be set at, and what the scan volume should be. It is executed when signal INIT is asserted. The RANGE_BIN_LIMIT is also set at this time (one delta time later in order to allow signal MAX_DET_RANGE to be assigned its new value). Function MAX_RANGE_BIN in package RADAR is used to compute the RANGE_BIN_LIMIT, given the maximum detectable range desired, and the pulse_on_time of the transmitter.

ENTITY NOISE_GENERATOR :

```
-----  
use work.all, work.RADAR.all;  
use std.textio.all;  
entity NOISE_GENERATOR is  
port (POINTER : inout POSITIVE;  
      RANGE_BIN : in NATURAL;  
      RANDOM_NOISE : inout GAUSSIAN_REAL;  
      RCVR_NOISE : inout REAL := 0.0;  
      INIT : in BIT);  
end NOISE_GENERATOR;  
  
use work.all, work.RADAR.all;  
use std.TEXTIO.all;  
architecture BEHAVIOR of NOISE_GENERATOR is  
begin  
  -- READ_NOISE_FILE :  
  process (INIT)  
  begin  
    if INIT = '1' then  
      READ_GAUSSIAN_NOISE(RANDOM_NOISE);  
    end if;  
  end process;  
  -- ASSIGN_RCVR_NOISE :  
  process (RANGE_BIN)  
  begin
```

```

    RCVR_NOISE <= RANDOM_NOISE(POINTER);
end process;
-- UPDATE_POINTER :
process (RCVR_NOISE)
begin
    if POINTER < 100 then
        POINTER <= POINTER + 1;
    else
        POINTER <= 1;
    end if;
end process;
end BEHAVIOR;

```

Entity NOISE_GENERATOR was introduced to more accurately model the radar process. In order to model for false alarms (where a target that should not be detected is detected due to excess noise in the receiver), and to model for missed detections (where a target that should be detected is not detected due to noise in the receiver), we need some way to introduce noise into the receiver. This was achieved by reading a file of gaussian distributed (random) real numbers which were to represent randomly distributed noise power levels at the input to the receiver. Since this noise is bipolar, after amplification in the receiver, it tends to either enhance or attenuate the received signal strength. If the received signal strength (of a detectable target) is attenuated due to noise to a power level less than that required for detection, a missed detection scenario is modeled. If the noise power levels enhance the power level of an otherwise undetectable return to the point where it becomes detectable, a false alarm scenario is modeled.

When INIT is asserted, procedure READ_GAUSSIAN_NOISE is called which reads in this file of gaussian distributed real numbers. These numbers are assigned to a signal RANDOM_NOISE which is an array of real.

Each time the RANGE_BIN signal changes value, process ASSIGN_RCVR_NOISE is executed in which a noise level is assigned to the input of the receiver. This is achieved by assigning a real value from the array RANDOM_NOISE to the RCVR_NOISE signal. The value of POINTER (integer) is used to point to this noise level within the array.

The value of pointer is also updated at each new value of the RANGE_BIN signal. It is accomplished by process UPDATE_POINTER which is executed each time RCVR_NOISE changes.

Chapter 6. Some Problems Posed by VHDL and VHDL Tools

In previous chapters, a modeling methodology for modeling RF systems was developed, and a RADAR system was modeled using this methodology. Though the RADAR system model works well, there are a few inherent problems that are posed by VHDL in modeling RF systems.

These problems will be discussed here, and the solutions that were adopted, along with the consequences of those solutions are also presented.

In addition, it was found that there is a vast difference in the way in which VHDL tools simulate these models. Tests were run using two simulators in particular. A discussion pertaining to these tests is also presented.

6.1. Type Conversions

Since VHDL is a very strongly typed language, it does not allow signals or variables of different types to be used in a mathematical relation for purposes of calculation. Since, the modeling methodology heavily involves the use of arithmetic to model an analog entity, values of all physical types must first be converted to real numbers before they can be used in a mathematical relation for the purposes of calculation. This is currently achieved by first converting the physical type to a universal integer by dividing it by one unit of its base type. Once this is done, the `REAL` operator is used on this universal integer to convert it to a real number. This real number is then used in calculations, and needs to be converted back to its physical type after the calculations are done. This is achieved by multiplying the intermediate real value obtained after calculations by one unit of the base type.

An example is presented below. Suppose that the `POWER` of an analog signal `ANALOG_IN` needs to be multiplied by a real factor `K` and assigned to an analog signal called `ANALOG_OUT`, and that the base unit of `POWER` is `pW`. The following VHDL code achieves this.

```
TEMP_REAL := REAL(ANALOG_IN.POWER/1 pW);  
ANALOG_OUT.POWER <= (K*TEMP_REAL)*1 pW;
```

Even though this procedure is straight-forward, the problem faced due to this procedure is that simulator overhead is required in order to perform type conversions and real number arithmetic. Moreover, since calculations need to be performed using real numbers (as opposed to performing calculations with abstract data types that have integer

representations), these take a longer time (since floating point arithmetic takes longer to accomplish relative to integer arithmetic). As a result of this, simulation times are longer than they would have been otherwise. This would be true particularly when the simulation is expected to perform calculations heavily and repeatedly, and when these calculations involve physical types.

6.2. The Range Restriction Problem

Another problem that VHDL poses is the restriction on the range of values that physical types can take on. Physical types can only take on values ranging from approximately $-2E9$ to $2E9$. This poses a problem when attempting to model a RADAR system for two reasons :

- It is required to represent a wide range of power levels; from Mega watts (during transmission) to several pico watts (reception). Clearly, this cannot be achieved with the restriction on the range of values that physical types can take on. Since, if the base unit of an abstract type POWER is defined as pico watts, then a signal of type POWER can at most represent a power level of approximately 2 mW.
- A similar problem is faced in representing frequency. The frequency of operation of the system is well up in the X band (8 - 12 GHz). Whereas, the Doppler frequency that we need to represent is anywhere from a few Hz to several KHz. There again arises a problem, since if the base unit of type FREQUENCY is defined as Hz, then one can at most represent approximately 2 GHz.

The solution that was arrived at for this problem was to declare two different types for representing power and two different types for representing frequency of a RADAR_SIGNAL. One type would represent the low power range or low frequency range and the other would represent the high power range or high frequency range. For example, consider the definition of an analog signal as :

```
type LOW_FREQUENCY is range 0 to 2E9
```

```
  units Hz;
```

```
    KHz = 1000 Hz;
```

```
  end units;
```

```
type HIGH_FREQUENCY is range 0 to 2E9
```

```
  units MHz;
```

```
    GHz = 1000 MHz;
```

```
  end units;
```

```
type LOW_POWER is range 0 to 2E9
```

```
  units pW;
```

```
    nW = 1000 pW;
```

```
    uW = 1000 nW;
```

```
  end units;
```

```
type HIGH_POWER is range 0 to 2E9
```

```
  units mW;
```

```
    W = 1000 mW;
```

```
    KW = 1000 W;
```

```

        end units;

type ANALOG_SIGNAL is record
    LO_FREQ : LOW_FREQUENCY;
    HI_FREQ : HIGH_FREQUENCY;
    LO_POWER : LOW_POWER;
    HI_POWER : HIGH_POWER;
end record;

```

Once an analog signal is thus defined, we can find the total power in the system as the sum of the power in both the fields, LO_POWER and HIPOWER. Likewise, when the frequency of the signal is needed for the purposes of calculation, we can sum the LO_FREQ and HI_FREQ fields.

The disadvantage of this solution is that simulator overhead is required when converting the low and high range types to real, summing them up, using the intermediate value in calculations, and then converting them back to low and high types.

Though the solution is not an elegant one, it seems to be the only way to solve the problem given the range restriction. Another possibility that was considered was to use a log scale (db) to represent power and frequency. In the case of frequency, this was not possible since the operation of the model required frequencies to be added and subtracted, and this would not be possible if a log scale was used. Power, however can be represented using a db scale, since the model involves only multiplications and divisions with power levels. However, a separate mathematical package would be needed (for the log function) to convert powers to a logarithmic scale. Furthermore, since such logarithmic functions are an approximation that involve summing of a series, this approach

is viewed as inefficient since it would add a greater simulator overhead than the approach that was adopted.

6.3. Problems Posed by VHDL Tools

It was found that simulation run times varied widely depending on the simulator being used, and the machine on which the model was run. Though some difference in simulation times is expected due to simulation tools originating from different vendors, some interesting points were noted about the implementation of these tools, and a brief discussion follows. In particular, two tools were compared on an Apollo DN3500 workstation. These were the **Synopsys VHDL System Simulator Version 2.1c**, and the **MCC CAD VHDL System Version 2.0**.

On finding initially that simulation of the model took a very long time to complete (88 minutes on the **Synopsys** simulator with scan limits of ± 60 degrees, and a maximum detectable range of 100 miles), it was felt that the long run time was due to the simulator spending a considerable amount of time in performing type conversions, and real arithmetic. (Note that for the above values of range, and scan limits, the **RANGE_BIN** signal changes value approximately $41 \times 41 \times \text{MAX_RANGE_BIN}$ times. This works out to 179866. This is the number of times that the target array is scanned, and the number of times that noise is amplified in the receiver). It can thus be appreciated that the model is inherently compute intensive.

However, after tests were run, it was determined that the simulator spent a large part of this simulation time in scheduling events, and monitoring processes and signal values. In order to reduce the run time for the tests, the simulation parameters were changed to

a maximum detectable range of 50 miles, and antenna scan limits of ± 30 degrees. The value of MAX_RANGE_BIN for these parameters is 54. The number of times that RANGE_BIN changes value throughout the simulation in this case then is 23761. (The simulation then takes 11 minutes and 50 seconds to run to completion with the same simulator. This confirms that the run time is approximately proportional to the number of times that signal RANGE_BIN changes value.

A simple test was written to determine the amount of time the simulator spends in scheduling events, and monitor processes and signal values. The code for the test example appears below :

```
use work.all;
entity TEST is
port (A : in INTEGER := 0; B : inout INTEGER := 0; C : in bit);
end TEST;
```

```
use work.all;
architecture TEST of TEST is
begin
process
begin
if C = '1' and not C'STABLE then
for i in 1 to A
loop
```



```

    If B < 54 then
--      B <= B + 1;
        wait for 10 us;
    else
--      B <= 0;
        wait for 10 us;
    end if;
end loop;
end if;
wait on C;
end process;
end TEST;

use work.all;
entity TEST_BENCH is
end TEST_BENCH;

architecture T of TEST_BENCH is
    signal A, B : INTEGER := 0;
    signal C : bit;

    component TEST_THIS
    port (A : in INTEGER := 0; B : inout INTEGER := 0; C : in bit);
    end component;

    for L1 : TEST_THIS use entity TEST(TEST);

```

```

begin
L1 : TEST_THIS
port map (A, B, C);

A <= 23761;
C <= transport '1' after 1 ns;
end T;

```

This test example simply reads in the value of a signal A of type INTEGER and on assertion of signal C, a loop is entered whose body is executed A times, or in this case, 23761 times. Note that all that the body of the loop contains is a wait statement, “wait for 10 us”. The integer add is commented out, and does not take place. (It is “decommented” in a following test to determine how much time the simulation takes, if an integer add is inserted in the body of the loop). Surprisingly, it was found that the scheduling of events due to the wait statement takes a relatively long time to accomplish. (Note that there is a similar situation in the radar system model, where process RANGE_INCREMENT increments the value of RANGE_BIN after every PULSE_ON_TIME ns). In particular, it takes 29 seconds for the **Synopsys** simulator to execute the test model, but it takes 145 seconds for the **MCC** simulator to execute the test. After the integer add statements were “decommented”, the simulator from **Synopsys** ran the model in 38 seconds, whereas the **MCC** simulator ran the model in 153.7 seconds. This implies that the time spent in performing the integer additions is about the same for both simulators.

Furthermore, on changing the wait statement, and making the model wait for 10 ns (as opposed to 10 us) in the body of the loop, the run time changed dramatically. Using the **Synopsys** simulator, the run time reduced to 21 seconds with the integer additions “de-commented” (a reduction of 45 per cent), but the run time reduced to 24.7 seconds for the **MCC** simulator (a reduction in run time of 84 per cent).

However, on changing the `PULSE_ON_TIME` in the radar system model from 10 us to 10 ns, a minimal change was observed in the run time using the **Synopsys** simulator, but run time reduced by a significant amount using the **MCC** simulator. In particular, the **MCC** simulator took 19 minutes, and 40 seconds to run the model with a `PULSE_ON_TIME` of 10 us, but took 11 minutes, and 24 seconds to run with a `PULSE_ON_TIME` of 10 ns (a reduction in run time of 42 per cent). On the other hand, the **Synopsys** simulator took 44 minutes and 52 seconds to run the model with a `PULSE_ON_TIME` of 10 us, but took 44 minutes and 31 seconds to run the model with a `PULSE_ON_TIME` of 10 ns (a reduction in run time of just 0.8 per cent).

It is thus felt that a significant part of the run time is spent in scheduling of events, and not all of it is attributed to type conversions and arithmetic. Furthermore, VHDL tools vary as far as implementation of the scheduling of events is concerned, and one should first make comparisons before determining which tool to use to run simulations of VHDL models, so as to minimize run time.

Chapter 7. A Fault Diagnosis Methodology

7.1. Introduction

As the follow-on part of this research, it is proposed to develop a fault diagnosis methodology for locating faults at the system level from *first principles* [2] using knowledge of the behavior of the system. In future research work, it is hoped that this fault diagnosis methodology is adopted, and a reasoning system is built that uses VHDL Behavioral descriptions to perform diagnostic reasoning. Fault diagnosis systems of this type have been built in the past for digital systems, and one such system is due to Marcotte, Neiberg, Piazza, and Holtzblatt of the MITRE Corp. [6].

A method of reasoning from *first principles* [2] is required, since we need to reason from the behavior and structure of the system and its components. Once it is determined that the system is misbehaving (or is not behaving as it was intended to) then we need some method of using the VHDL model and localizing the fault to an entity or entities that seem most likely to be responsible. In real applications, it is intended that the model

and its fault diagnosis reasoning tool are used to diagnose faults in the actual RF system which is mis-behaving.

Work exists in the literature that uses techniques of artificial intelligence to perform diagnostic reasoning based on structure and behavior. Work in this area by Dr. Randall Davis [2] of the Massachusetts Institute of Technology, (mostly performed in the digital domain) is particularly interesting. This chapter borrows from much of that work, and it is proposed to apply some of these techniques for fault diagnosis of RF systems. By fault diagnosis is meant the localizing of faults to certain entities that are determined to be potentially responsible for the faulty behavior; and not the generation of test vectors that will detect the fault. In an actual RF system, this diagnosis methodology will help in determining which Line Replaceable Units (LRUs) to replace in a malfunctioning radar system (say), which would help in bringing up the system in a very short time.

Specifically, methods of *discrepancy detection* and *constraint suspension* [2], first discussed by R. Davis, are proposed to be used that make use of the structural and behavioral information about a system, provided by VHDL descriptions. These methods are elaborated upon later on in this chapter.

We have then, a VHDL model of an RF system, and we also have available the faulty symptoms of the real RF system, which reportedly is malfunctioning. We have to use this information to try and locate the fault to within an entity or an LRU, so that the system may be brought back up with a minimum of delay, by replacing the suspected component(s). An essential aspect to consider when attempting to use these techniques for diagnostic reasoning is the *paths of causal interaction* [2] to consider between these various components or entities that will be held accountable for the fault. That is, we need to ask "How are the different entities related to each other ?", so we may determine

the effect of one over the others. This is an important question to ask, since we are attempting to track down the fault to some component(s), using *first principles* and knowledge of behavior, and not by using some fault model or previously encountered fault data base. These latter methods become cumbersome and very time-consuming as systems grow large, and it is proposed that these methods be used at a later stage, after the fault has been narrowed down to within a few entities to further narrow down the search, if necessary.

One obvious path of *causal interaction* is that provided by the structural description of the entire system, which in turn provides information as to how the entities are connected together. This information is inherent in VHDL descriptions, and makes up *functional adjacency*. By *functional adjacency* we mean the adjacency that is provided in a VHDL model by the signal interconnect information. For systems that we are modeling though, it becomes important to consider RF effects, temperature effects, shielding effects, etc., which are *proximity effects*. By *proximity effects* we mean the effect an entity could have over some other entity because they are physically close together.

Fortunately, VHDL has the potential to allow one to include this information in the descriptions of entities by the use of user defined *attributes* [4]. Once knowledge of the physical proximity of these various entities in a real system are obtained, VHDL *attributes* can be used which allow one to specify the proximity of one entity to another.

For example, for boxes (sub-units or LRUs within the real system), knowledge of thermal adjacency is important since these boxes or units can transfer heat between them. In a real system, cooling methods may be incorporated externally that take heat away from these boxes in order to prevent high temperature effects. Once this knowledge about thermal adjacency and cooling effects in the real system is obtained, attributes can

be defined that will specify for each entity (according to the box in which it lies), its susceptibility to temperature effects from all the other entities. An additional attribute can be defined that specifies for each entity, whether the box in which it lies has some cooling mechanism, since this will allow for modeling of the failure of the cooling mechanism as well.

An example of how temperature susceptibility information could be extracted from the model is now discussed. After knowledge of the real system is obtained, one associates with the definition of each entity two attributes, its `ID_NUMBER` and its `TEMP_VECTOR`. `ID_NUMBER` takes the form of an integer number which is unique to that entity. This number identifies the entity, and the attribute could be called `ID_NUMBER`. The second attribute, `TEMP_VECTOR` takes the form of a bit vector, and it can specify the susceptibility of the entity to temperature effects from other entities.

All the entities that make up the system are then numbered from 1 through n , and thus uniquely identified by the `ID_NUMBER` (assuming there are n entities in all). The `TEMP_VECTOR` would then be n bits long and could completely specify the temperature susceptibility of the entity to all the other entities.

Consider for example that it is needed to determine the susceptibility of entity 'p' to temperature effects from entity 'q'. Where 'p' and 'q' are `ID_NUMBERS`. This can be achieved by looking up the `TEMP_VECTOR` of entity 'p', and referring to the element in it that is indexed by the value of 'q'. If this value is a '0', 'p' is not susceptible to temperature effects from 'q'. If it is a '1', then 'p' is susceptible to temperature effects from 'q'.

In order to do this it is required to first define these TEMP_VECTORs and assign values to them in the VHDL model. This is done by first determining from the knowledge of the real system, the temperature susceptibilities of the entities to each other. For example, one way to do this would be to assume that if entity 'q' lies in the same box as entity 'p', then TEMP_VECTOR[q] of entity 'p' = '1'. Again, if entity 'q' lies in an immediately adjacent box, and is not thermally insulated from it, or cooled, then again TEMP_VECTOR[q] of entity 'p' = '1'. On the other hand, if entity 'q' lies in a box that is some distance away, or is not in the immediately adjoining box to that of entity 'p', or it is thermally insulated, then TEMP_VECTOR[q] of entity 'p' = '0'.

Consider as an example that we have four entities ENTITY1, ENTITY2, ENTITY3, and ENTITY4. Then, the entity declaration of ENTITY1 would be as under :

```
package ATTRIBUTE_DEFS is
  attribute ID_NUMBER : INTEGER;
  attribute TEMP_VECTOR : BIT_VECTOR(1 to 4);
end ATTRIBUTE_DEFS;

entity ENTITY1 is
  port (...);
  generic (...);
  ...
end ENTITY1;
```


After the entity is thus defined, attribute specification can take place in the structural architecture in which the ENTITY1 is used. For example consider architecture STRUCTURAL of an entity EXAMPLE in which ENTITY1 is instantiated.

```
use WORK.ATTRIBUTE_DEFS.ALL;
entity EXAMPLE is
end EXAMPLE;

architecture STRUCTURAL of EXAMPLE is
signal ...
....
signal ...
component ENTITY1 is
port (...);
generic (...);
end component;
....
for L1 : ENTITY1 use ENTITY1(architecture_name);
attribute ID_NUMBER of ENTITY1 is 1;
attribute TEMP_VECTOR of ENTITY1 is (0100);
.....
```

The attributes specifications above indicate that ENTITY1 is identified as 1, and it is susceptible to temperature effects from entity 2 only, since TEMP_VECTOR[2] of 1 = '1'.

Attributes may again be defined which specify the susceptibility of the system to RFI. Whereas temperature effects are accounted for due to the proximity of boxes, RF effects can be accounted for due to the proximity of wires. It is then possible by a similar method to model for RF effects of one signal over another. This can occur for example if two signals share the same cable.

Consider for example that there are n signals associated with the top level entity. In the architecture description of the top level entity where all the signals to be used in the top level entity are declared, it is possible to again associate with each signal two attributes, a SIG_ID (1 through n), and an RFI_VECTOR (n bits long). If it is possible to have RFI between signal 'i' and signal 'j', then RFI_VECTOR[i] of signal 'j' is '1', and RFI_VECTOR[j] of signal 'i' is also '1'. The attribute declarations for the RFI case are similar to that for the temperature case, but are associated with the signals, rather than the entities.

Note that by changing the type of the TEMP_VECTOR or RFI_VECTOR from BIT_VECTOR to an array of integers, it is possible to scale the susceptibility of entities to RFI or temperature with respect to some maximum. This is in contrast to using only '1's and '0's to represent susceptibility, which might be viewed as a weak system of representation, if accurate data about RFI between signals, or temperature effects between boxes is available. Hence it becomes possible to more accurately represent temperature and RFI data to the diagnostic system.

7.2. Hierarchy of Paths of Interaction

A hierarchy of types of *paths of causal interaction* needs to be considered in order to consider a broad range of faults. That which is higher up in the hierarchy is that which is most likely to yield candidate information, and that which is lower on is only resorted to if the higher one fails to yield a candidate. It is natural then to consider the *functional adjacency* [2] first, as this is most likely to yield a candidate, whose symptoms of malfunction are realistic. If this fails, then we may look at proximity effects (temperature, RFI etc.)

7.3. Discrepancy Detection & Constraint Suspension

The methods of *discrepancy detection and constraint suspension* will now be defined and elaborated upon [2]. Each sub-system or entity that we consider as a candidate has a set of constraints associated with it which are complete in describing the behavior of that sub-system. That is, given these constraints, we can use them to figure the outputs of the sub-system for any combination of inputs. Furthermore, given the outputs, we can figure out what the inputs should have been. This process of back propagating through the model (figuring out the inputs or values at certain nodes given the outputs) is a difficult process and proves to be cumbersome for fault diagnosis of large digital systems. Fortunately, behavioral modeling of RF systems (like the radar system that we modeled), involve sub-systems whose behavior can be represented using some simple mathematical function (subtraction, addition, multiplication, division, etc.), usually on one

or two variables (as compared to digital systems, where the entire truth table need be considered). This makes the process of back propagating easier.

Given these constraints, we can model the system as a network made up of connected constraints. A VHDL model provides information about structure and behavior. From this, a network of constraints can be extracted. The behavioral descriptions provide information about the constraints themselves, and the structural hierarchy can provide information about the way in which these networks are connected.

Constraint suspension asks : Is there some constraint, the suspension of which will leave the network in a consistent state ? That is, we look for global consistency. If global consistency is found, each such constraint accounts for all the observed symptoms. The implication here is that the failure of the sub-system whose constraint(s) are suspended, explains all the observed symptoms. We assume here a single point of failure; where "point" here refers to a sub-system. But which constraint do we look to suspend ? Logically, any constraint lying along a path from an incorrect output to an input can be responsible for incorrect behavior. So, we need only consider those constraints that are on a path that lie from an incorrect output to an input. We can thus create *dependency chains* that trace outputs to inputs. This information is provided by VHDL and can be extracted from the various entity declarations of the VHDL model. Each constraint (or sub-system) lying on a chain from a defective output to an input (chain here refers to all those sub-systems that are interconnected via the path of causal interaction under consideration), can conceivably be responsible for the fault.

This is a good strategy since we simply assume that we know nothing about the correct functioning of the sub-system (or how it is supposed to function or what its behavior is), but assume that it is functioning in an incorrect manner; then we try and determine if

this faulty sub-system alone can explain all the discrepancies (as well as the good outputs) assuming all the other sub-systems are not faulty; i.e. we look for consistency.

We need a way of generating candidates that may be responsible for the faulty condition that we come across. The idea here is to exonerate those that clearly cannot be responsible for the faulty condition. Once multiple candidates are generated by considering one path of interaction, we can slip down one level of hierarchy in the structure of the system (using the VHDL description), and try and determine for each of these candidates, whether the sub-components of the candidate are likely to be responsible for the fault, using here the same techniques as for the parent component. If given the inputs and outputs of the parent candidate, the sub-components cannot interact in any way to produce that fault, that parent candidate can then be exonerated, and another candidate is considered.

So, in effect, after *constraint suspension* is performed at the top most level of the structural hierarchy, we move down one level in the hierarchy and try to determine from *first principles* (this time using the behavioral description of the parent candidate, and performing *constraint suspension* on the sub-components of the parent candidate) if these sub-systems can be globally consistent in explaining the parent candidate's symptoms. (where "symptoms" refer to the value on its ports that were obtained while checking for consistency of the parent candidate)

Thus, three things need to be done to generate candidates. First, simulate the system and collect discrepancies between predicted outputs and actual outputs. This is the *discrepancy detection* stage. Second, determine potential candidates that could be responsible by considering the dependency chain from the faulty output to the inputs. Third,

for each of these candidates, perform *constraint suspension* to determine if they are globally consistent. If they are globally consistent, then they are likely candidates.

An algorithmic approach is presented below. Once, the path of causal interaction to be considered has been selected, the constraint network is extracted from the VHDL model, and the following algorithm is performed.

1. Step 1.

- A) Simulate the VHDL model by providing primary inputs and collect all discrepancies in outputs. That is, find out all those outputs of the simulation that differ from the actual outputs of the system.

2. Step 2.

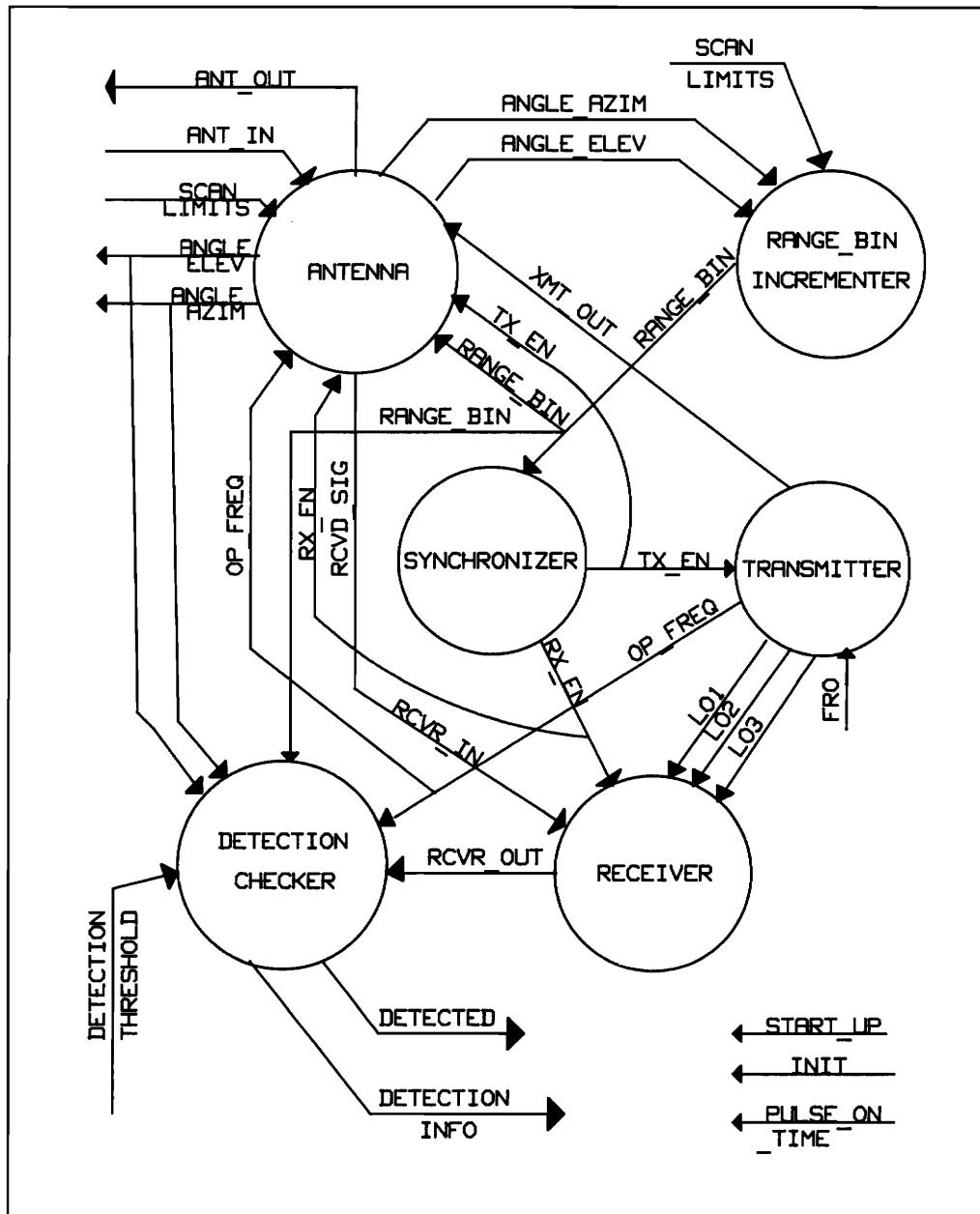
- A) For each of these outputs, determine the dependance chain for that faulty output. The sub-systems or entities lying on the dependency chain are potential candidates.
- B) Take the intersection of all the candidates obtained after considering all the discrepancies. The common ones are most likely to be at fault and to explain all the discrepancies.

3. Step 3. For each of these candidates, we need to determine global consistency. Consider for this purpose, the constraint network of the model.

- A) Select a candidate, if any, from the list of possible ones.

- B) Suspend the constraints that model that candidate's behavior. Turn on all other constraints.
- C) Apply the primary inputs to the constraint network, and also apply the "observed outputs" from the VHDL simulation at the outputs of the constraint network. By back propagating, and forward implication, determine whether the suspension of the candidate's constraints can explain all the observed outputs. If yes, then this process of back propagating of outputs and forward implication of inputs will provide a set of values on the ports of the suspected candidate. These are its "symptoms", and explain what the candidate may be doing. (These can, at a later stage, be used to go one level down in the candidate's hierarchy, whose sub-components may then be examined in a similar manner for consistency using these values or "symptoms") Add this candidate to the list of globally consistent candidates, and delete it from the list of possible candidates.
- D) If (C) does not lead to global consistency for the candidate, then abandon that candidate, and remove it from the list of possible candidates, since it cannot be held responsible for the observed outputs.
- E) Return to Step 3, part (A).

If the algorithm does not yield any consistent candidate, considering the current path of causal interaction, we can slip down one level in the hierarchy of paths of interaction to be considered, try another path of causal interaction, and extract the constraint network using this current path to determine if we can find a likely candidate here. This process is continued till a candidate is found, or till all the *paths of causal interaction* to be considered are exhausted.



7.4. A Diagnosis Example

In order to demonstrate how this methodology works in localizing faults in a high level behavioral model of an RF system, consider Figure 5 on page 96 which shows the pictorial representation of a subset of the model that was written for the radar system. It shows the major signals involved, indicates the analog_signal data path, and the control signals through the system. Note that the primary inputs to this system are START_UP, DETECTION_THRESHOLD, FRO, ANTENNA_SCAN_LIMITS, ANT_IN, INIT, and PULSE_ON_TIME. The primary outputs are ANGLE_ELEV, ANGLE_AZIM, ANT_OUT, DETECTED and DETECTION_INFO. The VHDL code for all these sub-systems can be found in the main code for the radar system which is in the appendix.

A brief explanation as to the working of this model is now given in order to understand the behavior of the system, which will aid in the fault diagnosis part. Upon START_UP, initialization of ANTENNA_SCAN_LIMITS (also used to position the antenna at START_UP) takes place, and initialization of DETECTION_THRESHOLD is achieved. Then, entity SYNCHRONIZER asserts TX_EN which signals the TRANSMITTER to transmit a high power analog signal (15 KW, 5700 MHz) to the ANTENNA unit.

The ANTENNA unit directs this signal out into the TARGET_ENVIRONMENT upon receipt of the TX_EN signal to a particular direction (initially to that specified by SCAN_LIMITS); that are specified by ANGLE_AZIM and ANGLE_ELEV. ANT_OUT is the signal sent out into the environment.

TX_EN is then de-asserted, and RX_EN is asserted; which in turn signals the RANGE_INCREMENTER to begin incrementing RANGE_BIN up from zero to RANGE_BIN_LIMIT at regular intervals. Meanwhile, RCVD_SIG is assigned a returned signal if a target is present in the current beam position, and if the return from it falls within the current RANGE_BIN. Otherwise, it is assigned (0 KW, 0 MHz).

The received signal RCVD_SIG is processed in the receiver, i.e. it is down converted (mixed with LO1, LO2, LO3) and amplified (3 stages), and passed on to the DETECTION_CHECKER, where the DETECTION_CHECKER checks to see if RCVR_OUT exceeds the threshold. If it does, it outputs information into the signal DETECTION_INFO about the target's range, its velocity, and its position (Azimuth and Elevation). At the same time, it asserts DETECTED, to inform of a target detection.

After one receive cycle is thus completed (specified by one complete cycle of the RANGE_BIN signal from 0 to RANGE_BIN_LIMIT), the ANTENNA advances by 2 degrees in AZIMUTH, and goes through the transmit and receive cycles again. This process continues till an entire scan of the environment is completed.

To model a fault scenario, suppose that a target that should have been detected, is not detected. That is, suppose that there exists just one target in the TARGET_ENVIRONMENT and of the various primary outputs, we find that signal DETECTED never goes high throughout the entire scan, and DETECTION_INFO does not provide information relating to a target detection. (DETECTION_INFO provides information about the target and its information is only updated, when a DETECTION is sensed. At all other times, its output is not valid, and stays at 0 degrees, 0 degrees, 0 miles, OPENING, 0 m/s). The other primary outputs ANGLE_ELEV,

ANGLE_AZIM, and ANT_OUT function properly. In other words, the only discrepancy in operation is sensed at the primary outputs DETECTED, and DETECTION_INFO.

Suppose that the target that should have been detected is present at -59 degrees Azimuth, 59 degrees Elevation, is 650 ns of round trip range time away, provides an attenuation of $4.5E+11$, and a Doppler shift of -2200 Hz. According to proper operation of the system (provided by the simulation), it is predicted that the target should be detected when the center of the beam is at -60 degrees Azimuth, 60 degrees Elevation, and it should be detected in RANGE_BIN 65.

As mentioned earlier, it is found that the only primary outputs where a discrepancy is found is signal DETECTED, and signal DETECTION_INFO. All other primary outputs function as predicted. We will now apply the above fault diagnosis methodology to the model using discrepancy detection and *constraint suspension*, to try and diagnose the fault.

Suppose that we have a VHDL model that specifies the Behavior and Structure of the system as explained above and as specified by the process model graph (refer to the model that was written for the radar system). What we then need is a program that uses this VHDL model to extract information about structure from the model (assuming functional adjacency), and then for each entity or sub-unit it finds, it forms a set of constraints using the VHDL behavioral descriptions. One such program (GMODS) has been written and tested by the MITRE corporation for digital circuit descriptions. See [2] for details. Once the constraints associated with the sub-systems are obtained, a constraint network is formed using information about the structure (connectivity) of the system (for a path of causal interaction corresponding to *functional adjacency*). For ex-

ample, the constraints for the Transmitter (referring to the VHDL model) could look like this :

TRANSMITTER_CONSTRAINTS :

Begin

IN : FRO.

OUT : OP_FREQ, LO1, LO2, LO3, XMT_OUT.

OP_FREQ = LO1 + LO2 + LO3 (TX_EN'DELAYED = 1).

XMT_OUT.FREQ = OP_FREQ (TX_EN'DELAYED = 1).

XMT_OUT.POWER = 15 KW (TX_EN'DELAYED = 1).

LO1 = 48 * FRO (TX_EN = 1).

LO2 = 8 * FRO (TX_EN = 1).

LO3 = 1 * FRO (TX_EN = 1).

End TRANSMITTER_CONSTRAINTS.

The constraints for the other sub systems can be found in Appendix B. Looking at the algorithm for fault diagnosis of this system, we see that the first step is to collect discrepancies. The only discrepancies that are found are on signals DETECTED, and DETECTION_INFO.

Next (Step 2), we have to follow the dependency chain back to the inputs for each of these outputs. Here, we find that each and every sub-system of the graph is part of the dependency chain for both the outputs. So, after taking the intersection of the sets, we find that we have to perform *constraint suspension* on each and every sub-system in order to determine global consistency, i.e. in order to determine if that sub-system could indeed

be responsible for all the symptoms (faulty and good) on its own. So, the candidates are :

1. Synchronizer
2. Transmitter
3. Antenna
4. Range_Bin_Incrementer
5. Receiver
6. Detection_Checker

Constraint Suspension on SYNCHRONIZER :

Turning off the constraints on SYNCHRONIZER alone, we have to see if there are any set of values on the ports of SYNCHRONIZER that can account for all the observed outputs (good and faulty). If so, then these values give an idea as to what the possible “symptoms” of the failed SYNCHRONIZER could be.

Since a DETECTION is never made, a value of RCVR_OUT = (0 uW, X Hz) can be assumed (by back propagating through the DETECTION_CHECKER) ; where ‘X’ implies a ‘don’t care’ state. This, when propagated back through the RECEIVER, gives a value of RCVR_IN (input to the receiver) = (0 pW, X MHz). Since ANT_OUT = (15 KW, 5700 MHz), a value of ANT_IN = (333 pW, 5700 MHz - 2200 Hz) is obtained (knowing the characteristics of the target). So, then considering the ports of SYNCHRONIZER, RX_EN = 0 (when it should be 1), TX_EN = 0, RANGE_BIN

= 65 can account for all the observed symptoms, so this makes SYNCHRONIZER a globally consistent candidate.

Constraint Suspension on TRANSMITTER :

Are there any values on the ports of TRANSMITTER that will justify all the observed outputs (good and faulty) ? If yes, then TRANSMITTER alone could be responsible for all the observed outputs. And if so, then these values are the possible “symptoms” of failure of the TRANSMITTER.

Since ANT_OUT = (15 KW, 5700 MHz) during the transmit phase and (0 KW, 0 MHz) in the receive phase, back propagating through the ANTENNA, we find that XMT_OUT must be (15 KW, 5700 MHz) and (0 KW, 0 MHz) respectively. Also, back propagating through DETECTION_CHECKER, RECEIVER, we see that LO1, LO2, LO3, OP_FREQ = X, since any arbitrary values on these do not explain the missed detection. No assignment of values on the ports of TRANSMITTER could be found, so TRANSMITTER is not a globally consistent candidate.

Constraint Suspension on ANTENNA : Suspending the constraints on ANTENNA alone, can we place some value on each of the ports of ANTENNA that will explain the malfunctioning of the entire system, assuming that ANTENNA alone is at fault ? Suppose that primary input SCAN_LIMITS = +/- 60 degrees. Since ANT_OUT (primary output) is (15 KW, 5700 MHz), ANT_IN must be = (333 pW, 5700 MHz - 2200 Hz) (for this target) when ANGLE_AZIM = -60 degrees, ANGLE_ELEV = 60 degrees, RANGE_BIN = 65, TX_EN = 0, RX_EN = 1. Looking at the ports of ANTENNA, we see that the outputs ANGLE_ELEV, ANGLE_AZIM, ANT_OUT, ANT_IN are as expected. The only other port that could explain the discrepancy is RCVD_SIG. Back propagating through the DETECTION_CHECKER and the RECEIVER, we see that

a value of RCVD_SIG = (0 KW, X MHz), when it should have been (333 pW, 5700 MHz - 2200 Hz), would explain all the observed symptoms of the system. Hence, ANTENNA is a likely candidate, and is globally consistent.

Constraint Suspension on RANGE_BIN_INCREMENTER :

Using the observed outputs of the system, and given the inputs to the system, is it possible to obtain a set of values (at the ports of RANGE_BIN_INCREMENTER) by forward and backward propagating through the constraints of the other entities ? If so, then constraint suspension yields a likely candidate that accounts for all the symptoms.

Since all other entities are assumed to work right, back propagating through them gives a value of TX_EN = 0, and RX_EN = 1 during the receive cycle, when ANGLE_ELEV = 60 degrees, ANGLE_AZIM = -60 degrees. The output port RANGE_BIN determines whether the return from the target is assigned to ANT_IN. So, back propagating through the ANTENNA, we see that since ANGLE_ELEV and ANGLE_AZIM are as predicted, and these depend on the successful completion of the RANGE_BIN cycle, there is no assignment to the RANGE_BIN port which explains all the observed outputs. We have considered all the ports of RANGE_BIN_INCREMENTER and there is no assignment of values to them out of the ordinary that can explain all the discrepancies, and so RANGE_BIN_INCREMENTER is not globally consistent.

Constraint Suspension on RECEIVER :

Likewise, does *constraint suspension* on RECEIVER prove RECEIVER to be a globally consistent candidate ? If so, the symptoms of the malfunctioning RECEIVER will be available at its ports. Back propagating through the DETECTION_CHECKER, we see that RCVR_OUT.POWER < DETECTION_THRESHOLD. Then, RCVR_OUT = (0 KW, X MHz), LO1 = 4800 MHz, LO2 = 800 MHz, LO3 = 100 MHz, RCVD_SIG

= (167 uW, 5700 MHz - 2200 Hz) are the "symptoms" of the malfunctioning RECEIVER. So, RECEIVER is a globally consistent candidate.

Constraint suspension on DETECTION_CHECKER : Can DETECTION_CHECKER alone explain the malfunctioning of the system ? Is there any assignment of values to the ports of DETECTION_CHECKER that makes the constraint network consistent ? Since RCVR_OUT = (167 uW, 2200 Hz) (output from RECEIVER), we see that RANGE_BIN = 'X', DETECTED = '0', DETECTION_INFO = (0 degrees, 0 degrees, 0 miles, OPENING, 0 m/s) will explain the malfunctioning. So, an assignment of values to the ports (out of the ordinary) have been found and DETECTION_CHECKER is a globally consistent candidate.

Thus, after having run the algorithm, we find that the possible entities that could be responsible for the malfunctioning of the system are :

1. SYNCHRONIZER
2. ANTENNA
3. RECEIVER
4. DETECTION_CHECKER

Hence, use of *constraint suspension*, and the use of *functional adjacency* as the path of causal interaction, has exonerated two candidates TRANSMITTER, and RANGE_BIN_INCREMENTER. It should be noted that this fault (where a detection is missed), is a very general fault and there can obviously be many reasons for it. Con-

sidering this, it does seem significant that two of the six candidates were exonerated by reasoning from *first principles*.

However, note that the process can be repeated on each of these four likely candidates (using the “symptoms” available at its ports as primary inputs and outputs for the sub-systems of the candidate) by going down one level in the structural hierarchy and performing *constraint suspension* on the sub-systems. If this process fails to prove any of the sub-systems of a parent candidate as a globally consistent candidate, then the parent candidate is exonerated, since there is no way it can account for the observed symptoms present at its ports. This process may further reduce the number of possible faulty components or Line Replaceable Units.

Though the method seems natural, what is required of future research in this area is a process of extraction of information about structure and behavior from VHDL models to form constraint networks on which the algorithm can be run. Clearly, this is not a simple task and it is hoped that future research efforts will concentrate in the direction of using this methodology to automate the process using VHDL models of RF systems.

Chapter 8 : Conclusions

8.1. Conclusions

A methodology for modeling the behavior of RF systems using the VHSIC Hardware description language was developed, and a representative RF system - a pulsed Doppler radar system in particular, was modeled successfully using this methodology. The methodology is general enough and can be applied to any other RF systems.

A methodology for fault diagnosis of the radar system using the VHDL model was suggested, and a fault diagnosis example was presented. It is hoped that this will aid future research efforts in developing an automated tool that will extract diagnosis information from these VHDL models, and automate the process of fault diagnosis of RF systems.

Bibliography

1. James R. Armstrong, "Chip Level Modeling with VHDL," Prentice Hall, New Jersey, 1989.
2. Davis Randall, "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence*, Vol 24, pp.. 347-410.
3. General Dynamics Corporation, "Fire Control Radar Training Manual (General Information), "
4. "IEEE Standard VHDL Language Reference Manual ," IEEE, New York, 1988.
5. Lipsett R, Schaefer C. F., Ussery C., "VHDL: Hardware Description and Design, " Kluwer Academic Publishers, Boston, 1989.
6. Marcotte R. A., Neiberg M.J., Piazza R.L., Holtzblatt L.J., "Using VHDL Models to Diagnose Faults within Digital Systems, " The MITRE Corporation, 1990.
7. Morris G.V., "Airborne Pulsed Doppler Radar, " Artech House, MA., 1988.
8. Skolnik M.I., "Introduction to Radar Systems, " McGraw Hill Inc., 1980.
9. Stimson G.W. , "Introduction to Airborne Radar, " Hughes Aircraft Co., Calif., 1983.
10. " VHDL Spring 1991 Users Group Conference Proceedings "

Appendix A. The Package Body

The package body of the package that was defined in chapter 4 appears below. Here all the procedures and functions that were defined in the package statement are expanded upon.

The package body "RADAR" :

```
use work.all;
use std.TEXTIO.all;
package body RADAR is

function MAX_RANGE_BIN (PULSE_ON_TIME : TIME;
                        MAX_DET_RANGE : REAL)
    return NATURAL is

variable TEMP1 : NATURAL := 0;
variable TEMP2 : REAL := 0.0;
begin
    if not (PULSE_ON_TIME = 0 ns) then
        TEMP2 := (MAX_DET_RANGE*10666.7)/
            (TIME_TO_REAL_IN_NS(PULSE_ON_TIME));
    end if;
    loop
        If TEMP2 > 0.0 then
            TEMP1 := TEMP1 + 1;
            TEMP2 := TEMP2 - 1.0;
        else
            exit;
        end if;
    end loop;
    return TEMP1;
end MAX_RANGE_BIN;

function TIME_TO_REAL_IN_NS (A : TIME) return REAL is
variable RETURN_THIS : REAL := 1.0;
variable TEMP : INTEGER := 1;
begin
    TEMP := A/1 ns;
    RETURN_THIS := REAL(TEMP);
```

```

    return RETURN_THIS;
end TIME_TO_REAL_IN_NS;

```

```

function LOFREQ_TO_REAL_IN_Hz (A : LOW_FREQUENCY) return REAL is
variable TEMP : INTEGER := 0;
variable RETURN_THIS : REAL := 0.0;
begin
    TEMP := A/1 Hz;
    RETURN_THIS := REAL(TEMP);
    return RETURN_THIS;
end LOFREQ_TO_REAL_IN_Hz;

```

```

function HIFREQ_TO_REAL_IN_MHz (A : HIGH_FREQUENCY)
    return REAL is
variable TEMP : INTEGER := 0;
variable RETURN_THIS : REAL := 0.0;
begin
    TEMP := A/1 MHz;
    RETURN_THIS := REAL(TEMP);
    return RETURN_THIS;
end HIFREQ_TO_REAL_IN_MHz;

```

```

function ANGLE_TO_REAL_IN_DEG (A : ANGLE) return REAL is
variable TEMP : INTEGER := 0;
variable RETURN_THIS : REAL := 0.0;
begin
    TEMP := A/1 degrees;
    RETURN_THIS := REAL(TEMP);
    return RETURN_THIS;
end ANGLE_TO_REAL_IN_DEG;

```

```

function BIN_DISTANCE (A : TIME) return REAL is
variable RETURN_THIS : REAL := 0.0;
begin
    RETURN_THIS := TIME_TO_REAL_IN_NS(A)/10667.0;
    return RETURN_THIS; -- in miles.
end BIN_DISTANCE;

```

```

procedure SCAN_ADVANCE (signal AZIM, ELEV : in ANGLE;
    signal ELEV_RANGE, AZIM_RANGE : in ANGLE;
    signal AZIM_1, ELEV_1 : out ANGLE) is
-- Assumes a 3 degree beamwidth in Azimuth and Elevation and
-- advances scan beam by 3 degrees in Azimuth and elevation
-- Process keeps repeating until scan mode is changed.

```

```

Begin
    If (AZIM_RANGE - 2 degrees) > AZIM then

```

```

    AZIM_1 <= (AZIM + 3 degrees);
    ELEV_1 <= ELEV;
    elsif
    0 degrees -(ELEV_RANGE - 2 degrees) < ELEV then
    ELEV_1 <= ELEV - 3 degrees;
    AZIM_1 <= 0 degrees -(AZIM_RANGE);
    else
    AZIM_1 <= 0 degrees - (AZIM_RANGE);
    ELEV_1 <= ELEV_RANGE;
    end if;
end SCAN_ADVANCE;

procedure INCREMENT_RANGE_BIN (signal RANGE_BIN : in NATURAL;
    signal RANGE_BIN_2 : out NATURAL;
    signal RANGE_BIN_LIMIT : in NATURAL) is
begin
    If RANGE_BIN = RANGE_BIN_LIMIT then
        RANGE_BIN_2 <= 0;
    else
        RANGE_BIN_2 <= RANGE_BIN + 1;
    end if;
end INCREMENT_RANGE_BIN;

procedure READ_TARGET_ENVIRONMENT (signal TARGET_MAP : out
    TARGET_ENVIRONMENT;
    signal NUMBER_TARGETS : out INTEGER) is
variable COUNT : INTEGER := 1;
variable TEMP_TIME : TIME;
variable TEMP_ANGLE : ANGLE;
variable TEMP_REAL : REAL;
variable TEMP_INTEGER : INTEGER;
variable L : LINE;
variable FILENAME : STRING(1 to 8);
file INFILE : text is in "TARGETS.";
begin
    Readline (I,L);
    Read (L, FILENAME);
    loop
        If not endfile(INFILE) then
            Readline (INFILE, L);
            Read (L, TEMP_INTEGER);
            TARGET_MAP(COUNT).AZIMUTH <= TEMP_INTEGER*1 degrees;
            Readline (INFILE, L);
            Read (L, TEMP_INTEGER);
            TARGET_MAP(COUNT).ELEVATION <= TEMP_INTEGER*1 degrees;
            Readline (INFILE, L);
            Read (L, TEMP_TIME);
            TARGET_MAP(COUNT).TIME_AWAY <= TEMP_TIME; -- in us.
            Readline (INFILE, L);

```

```

    Read (L, TEMP_REAL);
    TARGET_MAP(COUNT).ATTENUATION <= TEMP_REAL;
    Readline (INFILE, L);
    Read (L, TEMP_INTEGER);
    TARGET_MAP(COUNT).TARGET_DOPPLER <= TEMP_INTEGER*1 Hz;
    COUNT := COUNT + 1;
  else exit;
  end if;
end loop;
NUMBER_TARGETS <= COUNT - 1;
end READ_TARGET_ENVIRONMENT;

procedure WRITE_TARGET (signal TARGET_DOPPLER :
    in LOW_FREQUENCY;
    signal ANGLE_ELEV,
    ANGLE_AZIM : in ANGLE;
    signal PULSE_ON_TIME : in TIME;
    signal RANGE_BIN : in NATURAL;
    signal OP_FREQ : in HIGH_FREQUENCY;
    signal TARGET_INFO : out DETECTIONS;
    signal DETECTED : out BIT) is
  variable COUNT: INTEGER := 0;
  variable TEMP_REAL : REAL := 0.0;
  variable TEMP : DETECTIONS;
  variable TEMP_DOPPLER, TEMP_OP : REAL := 1.0;
  variable L : LINE;
  variable M : STRING(1 to 7);
  variable SPACE : CHARACTER := ' ';
  variable MILES : STRING(1 to 35) :=
    "MILES WITH A RELATIVE VELOCITY OF: ";
  variable ELE : STRING(1 to 18) := "DEGREES ELEVATION,";
  variable AZI : STRING(1 to 15) := "DEGREES AZIMUTH";
  variable MET : STRING(1 to 15) := "METERS PER SEC.";
  variable TAR : STRING(1 to 34) :=
    "TARGET DETECTED AT A DISTANCE OF: ";
  variable POSI : STRING(1 to 20) := ". IT'S POSITION IS: ";
  begin
    TEMP_DOPPLER := LOFREQ_TO_REAL_IN_HZ(TARGET_DOPPLER);
    If TEMP_DOPPLER < 0.0 then
      TEMP_DOPPLER := 0.0 - TEMP_DOPPLER;
      TEMP.VEL_DIR := OPENING;
      M := "OPENING";
    else
      TEMP.VEL_DIR := CLOSING;
      M := "CLOSING";
    end if;
    TEMP_OP := HIFREQ_TO_REAL_IN_MHZ(OP_FREQ)*1.0E6;
    TEMP.TARGET_ELEVATION := ANGLE_ELEV;
    TEMP.TARGET_AZIMUTH := ANGLE_AZIM;
    if PULSE_ON_TIME > 0 ns then

```

```

    TEMP_REAL := BIN_DISTANCE(PULSE_ON_TIME) *
                    (REAL(RANGE_BIN) - 0.5);
end if;
TEMP.TARGET_RANGE := TEMP_REAL;
if TEMP_OP > 0.0 then
    TEMP.REL_VEL := (TEMP_DOPPLER * C)/(2.0 * TEMP_OP);
end if;
TARGET_INFO <= TEMP;
WRITE (L, TAR);
WRITELINE (O, L);
WRITE (L, TEMP.TARGET_RANGE, DIGITS = > 2);
WRITE (L, SPACE);
WRITE (L, MILES);
WRITELINE (O, L);
WRITE (L, TEMP.REL_VEL, DIGITS = > 2);
WRITE (L, SPACE);
WRITE (L, MET);
WRITE (L, SPACE);
WRITE (L, M);
WRITE (L, POSI);
WRITELINE (O, L);
WRITE (L, TEMP.TARGET_ELEVATION/1 degrees);
WRITE (L, SPACE);
WRITE (L, ELE);
WRITELINE (O, L);
WRITE (L, TEMP.TARGET_AZIMUTH/1 degrees);
WRITE (L, SPACE);
WRITE (L, AZI);
WRITELINE (O, L);
WRITE (L, SPACE);
WRITELINE (O, L);
DETECTED <= '0';
end WRITE_TARGET;

procedure LOOK_FOR_TARGET (signal ANGLE_ELEV,
                           ANGLE_AZIM : in ANGLE;
                           signal RANGE_BIN : in NATURAL;
                           signal TARGET_MAP :
                               in TARGET_ENVIRONMENT;
                           signal NUMBER_TARGETS : in INTEGER;
                           signal FLAG : inout NATURAL;
                           signal PULSE_ON_TIME : in TIME) is

variable TEMP : integer := 0;
begin
for i in 1 to NUMBER_TARGETS
loop
    if ((TARGET_MAP(i).AZIMUTH < ANGLE_AZIM + 2 degrees)
and
(TARGET_MAP(i).AZIMUTH > ANGLE_AZIM - 2 degrees))
and

```



```

((TARGET_MAP(i).ELEVATION < ANGLE_ELEV + 2 degrees)
and
(TARGET_MAP(i).ELEVATION > ANGLE_ELEV - 2 degrees))
then
    if ((TARGET_MAP(i).TIME_AWAY >= PULSE_ON_TIME *
        RANGE_BIN)
        and (TARGET_MAP(i).TIME_AWAY < PULSE_ON_TIME *
            (RANGE_BIN + 1)))
        then
            FLAG <= i;
        else
            FLAG <= 0;
        end if;
    end if;
    if TARGET_MAP(i).AZIMUTH + 2 degrees > ANGLE_AZIM
        then
            exit;
        end if;
    end loop;
end LOOK_FOR_TARGET;
procedure POTENTIAL_TARGET_INFO
    (signal TARGET_MAP_FLAG : in TARGET;
     signal ANT_OUT : out RADAR_SIGNAL;
     signal OP_FREQ : in HIGH_FREQUENCY;
     signal FLAG : out NATURAL;
     signal PULSE_ON_TIME : in TIME) is
variable TEMP_HIPOWER, TEMP_LOPOWER : INTEGER := 0;
variable TEMP_POWER : REAL := 0.0;
begin
    ANT_OUT.HIFREQ <= OP_FREQ;
    ANT_OUT.LOFREQ <= TARGET_MAP_FLAG.TARGET_DOPPLER;
    TEMP_POWER := 15.0E15/TARGET_MAP_FLAG.ATTENUATION;
    if TEMP_POWER < 1.0E9 then
        ANT_OUT.LOPOWER_LEVEL <= TEMP_POWER*1 pW;
    else
        ANT_OUT.HIPOWER_LEVEL <= (TEMP_POWER/1.0E9)*1 mW;
    end if;
    FLAG <= 0 after PULSE_ON_TIME;
end POTENTIAL_TARGET_INFO;

procedure AMPLIFY_BY_K (variable K : in REAL;
    signal AMPLIFIER_IN : in RADAR_SIGNAL;
    signal AMPLIFIER_OUT : out RADAR_SIGNAL) is
variable TEMP_LOPOWER, TEMP_HIPOWER : INTEGER := 0;
variable TEMP_POWER : REAL := 0.0;
variable COUNT : INTEGER := 0;
begin
    TEMP_LOPOWER := AMPLIFIER_IN.LOPOWER_LEVEL/1 pW;
    TEMP_HIPOWER := AMPLIFIER_IN.HIPOWER_LEVEL/1 mW;
    TEMP_POWER := K * (REAL(TEMP_LOPOWER) +
        REAL(TEMP_HIPOWER)*1.0E+9);

```

```

If (TEMP_POWER > 1.0E+9) then
    TEMP_POWER := TEMP_POWER/1.0E+9 + 1.0;
    AMPLIFIER_OUT.HIPOWER_LEVEL <= TEMP_POWER*1 mW;
    AMPLIFIER_OUT.LOPOWER_LEVEL <= 0 pW;
else
    AMPLIFIER_OUT.HIPOWER_LEVEL <= 0 mW;
    AMPLIFIER_OUT.LOPOWER_LEVEL <= TEMP_POWER * 1 pW;
end if;
AMPLIFIER_OUT.LOFREQ <= AMPLIFIER_IN.LOFREQ;
AMPLIFIER_OUT.HIFREQ <= AMPLIFIER_IN.HIFREQ;

end AMPLIFY_BY_K;

procedure CHECK_FOR_DETECTION (signal RCVR_OUT :
                                in RADAR_SIGNAL;
                                signal AMPLIFIED_RCVR_NOISE : in REAL;
                                signal DETECTION_THRESHOLD : in LOW_POWER;
                                signal DETECTED_1 : out BIT) is

variable SIGNAL_REAL : REAL := 0.0;
variable THRESHOLD_REAL : REAL := 0.0;

begin
    SIGNAL_REAL := (REAL((RCVR_OUT.LOPOWER_LEVEL)/1 pW) +
                    REAL((RCVR_OUT.HIPOWER_LEVEL)/1 mW) * 1.0E9) +
                    AMPLIFIED_RCVR_NOISE;
    THRESHOLD_REAL := REAL((DETECTION_THRESHOLD)/1 PW);
    if THRESHOLD_REAL < SIGNAL_REAL then
        DETECTED_1 <= '1';
    else
        DETECTED_1 <= '0';
    end if;
end CHECK_FOR_DETECTION;

procedure READ_GAUSSIAN_NOISE
    (signal RANDOM_NOISE : out GAUSSIAN_REAL) is
variable TEMP_REAL : REAL := 0.0;
file INFILE : TEXT is in "NOISE.IN";
variable L : LINE;
variable COUNT : INTEGER := 1;
begin
    loop
        if not endfile(INFILE) then
            Readline (INFILE, L);
            Read (L, TEMP_REAL);
            RANDOM_NOISE(COUNT) <= TEMP_REAL;
            COUNT := COUNT + 1;
        else
            exit;
        end if;
    end loop;
end READ_GAUSSIAN_NOISE;

```

```
    end loop;  
end READ_GAUSSIAN_NOISE;  
end RADAR;
```

Appendix B. Constraints for the Diagnosis Example

The following are the constraints that were extracted from the model (keeping in mind that the diagnostic example is a subset of the model that was written for the radar system and does not include noise effects. First the input and output ports for the constraints of a sub-system are defined. The constraint are then either signal assignment statements, or signal assignment statements combined with mathematical expressions that are evaluated by some procedure which is within the package. In the latter case only the name of the procedure is specified. To the right of each statement of a constraint is a boolean expression that fires that constraint. That boolean expression is evaluated exactly once every time any of the elements in the expression whose names appear in capital letters changes value, and not otherwise. This is equivalent to a sensitivity list for a process statement. Note that a value of 'X' associated with an element or port implies a "don't care" condition.

SYNCHRONIZER_CONSTRAINTS :

Begin

IN : RANGE_BIN, START_UP.

OUT : TX_EN, RX_EN.

TX_EN = '1', RX_EN = '0' (RANGE_BIN = 0)
 *(START_UP = 1)

TX_EN = '0', RX_EN = '1' (RANGE_BIN /= 0)
 *(START_UP = 1)

END SYNCHRONIZER_CONSTRAINTS.

ANTENNA_CONSTRAINTS :

Begin

IN : ANGLE_AZIM, ANGLE_ELEV, ELEV_SCAN_RANGE,
 AZIM_SCAN_RANGE, RANGE_BIN, START_UP, ANT_IN,
 XMT_IN, FLAG, NUMBER_TARGETS, TARGET_MAP.

OUT : ANGLE_AZIM, ANGLE_ELEV, RCVD_SIG, ANT_OUT, TEMP_TARGET.

```

Procedure SCAN_ADVANCE
    ((angle_azim /= azimuth_scan_range)
    + (angle_elev /= elev_scan_range))
    *((RANGE_BIN = 0) * (start_up = 1))

ANGLE_AZIM = -AZIM_SCAN_RANGE
    (START_UP = 1)

ANGLE_ELEV = ELEV_SCAN_RANGE
    (START_UP = 1)

RCVD_SIG <= ANT_IN
    (rx_en = 1) * (range_bin /= 0)
    *(ANT_IN = 'X')

ANT_OUT <= XMT_OUT
    (tx_en = 1) * (range_bin = 0)
    *(XMT_OUT = 'X')

TEMP_TARGET <= TARGET_MAP(FLAG)
    (FLAG /= 0) * (start_up = 1)

Procedure LOOK_FOR_TARGET
    (RANGE_BIN /= 0)

Procedure POTENTIAL_TARGET_INFO
    (FLAG_DELAYED /= 0)
    *(start_up = 1)

ANT_IN.HIFREQ = 0 MHz
ANT_IN.LOFREQ = 0 Hz
ANT_IN.HIPOWER = 0 mW
ANT_IN.LOPOWER = 0 pW
    (FLAG_DELAYED = 0)
    *(start_up = 1)

END ANTENNA_CONSTRAINTS.

```

RANGE_BIN_INCREMENTER_CONSTRAINTS :

```

IN : AZIM_ANGLE, ELEV_ANGLE, AZIM_SCAN_RANGE,
    ELEV_SCAN_RANGE, RANGE_BIN, START_UP, INIT,
    PULSE_ON_TIME

OUT : RANGE_BIN

Begin

```

```

RANGE_BIN = 0          INIT = 1

RANGE_BIN = RANGE_BIN + 1 after PULSE_ON_TIME
    (angle_azim /= azimuth_scan_range)
    + (angle_elev /= elevation_scan_range)
    *(START_UP = 1)
    *(RANGE_BIN = 'X')

END RANGE_BIN_INCREMENTER_CONSTRAINTS.

```

RECEIVER_CONSTRAINTS :

IN : RCVD_SIG, AMP1_SIG, IF1, AMP2_SIG, IF2.
 OUT : RCVR_OUT, AMP1_SIG, IF1, AMP2_SIG, IF2.

Begin

```

AMPLIFY_BY_K (1000.0, RCVD_SIG, AMP1_SIG)
    RCVD_SIG = 'X'

IF1.HIFREQ = AMP1_SIG.HIFREQ - LO1
    AMP1_SIG.HIFREQ /= 0 MHz

IF1 = (0 MHz, 0 Hz, 0 mW, 0 pW)
    AMP1_SIG.HIFREQ = 0 MHz

AMPLIFY_BY_K (500.0, IF1, AMP2_SIG)
    IF1 = 'X'

IF2.HIFREQ = AMP2_SIG.HIFREQ - LO2
    AMP2_SIG.HIFREQ /= 0 MHz

IF2 = (0 MHz, 0 Hz, 0 mW, 0 pW)
    AMP2_SIG.HIFREQ = 0 MHz

RCVR_OUT.HIFREQ = IF2.HIFREQ - LO3
    IF2.HIFREQ /= 0 MHz

RCVR_OUT = (0 MHz, 0 Hz, 0 mW, 0 pW)
    IF2.HIFREQ = 0 MHz

RCVR_OUT.LOFREQ = IF2.LOFREQ
    IF2 = 'X'

RCVR_OUT.HIPOWER_LEVEL = IF2.HIPOWER_LEVEL
    IF2 = 'X'

RCVR_OUT.LOPOWER_LEVEL = IF2.LOPOWER_LEVEL
    IF2 = 'X'

```

END RECEIVER_CONSTRAINTS.

DETECTION_CHECKER_CONSTRAINTS :

IN : RCVR_OUT, DETECTION_THRESHOLD, DETECTED)
OUT : DETECTED

Begin

Procedure CHECK_FOR_DETECTION
 (start_up = 1)
 *(RCVR_OUT = 'X')

Procedure WRITE_TARGET
 (DETECTED /= 0)

END DETECTION_CHECKER_CONSTRAINTS.

Appendix C. Pascal Code for the Noise File

The following Pascal code produces an external text file that consists of 100 real numbers that are gaussian distributed between -10 and 10. The VHDL simulation reads in this file to represent noise in the receiver at simulation start.

```
var NOISE : array [1 .. 100] of real;
    count, i, j, k, l : integer;
    r, s, sum : real;
    TEMP : array [1 .. 12] of real;
    noiz : text;
    filevar : string;

begin
    count := 1;
    randomize;
    filevar := 'Noise.In';
    assign (noiz, filevar);
    rewrite (noiz);
    repeat
        i := 1;
        sum := 0.0;
        begin
            repeat
                begin
                    r := random (65535)/65535;
                    sum := sum + r;
                    i := i + 1;
                end
            until i = 13;
            sum := (sum - 6.0)/6.0;
            NOISE[count] := sum * 10.0;
        end;
        count := count + 1;
    until count = 101;
    for i := 1 to 100 do
        writeln (noiz, noise[i]);
    close (noiz);
end.
```


Appendix D. Pascal Code for the Targets File.

This Pascal program generates random target information for input to the Radar System Simulator written in VHDL. The target information is in the form of a record with five fields. The program generates from 1 to 20 targets at random. The azimuth angle is restricted to anywhere between -180 and 180 degrees. The Elevation angle is restricted to anywhere between -60 and 60 degrees. Time_Away is between 10 and 1000 us. Target_Doppler is anywhere between 0 and 40 KHz. Attenuation is dependant on the distance of the target from the radar, and is proportional to a randomly generated attenuation_factor.

```
type TARGET = record      { This is the type definition of the }
    AZIMUTH : integer;     { record that will be randomly generated }
    ELEVATION : integer;   { and written out to the file. }
    TIME_AWAY : integer;
    TARGET_DOPPLER : longint;
    ATTENUATION : real;
end;
```

```
type TARGET_ARRAY = array [1 .. 20] of TARGET;
```

```
var i, j, l, count : integer;
    { Some variable and constant declarations}
    k : longint;
    { for use within the program. }
    r, s, t, attenuation_factor : real;
    write_this : target;
    target_info : target_array;
    num_targets : integer;
    targets : text;
    filevar : string;
```

```
Procedure Write_Target_File;
    { This procedure writes out the target
      variable into the output file. }
```

```
begin
    writeln (targets, ' ',target_info[j].azimuth);
```

```

writeln (targets, ' ',target_info[j].elevation);
write (targets, ' ',target_info[j].time_away);
writeln (targets, ' us');
writeln (targets, target_info[j].attenuation:4);
writeln (targets, ' ',target_info[j].target_doppler);
end;

```

```

Procedure Sort_Target_Info;
    { This procedure sorts the target_info }
var a, b : integer; { by azimuth before it is written to the }
    temp : target; { output file. This is done to save search }
begin { time during the execution }
    { of the simulation }
    if num_targets > 1 then
    begin
        for a := 1 to num_targets-1 do
        begin
            for b := a+1 to num_targets do
            begin
                if target_info[a].azimuth >
                target_info[b].azimuth then
                begin
                    temp := target_info[a];
                    target_info[a] := target_info[b];
                    target_info[b] := temp;
                end;
            end;
        end;
    end;
end;

```

```

    { Main starts here }
begin
    l := 1;
    randomize; { Initialize the random number generator }
    filevar := 'TARGETS.IN';
    Assign (targets, filevar);
    rewrite (targets);
    repeat
        num_targets := random (20);
        until num_targets < > 0;
    repeat
        begin
            r := random (65535);
            repeat
                s := random (65535);
                until s < > 0.0;
            t := r/s;
        end
    until ((t > 0.001) and (t < 1.0));

```

```

attenuation_factor := t*t*1e3;
repeat
begin
  i := random (180);
  j := random (65535);
  If j/2 = trunc(j/2) then
    i := -i;
  write_this.azimuth := i;
  i := random (60);
  j := random (65530);
  If j/2 = trunc(j/2) then
    i := -i;
  write_this.elevation := i;
  repeat
    i := random (1000);
  until (i > 10);
  write_this.time_away := i;
  write_this.attenuation := attenuation_factor *
  write_this.time_away * write_this.time_away *
  write_this.time_away * write_this.time_away;
  k := random (500);
  j := random(65530);
  if j/2 = trunc(j/2) then
    k := -k;
  write_this.target_doppler := k;
  target_info[l] := write_this;
  l := succ(l);
end
until (l = num_targets + 1);
Sort_Target_Info;
for j := 1 to num_targets do
  Write_Target_File;
close (targets);
end.

```

Appendix E. Some More Test Simulations

There are four more test runs provided in this appendix. The test runs begin with the target file "TARGETS.", and are followed by the output file "DETECTED.OUT". The noise file used for all these test runs was the same, and a listing of this noise file is given at the end. The simulation was performed for a maximum detectable range of 100 miles, and antenna scan limits of +/- 60 degrees azimuth and elevation.

File "TARGETS." :

```
-3
50
620 us
6.4E+6
-208
83
-43
934 us
9.6E+06
-228
106
21
532 us
5.5E+06
-334
158
20
59 us
6.1E+05
401
171
-34
437 us
4.5E+06
70
177
22
49 us
```

5.0E+05
-264

File "DETECTED.OUT" created by VHDL for the above target scenario is :

TARGET DETECTED AT A DISTANCE OF:
57.65 MILES WITH A RELATIVE VELOCITY OF:
547.37 METERS PER SEC. OPENING. IT'S POSITION IS:
51 DEGREES ELEVATION,
-3 DEGREES AZIMUTH

File "TARGETS." :

-177
-55
698 us
1.6E+09
-300
-106
41
176 us
4.0E+06
-365
-73
22
712 us
1.6E+07
-107
-68
-1
808 us
1.8E+10
223
-47
-52
942 us
2.1E+12
462
-33
-12
607 us
1.4E+07
317
-22
26
397 us
9.1E+06
95
-1
-15
104 us
2.4E+06
-32
3
-44 1156 us
2.0E+15
-295
6
-26
140 us
3.2E+06

402
7
110
174 us
4.0E+06
247
51
-37
1091 us
1.5E+14
-73
78
-29
991 us
2.3E+07
327

File "DETECTED.OUT" created by VHDL for the above target scenario is :

TARGET DETECTED AT A DISTANCE OF:
37.03 MILES WITH A RELATIVE VELOCITY OF:
2.50 METERS PER SEC. CLOSING. IT'S POSITION IS:
27 DEGREES ELEVATION,
-21 DEGREES AZIMUTH

TARGET DETECTED AT A DISTANCE OF:
56.72 MILES WITH A RELATIVE VELOCITY OF:
8.34 METERS PER SEC. CLOSING. IT'S POSITION IS:
-12 DEGREES ELEVATION,
-33 DEGREES AZIMUTH

TARGET DETECTED AT A DISTANCE OF:
9.84 MILES WITH A RELATIVE VELOCITY OF:
0.84 METERS PER SEC. OPENING. IT'S POSITION IS:
-15 DEGREES ELEVATION,
0 DEGREES AZIMUTH

TARGET DETECTED AT A DISTANCE OF:
13.59 MILES WITH A RELATIVE VELOCITY OF:
10.58 METERS PER SEC. CLOSING. IT'S POSITION IS:
-27 DEGREES ELEVATION,
6 DEGREES AZIMUTH

TARGET DETECTED AT A DISTANCE OF:
88.59 MILES WITH A RELATIVE VELOCITY OF:
12.16 METERS PER SEC. CLOSING. IT'S POSITION IS:
-51 DEGREES ELEVATION,
-48 DEGREES AZIMUTH

File "TARGETS." :

-177
-22
488 us
8.7E+07
-107
-65
-16
985 us
1.8E+11
-150
44
-10
414 us
7.4E+06
-218
64
45
163 us
2.9E+06
59
70
4
912 us
1.6E+11
113
73
-31
965 us
1.7E+12
364
81
4
33 us
5.9E+05
-368

File "DETECTED.OUT" which was created by VHDL for the above target scenario is :

TARGET DETECTED AT A DISTANCE OF:
38.91 MILES WITH A RELATIVE VELOCITY OF:
5.74 METERS PER SEC. OPENING. IT'S POSITION IS:
-9 DEGREES ELEVATION,
45 DEGREES AZIMUTH

File "TARGETS." :

-153
17
568 us
4.7E+08
487
-108
-57
810 us
6.7E+10
252
-80
41
272 us
2.3E+06
148
-57
-28
758 us
6.3E+10
48
-24
46
365 us
3.0E+08
-164
114
58
155 us
1.3E+06
-325
160
-56
857 us
7.1E+13
130

File "DETECTED.OUT" which was created by VHDL for the above target scenario is
:

TARGET DETECTED AT A DISTANCE OF:
34.22 MILES WITH A RELATIVE VELOCITY OF:
4.32 METERS PER SEC. OPENING. IT'S POSITION IS:
45 DEGREES ELEVATION,
-24 DEGREES AZIMUTH

TARGET DETECTED AT A DISTANCE OF:
70.78 MILES WITH A RELATIVE VELOCITY OF:
1.26 METERS PER SEC. CLOSING. IT'S POSITION IS:

-27 DEGREES ELEVATION,
-57 DEGREES AZIMUTH

The noise file "NOISE.IN" that was used for these simulations is given below :

```
-2.3120978612E + 00
9.2701101189E-01
2.8122377350E-01
2.2354467025E-02
-5.1356781369E-01
3.8414587621E-01
-8.8283614357E-01
1.2885989674E + 00
5.9494926370E-01
-7.1267261772E-01
2.3891050583E + 00
-4.7659011727E-01
1.5404491238E + 00
-2.5378550902E + 00
2.2679229928E + 00
-2.2932529692E + 00
-8.0364182334E-03
-1.0312555637E-01
7.7482770016E-01
1.6896314946E + 00
-4.8956028588E-01
4.9243406826E-01
1.8645761806E + 00
6.1056432945E-01
-1.7014572366E + 00
2.4795147630E + 00
3.5089392436E + 00
2.3128353805E + 00
2.9511965616E + 00
5.0113171075E-01
1.4464281173E + 00
-1.2395920755E + 00
2.2049032323E + 00
-2.3031967651E + 00
-4.0072734675E-01
3.1222247653E + 00
-9.7833218893E-01
1.1187151903E + 00
-1.1766486102E + 00
1.8771140100E-01
-6.6124971394E-01
2.1482159660E + 00
-2.6751862873E + 00
1.7846697693E + 00
-8.4814730047E-01
-1.4204369167E + 00
-2.2568093386E + 00
5.2814017951E-01
```

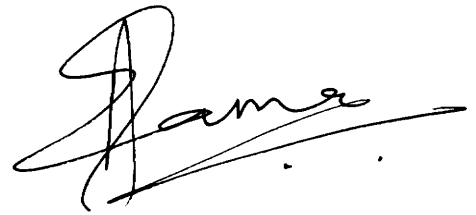
1.5988403143E + 00
1.7033391826E + 00
-7.6485847261E-01
-1.4287530836E + 00
7.0911217919E-01
5.5898883499E-02
2.8909742885E + 00
-3.6453548995E + 00
-3.1789628952E + 00
-1.6339869287E-01
-1.1825996287E + 00
-3.0955469090E-01
3.3150225068E-01
1.9905139747E + 00
-1.1116197456E-01
1.7797105868E-01
4.4586861978E-01
9.5936013829E-01
4.9624882374E-01
6.2149996179E-01
3.9281808702E + 00
-1.5695429923E + 00
2.9737290502E-01
-1.2205945933E + 00
3.8509956511E + 00
2.0875359222E + 00
1.3003992777E + 00
-2.2187126472E + 00
1.6788484524E + 00
1.0269067419E + 00
-5.2796215787E-02
1.5446453549E + 00
-6.4843213554E-01
-4.9225604643E-01
8.6299941501E-01
1.9094122729E-01
-3.6325373210E + 00
7.0188957549E-01
2.2734162406E + 00
-1.2034536253E + 00
-1.4662139824E + 00
2.1108059306E + 00
-2.4569568427E-01
1.1420869255E + 00
-1.6174817528E + 00
-6.8500292468E-01
-1.6848249028E + 00
3.4625518170E + 00
-1.0787111213E + 00

3.0582386002E+00
-3.8508939244E+00
-1.8025991201E+00

Vita

Anil Sama was born on September 7, 1966 in New Delhi, India. After completing his high school education from Jai Hind College in Bombay, India in June 1985, he attended the University of Bombay. Anil graduated from the University of Bombay in June 1989, after receiving his Bachelor of Engineering degree in Electronics Engineering. Anil enrolled at the Virginia Polytechnic Institute and State University in Blacksburg, VA in August 1989 and while there, completed requirements for a Master of Science degree in Electrical Engineering in May 1991.

Anil has been employed with Intel Corporation in Folsom, California since July 1991.

A handwritten signature in black ink, appearing to read 'Anil Sama', with a long horizontal flourish extending to the right.