

# **A Multi-Language Goal-Tree Based Functional Test Planning System**

By

Rajneesh Mahajan

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  
Computer Engineering

APPROVED

---

Dr. James R. Armstrong, Primary Advisor

---

Dr. F. Gail Gray

---

Dr. Dong Ha

July 2002

Blacksburg, Virginia

Keywords: Multi-Language, Test Plan, Test Bench, Validation, Testing

# **A Multi-Language Goal-Tree Based Functional Test Planning System**

By

Rajneesh Mahajan

Dr. James R. Armstrong, Primary Advisor

Computer Engineering

(ABSTRACT)

Test plans are used to guide, organize and document the testing activities during hardware design process. Manual test planning and configuration is known to be labor intensive, time consuming and error prone. It is desirable to develop efficient approaches to model testing and to develop test tools to automate test-planning activities.

With the emergence of new hardware design paradigms, there is a need to develop more specialized description languages. However, adopting a new language for hardware-based designs involves adapting the existing design and verification tool suite for the new language. This is a very time consuming and capital intensive process. To ease the adoption of new description languages, it is desirable to develop multi-language support methodologies for design and test tools.

This thesis addresses a subset of these problems. It presents a goal-tree based test methodology which is very effective for functional testing of hardware models in multiple application domains. Then it describes an approach for achieving a high degree of language independence using ideas of data abstraction. It also presents an automated test-planning tool called the “*Goal Tree System (GTS)*”, which provides an implementation of goal tree methodology and multi-language support ideas. We demonstrate the use of this tool by testing models developed in VHDL and SystemC. We also present the design aspects of the Goal Tree System, which enable it to work across multiple platforms and with multiple simulators.

## **Acknowledgements**

I would like to express my greatest thanks to my advisor, Dr. James R. Armstrong, for his constant support, advice and encouragement throughout the research period. His support made my work and learning experience, a very special one.

I would like to thank Dr. F. Gail Gray, for his advice and assistance in my research and for serving as a member of my graduate committee. I am thankful to Dr. Dong Ha for being on my graduate committee.

I am grateful to Mr. Ramesh Govindarajulu for providing the VHDL model of the GSM system, its goal tree and test results. I am also grateful to Mr. Anup Varma for providing the SystemC model of the GSM system.

# Table of Contents

1	Introduction.....	1
1.1	Introduction.....	1
1.2	Contributions.....	2
1.3	Organization.....	2
2	Design Process and Motivation for Test Planning Tools.....	4
3	Using Abstraction for Multi-Language Support.....	7
3.1	Language Abstraction Layer.....	7
3.2	Advantages of Language Abstraction Approach.....	11
4	Goal Tree Methodology and Test Planning Framework.....	12
4.1	Test Plan.....	12
4.2	Goal Tree Representation.....	13
4.3	Primitive Goals and Test Strategies.....	13
4.3.1	Confirmation Goals.....	14
4.3.2	Search Goals.....	16
4.4	Advantages of the Test Planning Framework.....	18
5	Goal Tree System (GTS).....	19
5.1	Multi-language support in GTS.....	22
5.2	Simulator Independence.....	24
5.3	Goal Tree System Commands.....	24
5.3.1	Commands for Project Organization.....	24
5.3.2	Commands for Tree Structure Construction.....	25
5.3.3	Commands for Goal Tree Syntax Checking.....	25
5.3.4	Commands for Goal Tree Library Maintenance.....	25
5.3.5	Node Property Editor Commands.....	26
5.4	Configuration and Specification files.....	28

5.5	Reuse of Test-vectors.....	30
5.6	Additional Advantages of the Goal Tree System (GTS) .....	30
5.6.1	Platform Independence .....	31
5.6.2	Project Organization Facilities.....	31
5.6.3	High Degree of Control .....	31
6	Implementation Details.....	32
6.1	Motivation for using Java for the Goal Tree System Implementation.....	32
6.2	Source Code Organization .....	33
6.3	Node Class .....	34
6.4	Generic Data-Types .....	34
6.5	Goal Tree Representation and Traversal.....	36
7	GTS Application to the VHDL Model of SAR System.....	37
7.1	Model Introduction .....	37
7.2	Tool Configuration and Goal Tree Construction.....	37
7.3	Test Results.....	41
8	GTS Application to the VHDL model of a GSM System .....	44
8.1	Model Introduction and Goal Tree Construction.....	44
8.2	Test Results.....	46
9	GTS Application to a SystemC Model of a GSM System.....	49
9.1	Model Introduction .....	49
9.2	Tool Configuration and Goal Tree Construction.....	52
9.3	Test Results.....	56
10	Conclusions and Future Work .....	58
11	Bibliography .....	59

## List of Illustrations

Figure 2.1: Synthesis Steps in Design Process.....	5
Figure 2.2: A Typical Test Bench.....	5
Figure 3.1: Abstraction Layer to Hide Language Details.....	8
Figure 3.2: Identification of Genetic Data-Types and Translation Functions.....	9
Figure 4.1: Even Sampling with Endpoints.....	15
Figure 4.2: Even Sampling without Endpoints.....	15
Figure 5.1: GTS in an Integrated Test Planning System.....	20
Figure 5.2: The Goal Tree System GUI.....	21
Figure 5.3: Upper portion of the Node Property Editor.....	21
Figure 5.4: Lower portion of the Node Property Editor.....	22
Figure 5.5: Multi-Language Support in GTS.....	23
Figure 6.1: Implementation Layers of the Goal Tree System.....	33
Figure 6.2: Generic Data-Type Class Hierarchy.....	35
Figure 6.3: Source Code of Abstract Class GenericDatatype.....	36
Figure 7.1: adjustable_parameters.list File for SAR model.....	38
Figure 7.2: A Goal Tree of SAR system.....	40
Figure 8.1: Goal Tree for VHDL model of GSM System.....	46
Figure 9.1: High Level Block Diagram of GSM Model.....	50
Figure 9.2: Input File (Time Domain).....	50
Figure 9.3: Input File (Freq. Domain).....	51
Figure 9.4: Output File (Time Domain).....	51
Figure 9.5: Output File (Freq. Domain).....	51
Figure 9.6: adjustable_parameters.list file for GSM model.....	52
Figure 9.7: A Goal Tree for SystemC Model of GSM System.....	54

## List of Tables

Table 4.1: Test Strategies and Control Variables.....	16
Table 7.1: SAR Primitive Goals.....	41
Table 7.2: Results of Test Group TG11.....	42
Table 7.3: Results of Test Group TG211 and TG212.....	42
Table 7.4: Results of Test Group TG221 and TG222 and TG223.....	43
Table 8.1: Description of the Goal tree for the GSM system modeled in VHDL.....	47
Table 8.2: Results for the Test groups TG1111 and TG1112.....	47
Table 8.3: Results for the Test Group TG21 and TG22.....	48
Table 8.4: Results for the Test Group TG311.....	48
Table 9.1: Description of test goals.....	55
Table 9.2: Test Results for Test Groups TG11, TG12 and TG13.....	56
Table 9.3: Test Results for Test Groups TG211, TG212 and TG213.....	57
Table 9.4: Test Results for Test Groups TG221, TG222 and TG223.....	57

# 1 Introduction

## 1.1 Introduction

In general, hardware design involves a series of synthesis steps, each of which involves a design transformation from one level to the other [1]. After each synthesis step, the design needs to be tested to ensure its compliance with the system specifications. A test plan is used to guide, organize and document these testing activities. Manual test planning and configuration is known to be labor intensive, time consuming and error prone. It is desirable to develop efficient approaches to model testing and to develop test tools to automate test-planning activities.

As new hardware design paradigms are emerging, there is a constant need to develop new system description languages. A good example is the emergence of SystemC because VHDL and Verilog were not sufficient and effective in modeling Systems-On-a-Chip. However, adopting a new language for hardware-based designs is not just a matter of developing a new compiler (even though this can be a very intensive task). New test tools, simulators and design compilers might need to be developed for new languages. Developing this infrastructure is a very time consuming and capital intensive process. This is probably the biggest reason that very few hardware description languages are available and popular in the current market. However, as we move towards new hardware design paradigms, we need more powerful and probably more specialized languages.

This thesis attempts to address a subset of the problems described in the above two paragraphs. It presents a goal-tree based test planning infrastructure which is very effective for functional testing of hardware models in multiple application domains. Then

we describe an approach for achieving a high degree of language independence using ideas of data abstraction. We also present an automated test-planning tool called the “*Goal Tree System (GTS)*”, which provides an implementation of goal tree methodology and multi-language support ideas. We demonstrate the use of this tool by testing models developed in VHDL and SystemC. We also present the design aspects of the Goal Tree System, which enable it to work across multiple platforms and with multiple simulators.

The work described in this thesis is an extension of the work carried out at the Virginia Tech Information Systems Center by Dr. Morris Lin for his PhD dissertation [2][11]. He introduced the idea of using a goal tree structure as a test-planning framework for functional validation of DSP real number models.

## **1.2 Contributions**

This thesis introduces an abstraction based approach to multi-language support for test planning tools. Multi-language support is achieved by introducing a language abstraction layer between a test tool and the model written in a particular language. The design and implementation details of this approach for data-type abstraction have also been presented.

This thesis also extends the applicability of the goal tree approach from DSP models to models developed in multiple application domains.

Another contribution of this thesis is platform independence of the Goal Tree System. The Goal Tree System has been implemented using Java and it can run on any platform capable of supporting Java Run Time Environment. Further, the Goal Tree System code need not be re-compiled for every platform.

This thesis also presents a command file based approach for achieving simulator independence for the Goal Tree System.

## **1.3 Organization**

The organization of this thesis is as follows.

Chapter 2, “Design Process and Motivation for Test Planning Tools,” discusses the motivation behind test planning tools.

Chapter 3, “Using Abstraction for Multi-language Support,” discusses the concepts behind abstraction based multi-language support in a test-tool.

Chapter 4, “Goal Tree Methodology and Test Planning Framework,” gives the background of *goal tree methodology* and *test planning framework*.

Chapter 5, “Goal Tree System (GTS),” is the discussion of actual Goal Tree System (GTS) test-planning tool, its features and advantages.

Chapter 6, “Implementation Details,” provides some of the implementation details of the Goal tree System (GTS).

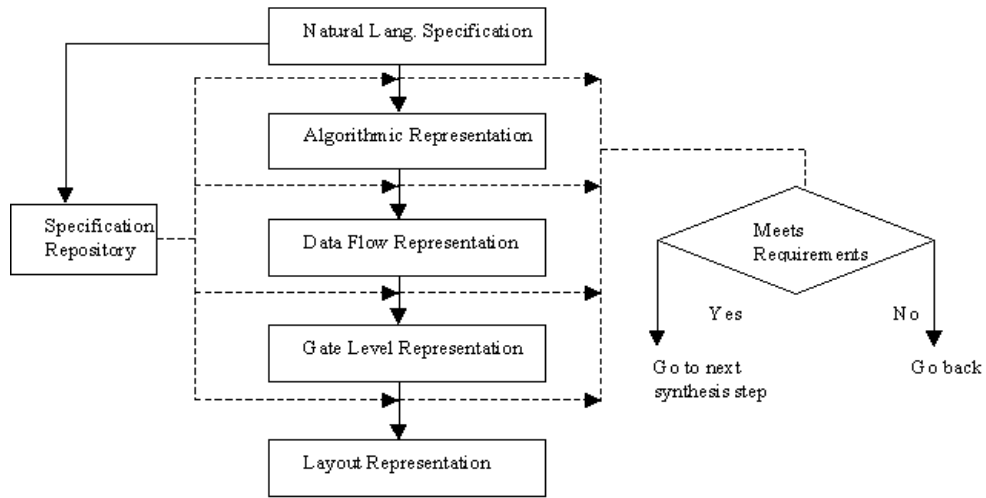
Chapter 7, “GTS Application to the VHDL Model of SAR System,” Chapter 8, “GTS Application to the VHDL Model of a GSM System,” and Chapter 9, “GTS Application to a SystemC Model of a GSM System,” demonstrate the language-independence and multi-application effectiveness of GTS by presenting test results of Synthetic Aperture Radar (SAR) model developed in VHDL, a GSM system model developed in VHDL and a GSM system model developed in SystemC.

## 2 Design Process and Motivation for Test Planning Tools

A typical top-down design process starts in the behavioral domain at a high abstraction level and ends in the structural domain at a low abstraction level [1]. This design process involves a series of synthesis steps, where each synthesis step involves a transformation from design at a level  $i$  to level  $j$  where  $j \leq i$ . In most cases  $j = i - 1$  or  $j = i$ , implying that either the levels are adjacent or the transformation is from the behavioral domain to the structural domain at the same level (there are also some situations where some levels are skipped). Common synthesis steps (Figure 2.1) are:

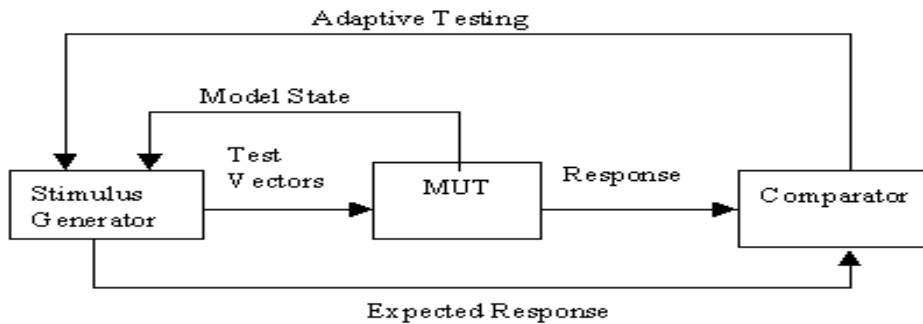
1. Natural Language Synthesis: Transformation from a written natural language specification to an algorithmic representation.
2. Algorithmic Synthesis: Translation from an algorithmic representation to a data flow representation or a structural gate level representation.
3. Logic Synthesis: Translation from a data flow representation to a structural logic gate representation.
4. Layout Synthesis: Translation from logic gate representation to layout representation. This completes the synthesis process since the layout information can be fabricated.

The natural language system specification is also used to build a specification repository (Figure 2.1), which stores all the information about the system functionality in an organized manner. This repository contains information about the configurable parameters in the system and the expected response of the system to various operational conditions. This repository can be used (directly or indirectly) to verify the functionality of an HDL model used to describe system state at various levels of synthesis.



**Figure 2.1: Synthesis Steps in Design Process**

HDL models are used at all levels to specify and document hardware designs. After every synthesis step, the synthesized model needs to be tested to verify system functions. Testing a model involves developing a test bench, which acts like a testing framework in which the model under test (MUT) is inserted (Figure 2.2). The test scenario (also called test case) is developed by configuring (either hard or soft configuration as discussed in Section 3.1) the test bench with some values, which are used by the test-bench to produce test vectors, which are sent to the MUT for simulation [2][3][4][11]. The MUT response is either compared to the expected system functions using a comparator and the results are then used to determine a pass/fail condition, or is used to generate the next set of test vectors to apply to the MUT (Figure 2.2). This test



**Figure 2.2: A Typical Test Bench**

process is repeated for all test cases and is used to ensure the compliance of model behavior with system specification. If a model satisfies the specifications, the design moves to the next synthesis step. Otherwise, the previous step is repeated with different design parameters (Figure 2.1).

A test plan serves as a guide and documentation tool for creating different test cases to apply to the MUT [2][11]. It works by creating a series of test-bench configurations. The test-bench used might be different at different synthesis levels, however the test plan and test cases are not changed significantly. Performing these configuration and test tasks manually at every synthesis level is very labor intensive, time consuming and unreliable. Therefore it is desirable to develop efficient approaches to HDL model testing and to create a software system to automate and organize the testing process.

## 3 Using Abstraction for Multi-Language Support

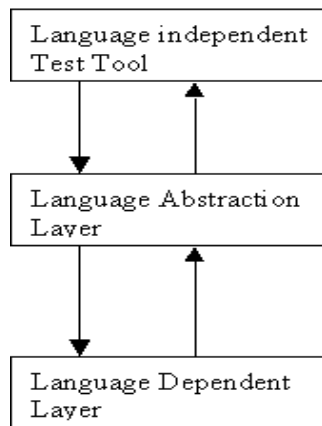
The approach used in this thesis to achieve language independence is to use the ideas of abstraction. Abstraction is the process of hiding the details. The idea of abstraction is not new; most systems use it in one way or the other. An example is the Java Run Time environment, which hides the platform details from java code, which allows Java programs to be platform independent. Another example is a device driver, which hides the device details to give a consistent view of hardware to software using that class of hardware. Figure 3.1 shows how abstraction can be used to make the test tools language independent.

Here we introduce a new language abstraction layer (LAL) between the test tools and the model written in a particular language. The tool doesn't know the details of the HDL in which the MUT (model under test) is to be tested. It interacts with only the LAL. The LAL in turn interacts with the Language Dependant layer (LDL). In effect the LAL hides the details of a particular language and provides a consistent language view to the test tool. To make the test tool work with any other hardware description language, all that is needed is to add that language support in LAL. Though adding a new language support to LAL doesn't look like a simple task, we will show that it's not really very complicated and time consuming (and it's much more effective than developing a new test tool for every language).

### 3.1 Language Abstraction Layer

The key to developing this language independent approach is the design of the LAL. Before we design LAL, we should look at the goals in developing such a layer:

1. It should hide HDL details from the Language Independent Layer of the test tool (LIL).
2. The test tool user knows about the model and the language in which it was developed. He is not expected to be familiar with the internal representation used in the Language Independent Layer. Therefore, the test planner should get a system interface where he is dealing with the language in which model was developed (e.g. VHDL).
3. The LAL should be able to provide all necessary language functionality to the test tool and it should be able to translate all the test tool operations to LDL operations.



**Figure 3.1: Abstraction Layer to Hide Language Details**

4. The LAL design should permit phased support for a language i.e., it should allow additions of a language-specific data types as and when they are required.
5. The LAL design should permit addition of new languages.

The system we are dealing with here is a functional testing system. A general functional test planning system handles hard and soft configuration of a test-bench. Configuring a test-bench involves value assignments to different model parameters. In *Hard Configuration*, the test-bench is modified for configuration e.g. modifying *generics* in VHDL or *parameters* in Verilog. In *Soft Configuration* the test-bench code itself is not modified and the configuration values are read by test-bench using file IO [3]. The most important language detail for test-bench configuration is the language data type. Any data type of a language is a set of attributes describing how that data-type should be interpreted and

assigned values. This means that the LAL should be able to provide the following language translation functions:

$$\begin{array}{ccc}
 & \mathbf{f} & \\
 (x_1, x_2, \dots, x_n) & \longrightarrow & (y_1, y_2, \dots, y_m) \\
 & \mathbf{g} & \\
 (y_1, y_2, \dots, y_m) & \longrightarrow & (x_1, x_2, \dots, x_n)
 \end{array}$$

Function  $f$  should be able to translate a particular language data type (example STD\_LOGIC\_VECTOR in VHDL) with attributes  $x_1, x_2, \dots, x_n$  to a language independent generic data type with attributes  $y_1, y_2, \dots, y_m$ , which will be exported to the test tool. Function  $g$  will have to do the reverse functions. Supporting a new HDL in the test-tool will involve writing these data-type translation functions for required data-types of the new HDL.

As a practical example, consider adding support for VHDL and SystemC in LAL (Figure 3.2). First, we can identify the required data types of these languages to be

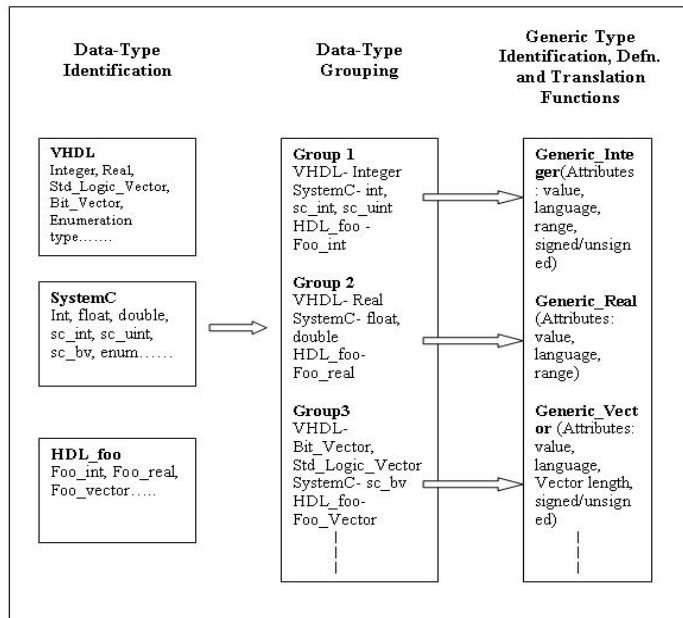


Figure 3.2: Identification of Genetic Data-Types and Translation Functions

supported. For VHDL we might want to add support for data types *Integer*, *Real*, *Std\_Logic\_vector*, and *Bit\_Vector*. For SystemC we might want to add support for *int*, *float*, *double*, *sc\_int*, *sc\_uint*, and *sc\_bv*. Once we have identified the data-types to be supported, we can group them by their common characteristics (Figure 3.2). For every such group, we can either reuse/extend existing generic types defined for LIL or we can add another generic data-type with appropriate attributes to ensure ability of translation from generic types to language specific types, and vice versa. Then we can define language translation functions for translation to and from these generic types. Since a similar set of data-types is supported across different languages, these language translation functions are highly re-usable across different languages.

If support for a new data-types of a language is to be added to LAL, all that is needed is identification (and if required definition) of corresponding generic types and writing the corresponding translation functions. E.g. if support for *Std\_Logic\_Vector* is to be added to tool, we can identify the internal type as *Generic\_Vector* (Figure 3.2) and re-use the translation function of *Bit\_vector* with little modification.

Similarly, if we want to add support for another language (e.g. Verilog) in the tool, we need to identify the mappings of the new language's data types to existing generic types. Most languages support a similar set of data-types, therefore one should be able to map most of these to existing generic types. However, if no appropriate mapping exists, a new generic data-type should be defined. Again the translation functions can either be reused or written from scratch.

The user interacts with the tool using language dependent data-types. E.g., if he is using a VHDL model and he needs to enter a value in *STD\_LOGIC\_VECTOR*, he shouldn't be forced to deal with the language independent data type. He will enter values in *STD\_LOGIC\_VECTOR* and the LAL will take that value and present it to the test tool in the language independent data format. The test tool will do all the computations using language independent data-types and will generate another set of values to be given to the test bench. The LAL will convert these values to the language dependent data format and give them to the test bench.

The Goal Tree System described in this thesis uses these data-type translation functions for support of multiple languages by the test planning tool. Section 5.1 describes the details of this implementation

### **3.2 Advantages of Language Abstraction Approach**

The language abstraction approach described in this chapter has following advantages:

1. This approach allows the test tool to do all its operations on generic data-types. Therefore the test-tool need not be re-implemented if it needs to support a new hardware description language.
2. This approach allows phased implementation of a hardware description language in the LAL.
3. This approach allows easy integration of new hardware description languages in the LAL.
4. Finally, this approach permits heavy reuse of language translation functions among data-types in a single hardware description language and across different hardware description languages.

## 4 Goal Tree Methodology and Test Planning Framework

This section presents the background of goal tree methodology and test planning framework. This methodology has been discussed in detail in [2][11]. The novel work done in this thesis is developing a methodology for multi-language support in test tools and integrating it with goal tree methodology for its application on models developed in multiple languages and in multiple application domains. Therefore, most of the elements of the basic goal-tree methodology itself remain unchanged from [2][11] to this thesis. Because of their similarity, some of the paragraphs of this chapter have been directly reproduced from [2][11] and some have been reproduced with appropriate modifications.

### 4.1 Test Plan

A *test plan* is a document that organizes the system requirements in terms of how these requirements will be tested. It defines the elements required in test planning such as test goals, test groups, test cases and test oracles [2][11].

In a goal tree based testing framework, a *test case* is a testing scenario in which all system parameters have been specified. A test case corresponds to a fully configured test bench, which can be directly executed (simulated). A *test group* is a set of test cases, in which all system parameters except one (called the adjustable parameter) have been specified. The adjustable parameter's value is allowed to vary according to some test value selection strategy (Section 4.3). Each generated value corresponds to a test case. A test group can have a fixed or a variable number of test cases depending on the test value selection strategy.

A test plan partitions the overall test goal into smaller goals. These goals are further divided into sub-goals until primitive goals are reached which can be directly met by applying test groups.

A *test oracle* is a rule, which specifies the correct response of the MUT for a given input case. This is used by the comparator to determine success or failure of a test and for feedback testing.

## 4.2 Goal Tree Representation

A goal tree can be represented with a node set and an edge set. Nodes in the goal tree represent goals at various levels and edges represent the relationship among goals. A node can be a Goal Node, a Test Group Node or an Operator Node [2][11].

The Goal nodes are used to subdivide goals into smaller goals. No processing/testing takes place while traversing a Goal Node, they are meant to document the subdivision of goals. The Test Group Nodes are the leaves of a goal tree. They represent the primitive goals (Section 4.3) of the test plan. The traversal of Test Group Nodes results in the generation of test cases, which are applied to the MUT. The number of these cases depends on the test value selection strategy. The Operator Nodes can be either AND nodes or OR nodes. They help in a more complex sub-division of test goals. Satisfaction of a goal is contingent on the satisfaction of goals in all of its children for the AND type node and any one of its children for the OR type node.

The order in which these test groups are executed is equivalent to the post-order traversal in a tree data structure. The left sub tree of any node is traversed first, followed by the right sub tree and then finally the parent is traversed.

## 4.3 Primitive Goals and Test Strategies

Model testing efficiency is directly related to the number of test cases to be executed. Therefore, one wants to limit the number of test cases to a small subset of the possible cases in the test space. References [2][11] define two types of primitive goals and various test value selection strategies to select the test values from the test space.

The primitive goals can be sub-divided into confirmation goals and search goals. A confirmation goal aims to confirm the correctness of the model by sampling a set of

values from the test space. A search goal aims to explore the boundary conditions from the output space.

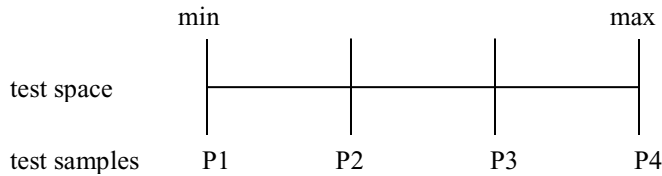
References [2][11] also define three classes of test value selection strategies: sampling strategies, search strategies and file IO strategies. Sampling strategies are applied on confirmation goals. Examples of such strategies are 1) Even Sampling With Endpoints 2) Even Sampling without Endpoints 3) Random 4) Enumeration (Enumeration can be used to handle the difficult corner cases known by the tester). Search strategies use previous test results of the same test group to adaptively choose test values for next test case. They are used to get ranges of extreme values, that is, lower bounds and upper bounds. The search strategies defined in [2][11] are 1) Arithmetic Search 2) Geometric Search 3) Binary Search and 4) Combinations of the first three. File IO strategy allows reuse of test values. It allows a test group to sequence through the list of enumerated test files. The test values are read from these files rather than being generated by the stimulus generator.

### 4.3.1 Confirmation Goals

The *test space* can be defined as the set of all values from the space of adjustable requirement. Testing all values from the *test space* is usually impractical. It is desirable to limit the number of test values. By sampling a set of values from the test space to form the test cases, a confirmation goal aims to confirm the correctness of the MUT for all values in the test space, provided that the other requirements remain unchanged. A test case is then formed by a sample value as well as the constrained requirements. Determining the number of test cases is a trade-off between testing time and level of confidence the tester gains after applying the test group. The more test cases the test group issues, the more time it takes to simulate, and the more evidence one can collect to draw conclusion on the primitive goal [2][11].

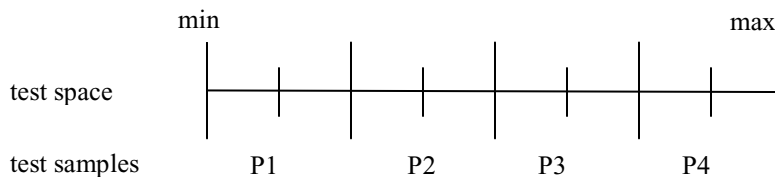
Four possible sampling strategies that can be used for confirmation goals are as follows [2][11].

**4.3.1.1 Even Sampling with Endpoints.** This strategy selects a number of equally spaced samples for the adjustable requirement from the test space, including the values at both ends. Figure 4.1 depicts the selection of 4 samples using this test strategy.



**Figure 4.1: Even Sampling with Endpoints**

**4.3.1.2 Even Sampling without Endpoints.** Given the number of samples  $N$ , this strategy divides the test space into  $n$  sections of equal length and picks the midpoints of these sections as samples. In contrast to the previous strategy, this strategy selects interior values only. Figure 4.2 illustrates the selection of 4 samples using this strategy.



**Figure 4.2: Even Sampling without Endpoints**

**4.3.1.3 Random.** A sequence of random values is chosen from the test space.

**4.3.1.4 Enumeration.** All test values are enumerated individually.

**4.3.1.5 File IO.** All test values are read from specified files. The values are picked one per-line, per-file. This strategy is useful for reusing previously generated test values.

**4.3.1.6 File Enumeration.** This is similar to enumeration strategy. The only difference is that the enumerated values are file names.

Table 4.1 lists the control variables that the test strategies use for test value generation. Note that the enumeration test strategy specifies all the test values so as to involve no control variables.

<b>Test Strategy</b>	<b>Control Variables</b>
even sampling with endpoints	test space, no. of tests
even sampling without endpoints	test space, no. of tests
Random	test space, no. of tests,
Enumeration	None
arithmetic search	test space, initial value, increment, direction
geometric search	test space, initial value, initial increment, direction
binary search	test space, lower bound, upper bound, precision
arithmetic-binary search	test space, initial value, increment, direction, precision
geometric-binary search	test space, initial value, initial increment, direction, precision
file enumeration	file names
file I/O	file names

**Table 4.1: Test Strategies and Control Variables [2][11]**

### 4.3.2 Search Goals

The objective of a search goal is to search for the extreme value of the adjustable requirement with which the MUT can barely pass or fail the test, provided that the adjustable requirement has a monotonic effect on the MUT in the test range, i.e., if the MUT passes (fails) both tests with test values  $a$  and  $b$ , respectively, then the MUT passes (fails) any test with test value  $c$ , where  $c$  lies between  $a$  and  $b$ . The requirements other than the adjustable requirement remain unchanged.

Defined as follows are several search strategies which choose test values adaptively from previous test results in the same test group, and give ranges of the extreme values, that is, lower bounds and upper bounds [2][11].

#### 4.3.2.1 Arithmetic Search

An arithmetic sequence is chosen to form the test values, i.e., a sequence  $a, a \pm d, a \pm 2d, a \pm 3d, \dots$ , where  $a$  is the initial value and  $d$  is the increment. The sequence continues until the test result turns from a failure to a success or vice versa, or until the sequence goes beyond the test space. Let's call these two consecutive test values that cause the test result to change status  $P_i$  and  $P_{i+1}$ . The extreme value is thus bounded by the interval  $(P_i, P_{i+1})$  with a precision  $P_{i+1} - P_i = d$ . The average and worst case search times grow as  $O(\text{len} / d)$ , where  $\text{len}$  is the length of the test space.

#### 4.3.2.2 Geometric Search

An offset geometric sequence of ratio 2 is applied to choose the test values, i.e., a sequence  $a, a \pm 2^0 d, a \pm 2^1 d, a \pm 2^2 d, a \pm 2^3 d$ , and so on, where  $a$  is the offset and  $d$  is the initial increment. The sequence continues until the test result turns from a failure to a success or vice versa, or until the sequence goes beyond the test space. The average and worst case search times are  $O(\log(\text{len} / d))$ , where  $\text{len}$  is the length of the search space. The precision of the resultant range of the extreme value is  $2^{n-3} d$  for  $n > 2$ , where  $n$  is the number of tests applied. Geometric search finds a range of the extreme value faster than arithmetic search, but with a coarser precision than that found in arithmetic search.

#### 4.3.2.3 Binary Search

Given a lower bound and an upper bound on the adjustable requirement, the binary search strategy nails down the extreme value within a given precision by narrowing the bounds iteration by iteration until their difference is satisfactorily small. To achieve this, one of the initial bounds has to cause the MUT to pass and the other cause the MUT to fail. The algorithm is described as follows. Suppose that the MUT passes the test with the lower bound and fails the test with the upper bound. In each iteration, choose the midpoint of both bounds as the test value. If the MUT passes, it is inferred by the monotonic assumption that the MUT passes all values between the test value and the lower bound, and the extreme value must fall between the test value and the upper bound. As a consequence, make the test value the new lower bound and go to the next iteration. On the other hand, if the MUT fails, it is inferred that the MUT fails all values between

the test value and the upper bound, and the extreme value must lie between the lower bound and the test value. As a result, make the test value the new upper bound and start a new iteration. This process iterates until the difference between the two bounds is smaller than the given precision. The algorithm is similar if in the beginning the MUT fails the test with the lower bound and passes the test with the upper bound. The complexity of binary search strategy is  $O(\log(len / p))$ , where  $len$  is the length of the search space and  $p$  is the precision.

#### **4.3.2.4 Combination of the above**

Use arithmetic search or geometric search to coarsely obtain a lower bound and an upper bound of the extreme value of the adjustable requirement in the first stage, and then apply binary search to finely tune the extreme value within the two bounds found above with a given precision in the second stage. The geometric-binary search, which runs in logarithmic time in both stages, is more efficient than the arithmetic-binary search, which runs in linear time in the first stage and in logarithmic time in the second stage.

#### **4.4 Advantages of the Test Planning Framework**

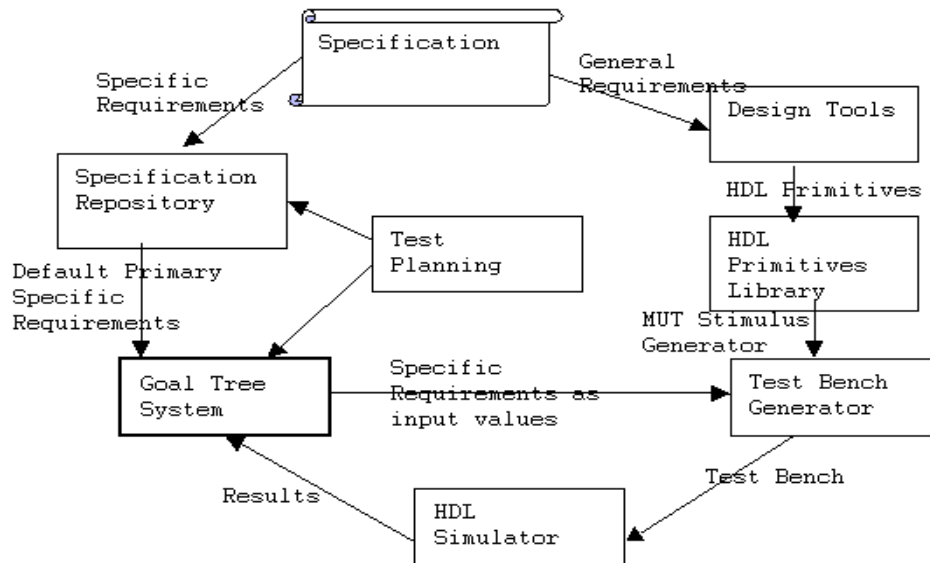
The test planning framework using goal tree based approach offers several advantages [2][11].

- 1) The hierarchical structure of a goal tree helps organize the large number of requirements to be tested; and helps define test goals, test groups and test cases by identifying different adjustable parameters.
- 2) Directly testing the grand goal is impractical since the test space grows exponentially. Use of the AND operator divides the grand goal into a collection of primitive goals and reduces the test space to smaller subspaces. This division allows one to test specific goals leading to enhanced efficiency of the whole testing process.
- 3) Use of the OR operator allows coexistence of equally effective test goals and test groups defined in the test plan. They can be used for tests where satisfying one of several conditions is sufficient for satisfying a goal.
- 4) The test plan can be easily modified by pruning and grafting the goal tree.

## 5 Goal Tree System (GTS)

The Goal Tree System is the test-planning tool developed as a part of this thesis to demonstrate the effectiveness of goal-tree based testing and of the multi-language support approach. This tool was written in Java and it can run on any platform supporting the Java Run Time Environment. GTS is a component of a system [2][4][11] shown in Figure 5.1. In this system, general requirements are used to develop the HDL primitives using different design tools. The code elements are then stored in an HDL Primitives Library. Specific requirements are stored in a Specification Repository and are linked to the test plans. Test plans define groups of tests and guide the testing process. The Goal Tree System manipulates the test plans and commands the Test Bench Generator. It overrides the specific requirements. These overridden values of a requirement along with default values of other requirements are used in the test bench to generate stimulus for Model Under Test (MUT). The test bench generator receives the MUT and Stimulus Generator from HDL primitives library and uses the specific requirements (as generic, input signals or other parameter values) to generate test bench for simulation. The simulation results are fed back to Goal Tree System and a new test is issued accordingly.

GTS provides a GUI (Figures 5.2, 5.3 and 5.4) to users for documenting test plans, creating goal trees, specifying the attributes of test goals and test groups, for controlling model simulation and for viewing test reports. It allows one to construct goal

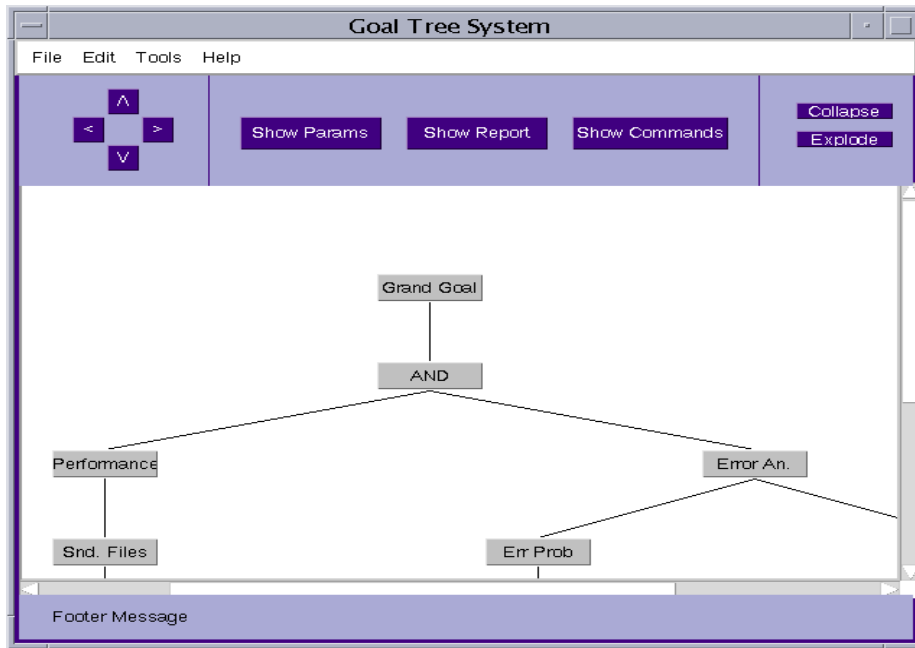


**Figure 5.1: GTS in an Integrated Test Planning System**

trees and enter information on test plans. The goal tree is traversed depth first and whenever a test group node is reached, a set of values for the specific requirement (corresponding to that test group) is produced.

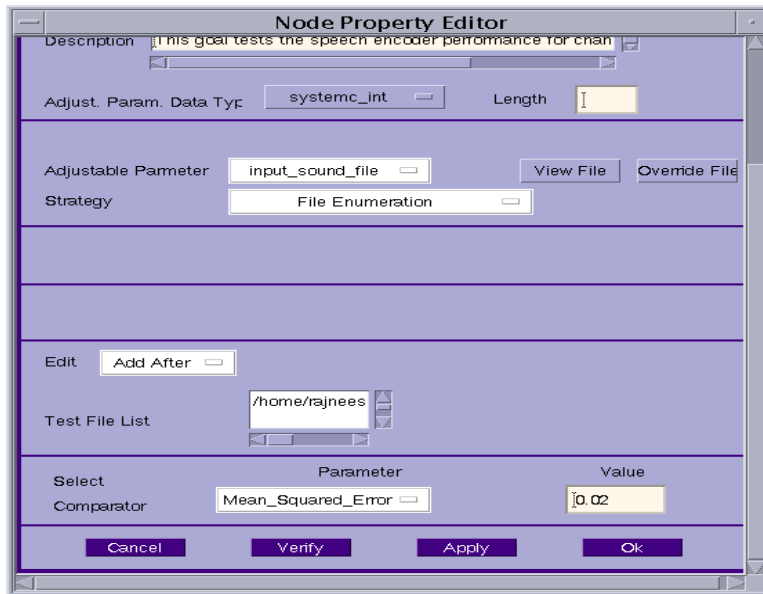
It also commands the test bench generator to create test benches and start a new test for simulation. Based on the simulations results, the Goal Tree System determines whether a new test should be initiated for the current test group. If not, this test group terminates and the tree traversal process continues.

Right clicking on a node in Goal Tree System opens a goal attribute specification window called *Node Property Editor*. Figures 5.3 and 5.4 show the Node Property Editor window for a leaf node (which corresponds to the test group). In this window the test planner can specify different parameters corresponding to a test group, viz., *Adjustable Parameter*, *Node Data Type*, *Test Value Selection Strategy* and different parameters specific to a strategy.



**Figure 5.2: The Goal Tree System GUI**

**Figure 5.3: Upper portion of the Node Property Editor**



**Figure 5.4: Lower portion of the Node Property Editor**

The backend of the *Goal Tree System* implements the functionality to make this test planning and execution independent of any hardware description language (Section 5.1), platform (Section 5.6.1) and simulator (Section 5.2).

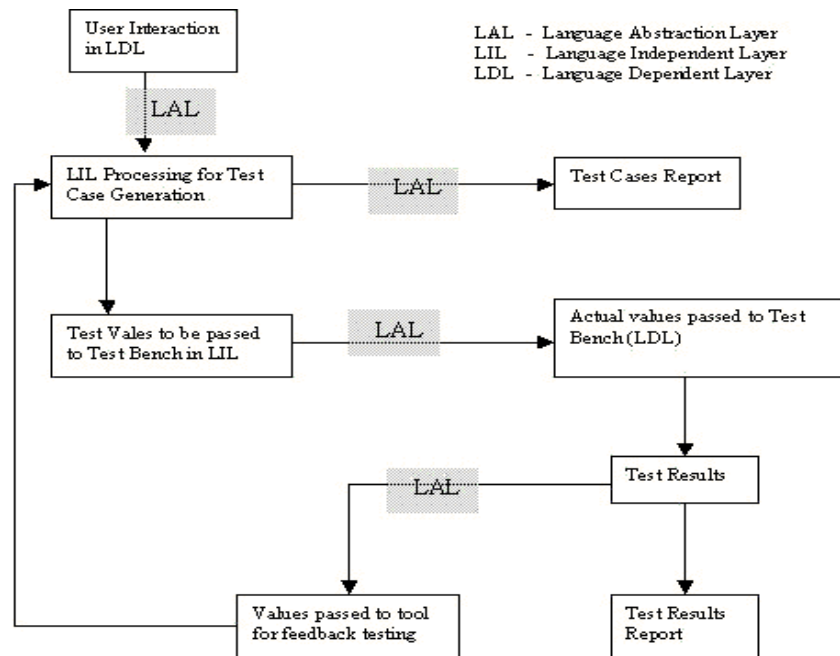
### 5.1 Multi-language support in GTS

Data-type abstraction explained in chapter 3 has been used in GTS to support multiple hardware description languages.

Figure 5.5 shows the system components where the Language Abstraction Layer (LAL) has been used to handle language details. The test tool user uses language dependent values to specify the test plan. He is not expected to know the internal data-type representation of the Goal Tree System. The Language Abstraction Layer converts these values into an internal representation. The tool's internal processing for test case generation is done in a language-independent format. When the test-bench is configured for individual test cases, LAL again kicks in, converting the internal representation to an HDL-specific representation. When a test-bench finishes execution and the test results have been generated, LAL again converts these values to the internal representation (if they are required for feedback testing). Reports are generated to document generated test

cases and test results. These reports show values in representation specific to the HDL used. It can be observed that LAL functionality is usually required at the time of some user/GUI interaction and report generation. All the back-end tool processing is language independent. Therefore, if a new language needs to be supported, no change is required in the back-end. As explained in chapter 3, even the LAL changes are very simple and can be accomplished in a fast and cost effective way.

The internal representation of data-types can be customized to support a wide variety of language data types. The representation currently used in GTS has one generic type with three attributes; the *value* attribute, *length* attribute and *specificType* attribute. The value attribute is of JAVA type *double*, the length attribute is of JAVA type *int* and the *specificType* attribute is of JAVA type *string*. The combination of these attributes is sufficient to support most of the data-types of VHDL and SystemC. More attributes would be needed to support other data-types (e.g. VHDL Enumeration Types) required



**Figure 5.5: Multi-Language Support in GTS**

for tested models. More data-types and attributes would be needed to support rest of the data-types (e.g. VHDL Enumeration Types).

## 5.2 Simulator Independence

HDL models could be created in different application domains with different simulation needs. A test planning system restricted to a specific simulator is not very useful in such a situation. Therefore, our goal was to make the Goal Tree System work with as many simulators as possible without any modification requirements in the Goal Tree System code. Again the idea of abstraction was employed. The Goal Tree System was designed such that it treats the simulation steps as just a series of commands. For every test, it executes these commands sequentially without assuming any ordering or any particular simulator behavior. Though this approach seems to be too primitive and simple, it proved to be very powerful in dealing with diverse models and simulators. This is because most of the simulators support textual command based execution. Also this approach gives the independence of changing test configurations in complex ways, which might not be supported by the tool itself.

## 5.3 Goal Tree System Commands

Goal Tree System has a GUI which can be used to manage the construction of goal trees. It provides a user with commands for tree structure construction, node property editing, goal tree syntax checking and model simulation etc.

Listed below are the commands that the Goal Tree System provides for manipulating goal trees.

### 5.3.1 Commands for Project Organization

**Project Location:** Directory for storing files related to current project.

**Project Name:** Name of the current project

**HDL:** Hardware description language used to construct the models. Currently two hardware description languages are supported (VHDL and SystemC).

### 5.3.2 Commands for Tree Structure Construction

**Add:** Add a new node to the goal tree. The new node can be the parent, right sibling, left sibling, or rightmost child of the currently selected node.

**Delete:** Delete the current node or the sub tree rooted at the current node from the goal tree.

**Copy:** Copy the current node or the sub tree rooted at the current node onto the clipboard.

**Paste:** Insert the content of the clipboard to the right, to the left, or as the rightmost child of the current node.

**Collapse:** Collapse the sub tree rooted at the current node into a super node.

**Explode:** Explode a super node and bring back its underlying tree structure.

### 5.3.3 Commands for Goal Tree Syntax Checking

**Node property checking:** Check the intra-node properties of a node or all nodes in the tree. This is done using “Verify” button on the Node Property Editor.

**Tree structure checking:** Check the inter-node property for the whole goal tree. This is done using “Verify Goal Tree” menu item.

### 5.3.4 Commands for Goal Tree Library Maintenance

**New:** Start construction of a new goal tree.

**Open:** Open a goal tree from the goal tree library and display it as the current goal tree.

**Save:** Save the current goal tree to the goal tree library with its current filename and location.

**Save as:** Save the current goal tree to the goal tree library with a new filename and new location the user specifies.

### 5.3.5 Node Property Editor Commands

Figures 5.3 and 5.4 show the GUI of Node Property Editor. Node Property Editor can be opened by a right click on any of the nodes in goal tree. In this editor a user can enter the properties pertinent to the node of interest. The editor either prompts the user with a list of possible selections, or asks the user to type the information on the designated text fields. The set of menus provided by the Node Property Editor are described as follows.

**Type:** The node type can be set to a goal node, an AND operator node, an OR operator node, or a test group node as discussed in Section 4.2.

**Label, Name, and Description:** These fields allow the user to document the label and name of the node of interest and describe its functionality.

**Node Data Type and Length:** These fields are applicable if the node of interest is of type “Test Group”. Node data type is use to specify the type of data of the selected node. This type is dependent on the HDL being used. Length field is used to specify any additional attributes of the data-type (e.g. if the data type is `STD_LOGIC_VECTOR`, this field could be used to specify the length of vector).

**Adjustable requirement:** This pull-down menu lists all the system requirements of the application domain as well as their units, and allows the user to select the adjustable requirement, whose values are generated according to the given test strategy of the test group. Other requirements simply accept default values stored in the specification repository. The names of adjustable parameters are parsed from a file *adjustable\_parameters.list*, which will be described in next section.

**View File:** Clicking on this button shows the adjustable parameter file in use for current test-group. This file is same as the *adjustable\_parameters.list* file.

**Override File:** A different adjustable parameters file (different from *adjustable\_parameters.list*) can be chosen by clicking on this button. This can be used to override the default values for a particular test-group. If this option is not used, the default values are picked from file *adjustable\_parameters.list*.

**Test strategy and Control variables:** Depending on the primitive goal of the test group, the user is prompted to select the test strategy and enter the values of the control variables associated with the test strategy, which are required for the Goal Tree System to prepare the test values of the adjustable requirement.

**Comparator Parameters:** The choice “comparator parameter” and “value” can be used to choose the comparator parameter and its value for a test-group. The list of parameters is supplied by the user in a file *comparator\_params.txt*. This value will be printed in the file *comparator.txt* after every simulation run. The exact format of this output is shown in next section. The parameter name and value are expected to be used by the model to decide pass-fail results for a simulation. The project directory must contain file *comparator\_params.txt*, however it can be left blank if no comparator parameters are required.

## 5.4 Configuration and Specification files

The Goal Tree System relies on the use of files for configuring a model, specifying different attributes and results reporting. This section describes different files used by GTS.

### **adjustable\_parameters.list**

This file is used to specify all the adjustable parameters. An adjustable parameter can be either equivalent to a VHDL generic or it could be an input. This file should be present in the project directory. These adjustable parameters are specified one parameter per line with following format:

`<parameter name>@<parameter type>@<parameter default value>@<filename>`

E.g. `INT_GENERIC@generic@4321@testbench.vhd`

**Parameter Name:** It's the name of adjustable parameter. For generics the parameter name should be same as the name used in model. For input parameters, the name can be anything.

**Parameter Type:** The type of an adjustable parameter can be either “generic” or “input”.

**Parameter Default Value:** The default value of the adjustable parameter. For a particular test case, all parameters other than test-group parameter, will be assigned their default value.

**Filename:** Filename is the name of the file to which values of all adjustable parameters would be sent. For input parameters, the adjustable input parameter values would be externally generated and stored in a file and model is expected to read the values from that file. For generics, the filename should correspond to the file in which the generics are declared and initialized. For the adjustable generic (of current Test Group) the value derived using current test strategy is printed. For other generics (which are not adjustable

parameters for this Test Group), the default values are printed. The Goal Tree System parses the given file and substitutes the new values of these generics.

### **commands.cfg**

This file should contain the commands which are needed to simulate the model for a given tool and HDL. This file should be present in the project directory. For example if we are using Synopsis VSS on unix for VHDL, a sample command file would be:

```
cd my_project; vhdlan testEntity.vhd  
cd my_project; vhdsim TEST_BENCH <algo.con
```

For every command line, “cd <path>,” should be used in the front to change to directory (w.r.t. directory from where Goal Tree System GUI is running) where the command is to be executed.

In windows environment, the full path-name of the command executable file should be given.

### **Configuration Files**

Configuration files are the files, which contain value-assignment statements for generic parameters. Normally it's part of the test bench for a model. If the list of adjustable parameters contains generics as well, the corresponding configuration files should also be there. If these files are not in the project directory, then the relative path (relative to the directory from which GTS is running) should be given in the adjustable\_parameters.list file.

### **comparator\_param.txt**

This file is used to specify the list of comparator parameters being used by the model. The values picked from this file would be used to populate the comparator choice component in node property editor. An example comparator\_param.txt file is:

```
Range_Error_Within  
Detected_Range_Bin_Number
```

### **comparator.txt**

If a comparator parameter has been selected for a test-group, then that parameter name and value are printed to this file after every simulation run. This file can be read by model to decide test pass-fail result after simulation. Example contents of comparator.txt file are:

```
Detected_Range_Bin_Number 1004
```

### **gte\_sim\_result.txt**

This is the file in which MUT indicates results of a test (pass or fail). These results are used in search strategies and in preparing the report.

### **report.txt**

This file contains the comprehensive report of all the tests conducted by GTS in one run.

## **5.5 Reuse of Test-vectors**

The Goal Tree System stores the test values used for every simulation run of every test-group in different files. There is one file for every test-group. The filename containing stored test-vectors is <project name>\_<node name>\_<adjustable requirement name>.dat. These files can then be used to reuse test-vectors in a file-I/O strategy or using some other method. They can also be used as a quick reference to the adjustable parameter values applied for every test-group.

## **5.6 Additional Advantages of the Goal Tree System (GTS)**

In addition to providing extensive test generation facilities, multi-language support and simulator independence described in earlier sections, GTS provides the following facilities:

### **5.6.1 Platform Independence**

The Goal Tree System (GTS) has been designed to be platform independent. It can run on any operating system that has a simulator for HDL being used and support for JAVA run-time environment. GTS has been written using JAVA and care has been taken to ensure that there are no platform dependencies. Because it has been written in JAVA, running GTS on a platform doesn't even require recompiling the code. User can compile code on any OS (or get a precompiled code) and run it.

### **5.6.2 Project Organization Facilities**

GTS has been implemented with many project management features. Every project has a project directory, whose name/location can be specified by the user. There is a well defined set of configuration and command files, which are all stored in the project directory. All input and output files are also generated in the project directory.

There might be some situations when a user wants to generate a file in some different place (not in project directory). Facilities have been provided to override the default location of files for this purpose.

### **5.6.3 High Degree of Control**

Using GTS can provide very high degree of control in a model testing process. GTS could be used with not only generic parameters and input signals, but with any other parameter. These parameters can be configured as adjustable parameters with their type defined as input parameters in the goal tree. Then these parameters can be controlled like any other generic or input signal. As an example, consider a situation where GTS is being used to test a multiprocessor model in which number of processors is defined as a constant rather than a generic. If the number of processors needs to be varied in testing process, it can be done without changing the basic model structure by declaring it as input parameter.

## 6 Implementation Details

The Goal Tree System described in Chapter 5 has been implemented using Java programming language. This chapter discusses important implementation aspects of the Goal Tree System.

### 6.1 Motivation for using Java for the Goal Tree System Implementation

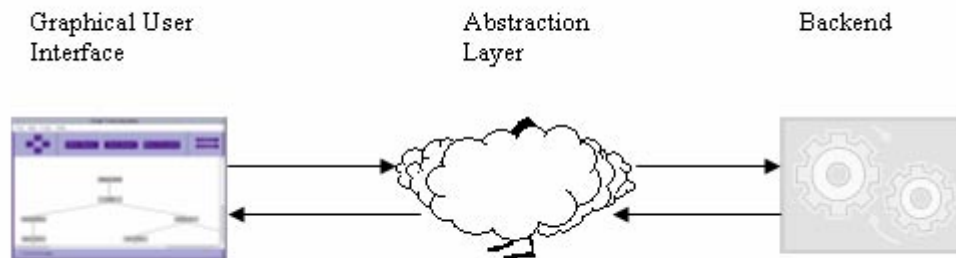
The Goal Tree System described in this thesis was written from scratch. The use of Java programming language for its implementation was motivated by the following considerations:

1. Java language features are not dependant on supported platforms. Java code runs on a virtual platform called Java Virtual Machine (JVM) [12]. The JVM handles all the platform specific details. This allows the behavior of Java code to remain consistent across platforms. Also because of the JVM, a program written in java can run on any supported platform without the need for recompilation.
2. The Goal Tree System needed a rich graphical user interface for the manipulation of goal trees. Java has extensive support for user interface construction.
3. The object-oriented nature of Java is ideal for the Language Abstraction Layer implementation. Object oriented language features like inheritance allows the language data-types to be added in phases without requiring any changes in existing code.
4. The Goal Tree System commands the simulator, it does not simulate the model itself. Therefore, slower speed of Java code execution compared to traditional languages like C and C++ is not a cause for concern.

## 6.2 Source Code Organization

From implementation point of view, the Goal Tree System can be broadly divided into three components. These components are the graphical user interface (GUI), the abstraction layer and the backend (Figure 6.1).

The graphical user interface provides facilities for goal tree construction, goal tree manipulation, test-plan documentation, specifying the attributes of test goals and test groups, controlling model simulation and for viewing test reports. Java provides two options for user interface implementation; one is the use of Abstract Windows Toolkit classes (AWT) and the other is the use of Swing classes [13]. Swing classes provide richer user interface functions, but the user interface constructed using Swing classes is slower than AWT. AWT was used for the Goal Tree System GUI because it had the required functionality and it is faster than Swing classes [13].



**Figure 6.1: Implementation Layers of the Goal Tree System**

The abstraction layer of Goal Tree System implements the LAL described in Section 3.1. More Java related details on the LAL implementation in the Goal Tree System have been provided in Section 6.3.

The backend of the Goal Tree System implements the test strategies and functions for execution of goal nodes. In the internal representation used for generic data-types (Section 5.1), we have use Java data-type *double* for storing value attribute of data-types. This allows a single implementation of test strategies across all the data-types.

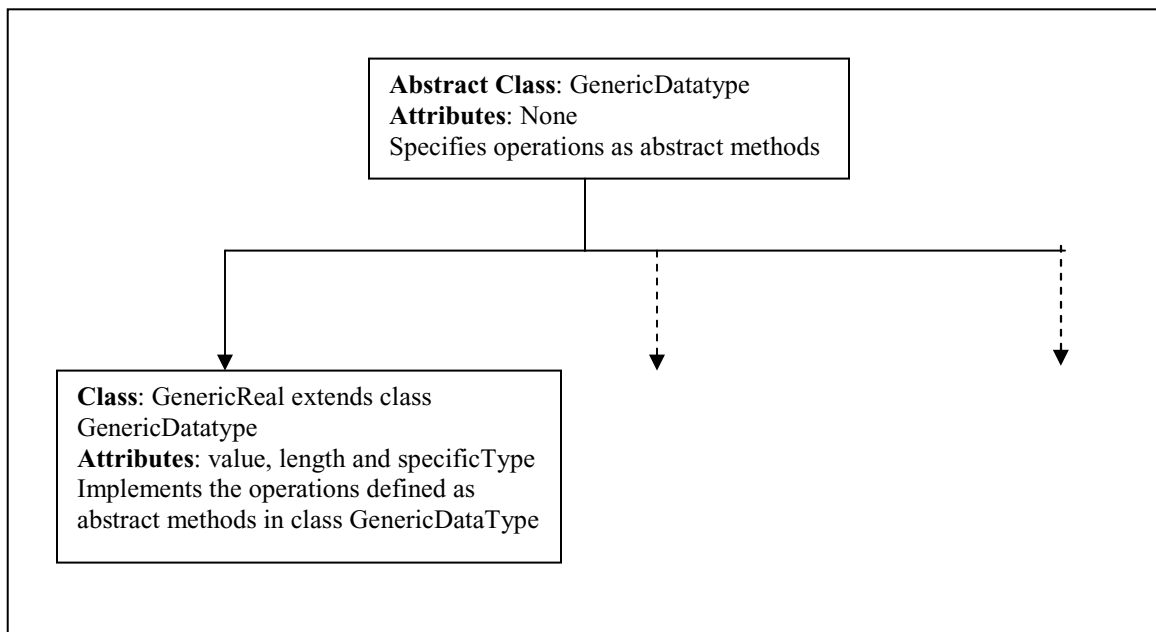
### 6.3 Node Class

In the Goal Tree System, the test plans are represented as goal trees. Construction of goal tree and its manipulation is the heart of the Goal Tree System. A class *Node* has been defined for representing the goals i.e. every node of the goal tree is an object of class *Node*. Like pre-defined data-types of a language, *Node* class has the data members and the functions to manipulate these data members. The data members of *Node* class can be classified into two types: data members for GUI manipulation and data members for Goal Tree Methodology implementation. Examples of data members used for GUI manipulation are: the parent node, the child node, the right sibling node and the left sibling nodes of the current node in goal tree and the (X,Y) co-ordinates to be used for drawing the node on screen. Examples of data-members used for Goal Tree Methodology implementation are: node type (Goal Node, a Test Group Node or an Operator Node), node data-type, adjustable parameter, test strategy etc. All the data-members are declared as private, they can be accessed only with the access functions provided in *Node* class. The source code external to *Node* class needs to use the access functions for reading or changing values of data members. With this approach, if representation of a data member needs to be changed in future, the code external to *Node* class need not be changed. This approach also prevents accidental change of data member values.

### 6.4 Generic Data-Types

As explained in Section 3.1, the backend of the Goal Tree System uses generic data types. The generic data-types are defined with appropriate attributes to ensure ability of translation from generic types to language specific types, and vice versa. If support for a new data-types is to be added to the LAL, all that is needed is identification (and if required definition) of corresponding generic types and writing the corresponding translation functions. In some instances, a generic-type might need to be extended to support the new data-type. To ensure code reuse, generic types need to be easily extendible with the requirement that the extension of a generic-type should not affect the support for existing data-types.

To ensure these characteristics, the class inheritance capabilities of Java have been used for generic-type definition. Using inheritance, we can create a general class that defines traits common to a set of related items [13]. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the Goal Tree System, the generic types have been defined as classes. The root of this class-hierarchy is the abstract class *GenericDatatype*. This class defines various translation operations as abstract methods. All other generic data-types extend from this type and add required attributes. If a generic-type needs to be extended to support a new language-specific data-type, we can create a new generic-type as a class which inherits from the type to be extended. This ensures non-disruptive support for new data-types in the Goal Tree System. The implementation of abstract methods defined in class *GenericDatatype* is provided in inherited classes. Figure 6.2 illustrates the class inheritance and Figure 6.3 shows the abstract class *GenericDatatype*.



**Figure 6.2: Generic Data-Type Class Hierarchy**

The variables in language independent source code of the GTS are of abstract type *GenericDatatype*. However, the value assigned to these variables is an object of the implemented generic-types (E.g. *GenericReal*). When the methods are called on these

variables, the Dynamic Method Dispatch [13] capabilities of Java language ensure that methods implemented in appropriate inherited class are invoked.

```
abstract class GenericDatatype
{
    //definition of operations for a data-type
    abstract void setHDLValue(Object valueToSet);
    abstract Object getHDLValue();
    abstract Object getToGenericValue();
    abstract Object convertToGenericValue(Object valueToConvert);
    abstract Object convertFromGenericValue(Object valueToConvert);
}

class GenericReal extends GenericDatatype
{
    private Object value;
    private String specificType;
    private int length;

    //implementation of abstract methods defined in parent class.....
```

**Figure 6.3: Source Code of Abstract Class GenericDatatype**

## 6.5 Goal Tree Representation and Traversal

In a goal tree, we have one root node (corresponding to the grand system goal) and any node in tree can have unlimited number of children. Therefore, a *general tree* representation has been used for the goal tree implementation. Each node of the tree is an object of *Node* class. Every node maintains a pointer to its parent, the left-most child, the right sibling and the left sibling. The tree traversal is equivalent to the post-order traversal in a tree data structure. The left-most sub tree of any node is traversed first, followed by the right sibling of left-most sub tree's root. This process is repeated for all the sub trees and then finally the parent is traversed.

## **7 GTS Application to the VHDL Model of SAR System**

### **7.1 Model Introduction**

The goal tree based testing methodology discussed in this thesis was used for testing a Synthetic Aperture Radar (SAR) processor model [2][11]. The original idea of goal tree based approach [2][11] (using only Real Number DSP models) was applied on this model. Therefore this model can be used to verify the correctness of the new generic and language independent Goal Tree System.

The model under test (MUT) is a VHDL real number model that simulates the behavior of the range compression processing algorithm of the SAR system. The test vector is a pulse of chirp signal returned by the targets in the swath. The MUT output is a one dimensional target range profile and the detected target ranges after post-filtering. The gold values were either the detected target ranges in meters or the range bin numbers of the expected targets, depending on what features of the model are being tested.

### **7.2 Tool Configuration and Goal Tree Construction**

First step in tool configuration is to specify the adjustable parameters of the SAR system model. This system has the following 10 adjustable parameters.

1. Bandwidth\_of\_Transmitted\_Signal
2. Carrier\_Frequency
3. Nominal\_Range
4. Pulse\_Repetition\_Rate
5. Pulse\_Width\_of\_Transmitted\_Signal
6. Resampling\_Frequency
7. Gaussian\_Noise\_Standard\_Deviation

8. Swath\_Width
9. Target\_Range
10. Target\_Size

The system model includes a C program which generates some secondary parameters from these adjustable parameters. A combination of all these parameters is then used as VHDL generics of MUT. Therefore, in tool configuration all the adjustable parameters are listed as input parameters. Following is the adjustable\_parameters.list file for this model (the first letter is line number)

```

1 Bandwidth_of_Transmitted_Signal@input@600@Bandwidth_of_Transmitted_Sig
nal.txt
2 Carrier_Frequency@input@33.56@Carrier_Frequency.txt
3 Nominal_Range@input@7260@Nominal_Range.txt
4 Pulse_Repetition_Rate@input@1.5@Pulse_Repetition_Rate.txt
5 Pulse_Width_of_Transmitted_Signal@input@30@Pulse_Width_of_Tran
mitted_Signal.txt
6 Resampling_Frequency@input@125@Resampling_Frequency.txt
7 Gaussian_Noise_Standard_Deviation@input@0@Gaussian_Noise_Standard_D
eviation.txt
8 Swath_Width@input@375@Swath_Width.txt
9 Target_Range@input@7260@Target_Range.txt
10 Target_Size@input@0@Target_Size.txt

```

**Figure 7.1: adjustable\_parameters.list File for SAR model**

Figure 7.2 is one of the goal trees of SAR system and Table 7.1 gives the features of primitive goals. Since the purpose of this testing is to verify correctness of GTS, the goal tree used was similar to the original goal tree [2][11].

The SAR system goal tree breaks the grand goal G into two smaller sub-goals: the footprint location goal G1 and the constraint goal G2. G1 has a footprint range error primitive goal G11 which evaluates the error of footprint range in meters produced by the MUT.

The constraint goal G2 is decomposed into range resolution goal G21 and noise sensitivity goal G22. The range resolution goal G21 is composed of two primitive goals G211 and G212. Starting at the nominal target location, the goal G211 applies the test

group TG211 to search downwards for the nearest location mapping to the same range bin number as mapped by the nominal location. Also, starting at the nominal location, the goal G212 applies the test group TG212 to search upwards for the farthest location mapping to the same range bin number. The difference between these two locations found above is then the range resolution of the MUT.

The noise sensitivity goal G22 is divided into three primitive goals G221, G222 and G223. They search for the maximum Gaussian noise that the MUT can tolerate at the nearest, the central and the farthest locations of the range bin mapped by the nominal location as found in G21. The nearest, central and farthest locations were found from tests conducted in G21.

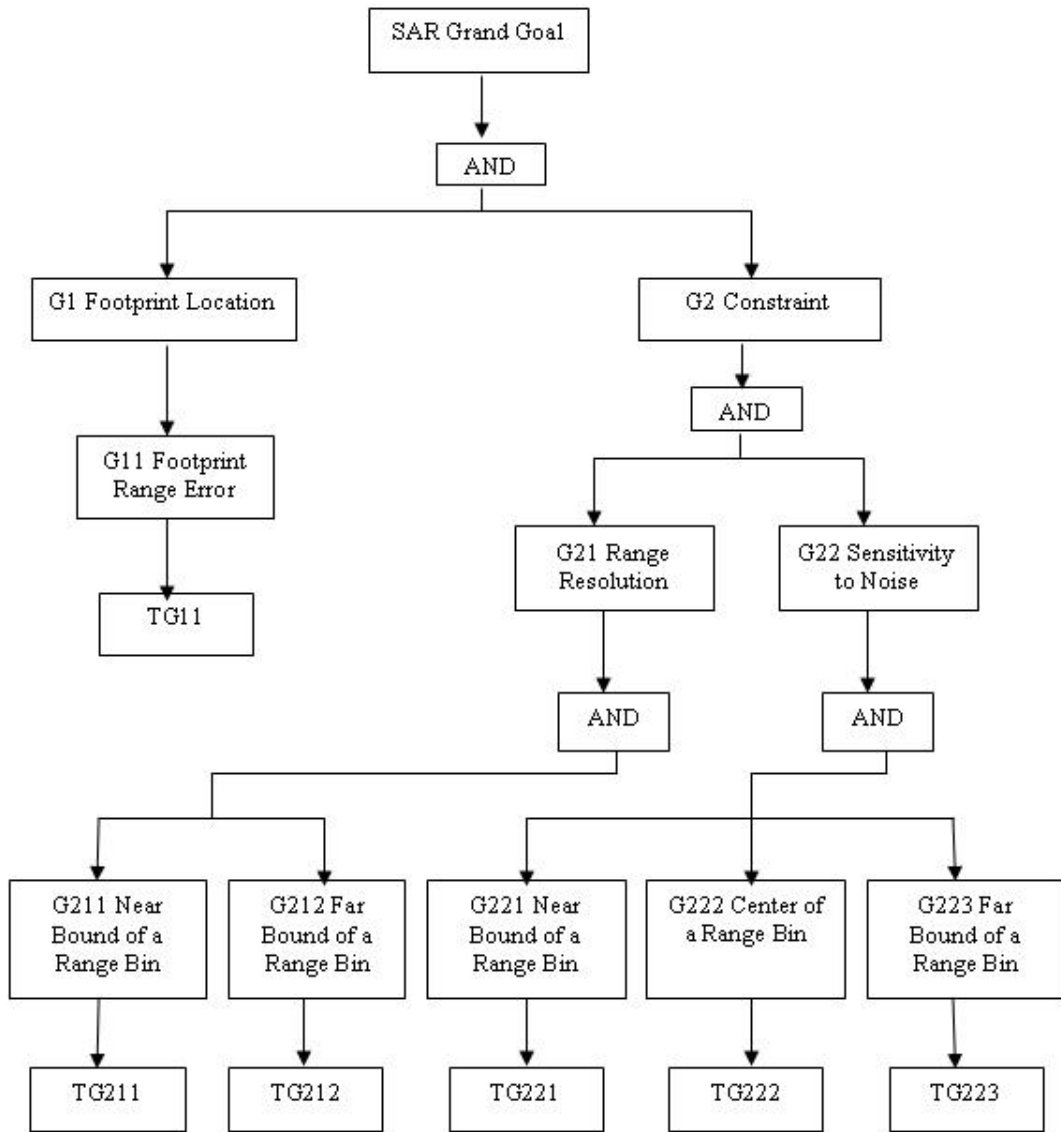


Figure 7.2: A Goal Tree of SAR system

Primitive Goal	Test Group	Adjustable Parameter	Test Strategy	Number of Test Cases	Comparator Parameter
G11	TG11	target range	Even division with endpoints (lower bound = 7072.5, upper bound = 7447.5)	fixed (9), given by test plan	target range (meters)
G211	TG211	target range	Geometric + Binary search (initial value = 7259.99, initial stepsize = 0.01, precision = 0.01, downwards)	$O(\log(\text{swath width} * \text{sampling rate}))$	range bin number (1004)
G212	TG212	target range	Geometric + Binary search (initial value = 7260.01, initial stepsize = 0.01, precision = 0.01, upwards)	$O(\log(\text{swath width} * \text{sampling rate}))$	range bin number (1004)
G221	TG221	gaussian noise standard deviation	Geometric + Binary search (initial value = 1, initial stepsize = 2, precision = 0.1, upwards, target range = 7259.79)	$O(\log(1/\text{precision}))$	target range (meters) (0.2)
G222	TG222	gaussian noise standard deviation	Geometric + Binary search (initial value = 1, initial stepsize = 2, precision = 0.01, upwards, target range = 7259.90)	$O(\log(1/\text{precision}))$	target range (meters) (0.2)
G223	TG223	gaussian noise standard deviation	Geometric + Binary search (initial value = 1, initial stepsize = 2, precision = 0.01, upwards, target range = 7260.01)	$O(\log(1/\text{precision}))$	target range (meters) (0.2)

**Table 7.1: SAR Primitive Goals**

### 7.3 Test Results

Table 7.2 shows the test results for footprint location test-group TG11. Table 7.3 shows the test results for range resolution test-groups TG211 and TG212. We see that near bound falls in the range (7259.79, 7259.795) and far bound falls in the range (7260.015, 7260.02). Table 7.4 shows the test results for noise sensitivity test-groups

TG221 and TG222 and TG223. We see that limiting values for these test groups fall in the ranges (0.5625, 0.625), (0.625, 6.6875) and (0.5625, 0.625) respectively.

A comparison of the test results shown in tables 7.2, 7.3 and 7.4 with the results obtained from old goal tree based simulation approach [2][11] shows that the similarity of results from both approaches. The results for test-groups TG11, TG221 and TG223 match perfectly, whereas the difference for test-groups TG211, TG212 and TG222 is very small. This verifies the correctness of new goal tree based approach and test-tool for at least a sub-domain of possible models.

Test No	Test Group TG11			
	Gold Range	Detected Range	Error	Result
1	7072.50	7072.59	0.09	Pass
2	7119.38	7119.47	0.09	Pass
3	7166.26	7166.36	0.10	Pass
4	7213.12	7213.24	0.12	Pass
5	7260.00	7259.90	-0.10	Pass
6	7306.88	7306.79	-0.09	Pass
7	7353.75	7353.67	-0.08	Pass
8	7400.62	7400.56	-0.06	Pass
9	7447.50	7447.44	-0.06	Pass

**Table 7.2: Results of Test Group TG11**

Test Group	Search Strategy	Test No	Target Range	Detected Range Bin	Result
TG211	Geometric	1	7259.99	1004	Pass
		2	7259.98	1004	Pass
		3	7259.96	1004	Pass
		4	7259.92	1004	Pass
		5	7259.84	1004	Pass
		6	7259.68	1003	Fail
	Binary	7	7259.76	1003	Fail
		8	7259.80	1004	Pass
		9	7259.78	1003	Fail
		10	7259.79	1003	Fail
		11	7259.795	1004	Pass
TG212	Geometric	1	7260.01	1004	Pass
		2	7260.02	1005	Fail
		3	7260.015	1004	Pass

**Table 7.3: Results of Test Group TG211 and TG212**

Test Group	Search Strategy	Test No.	Noise Level	No. of Detected Targets	Detected Range Error	Pass/Fail
TG221	Geometric	1	1	1	0.1	pass
		2	3	1	0.1	pass
		3	7	3	N/A	fail
	Binary	4	5	1	0.1	pass
		5	6	1	0.1	pass
		6	6.5	1	0.1	pass
		7	6.75	2	N/A	fail
		8	6.625	2	N/A	fail
		9	6.5625	1	0.1	pass
TG222	Geometric	1	1	1	0.00	pass
		2	3	1	0.00	pass
		3	7	4	N/A	fail
	Binary	4	5	1	0.00	pass
		5	6	1	0.0	pass
		6	6.5	1	0.0	pass
		7	6.75	3	N/A	fail
		8	6.625	1	0.0	pass
		9	6.6875	2	N/A	fail
TG223	Geometric	1	1	1	-0.11	pass
		2	3	1	-0.11	pass
		3	7	3	N/A	fail
	Binary	4	5	1	-0.11	pass
		5	6	1	0.12	pass
		6	6.5	1	0.12	pass
		7	6.75	3	N/A	fail
		8	6.625	2	N/A	fail
		9	6.5625	1	0.12	pass

**Table 7.4: Results of Test Group TG221 and TG222 and TG223**

## 8 GTS Application to the VHDL model of a GSM System

### 8.1 Model Introduction and Goal Tree Construction

A VHDL model<sup>1</sup> for the GSM communication system [5] was employed to study the various advantages achieved by using the features of the GTS. Configuring test benches [3] manually for the complete model was a very difficult task. GTS was used for this application as it provides all the necessary features such as geometric, arithmetic & binary search strategies, data enumeration, file enumeration and comparator facilities that are required to carry out the tests. The GTS was also used to visualize the goal tree structure for the GSM system. The complete goal tree, shown in Figure 8.1, includes all the test goals for the GSM model.

We have considered two basic types of test goals. The first major goal type was to test the system for hardware faults in the transmitter and the receiver modules. This goal was subdivided into two sub-goals. The first sub goal was to determine the fault coverage (G1). The second sub goal was to determine fault tolerance in the presence of hardware faults (G2). For both G1 and G2, faults were inserted in different modules (G11-G1N, G21-G2N). The most common faults such as line stuck at faults and bridging faults were modeled [7] and the faulty models were simulated and analyzed for fault coverage (G111). Based on the results, the faults were classified as hard or easy faults [8]. The error correction capability of the faulty models was also studied (TG211). In addition to these common faults, other functional faults that would correspond to an equivalent faulty circuit component, an incorrect mapping, an incorrect register assignment, etc.

---

<sup>1</sup> The VHDL model of GSM system and its goal tree was provided by Mr. Ramesh Govindarajulu.

were also modeled (G112, TG212). For each stuck at fault in G1, several random input vector sets were applied and the faults were tested (TG1111-TG111N).

The second goal type was to study the performance of the system by introducing different types of channel errors (G3). In addition to studying the specified system, we introduced alternative versions of some modules (G31, G32). The channel has been modeled as a white gaussian noisy channel. The user may specify the channel error rate and the tolerable system error rate as a probabilistic measure. There are many test groups for this goal type (TG311-TG31N), which can be classified into four cases.

In the first case, for a user specified tolerable system bit error rate, the maximum channel error rate that satisfies the specification is determined. The channel is assumed to produce single bit random errors. In the second case, the channel is assumed to produce multiple random burst errors. The duration of the random burst errors is varied within a specified range. The user specifies the range of the burst length. The goal is to find the maximum channel error rate such that the system specification is not violated. The third case is a slight variant of the second one, in which a single random burst error is assumed. The worst case burst length that satisfies the quality requirement is determined. In the fourth case, the user may specify a target confidence level for the results. In this type of goal, the tests are repeated until the specified confidence level is achieved. If the results involve a range of values, then a confidence interval precision may be specified.

The goal tree also has an additional branch used to obtain results with no faults in either the channel or the modules (G4).

The VHDL model is supplied with the test parameters from the GTS. The GUI of the GTS makes it easy for the user to select the options and provide specifications and test values. The comparator operation available in the GTS makes it easier to compare the current test results with the golden results.

Table 8.1 shows the description of the test goals mentioned above. The tool configuration for a VHDL model has already been given in Chapter 7, therefore it has not been given for this model.

## 8.2 Test Results

Table 8.2 shows the results for two of the fault coverage goals, Table 8.3 shows the results for two of the fault tolerance goals and Table 8.4 shows the results for one of the performance analysis goals.

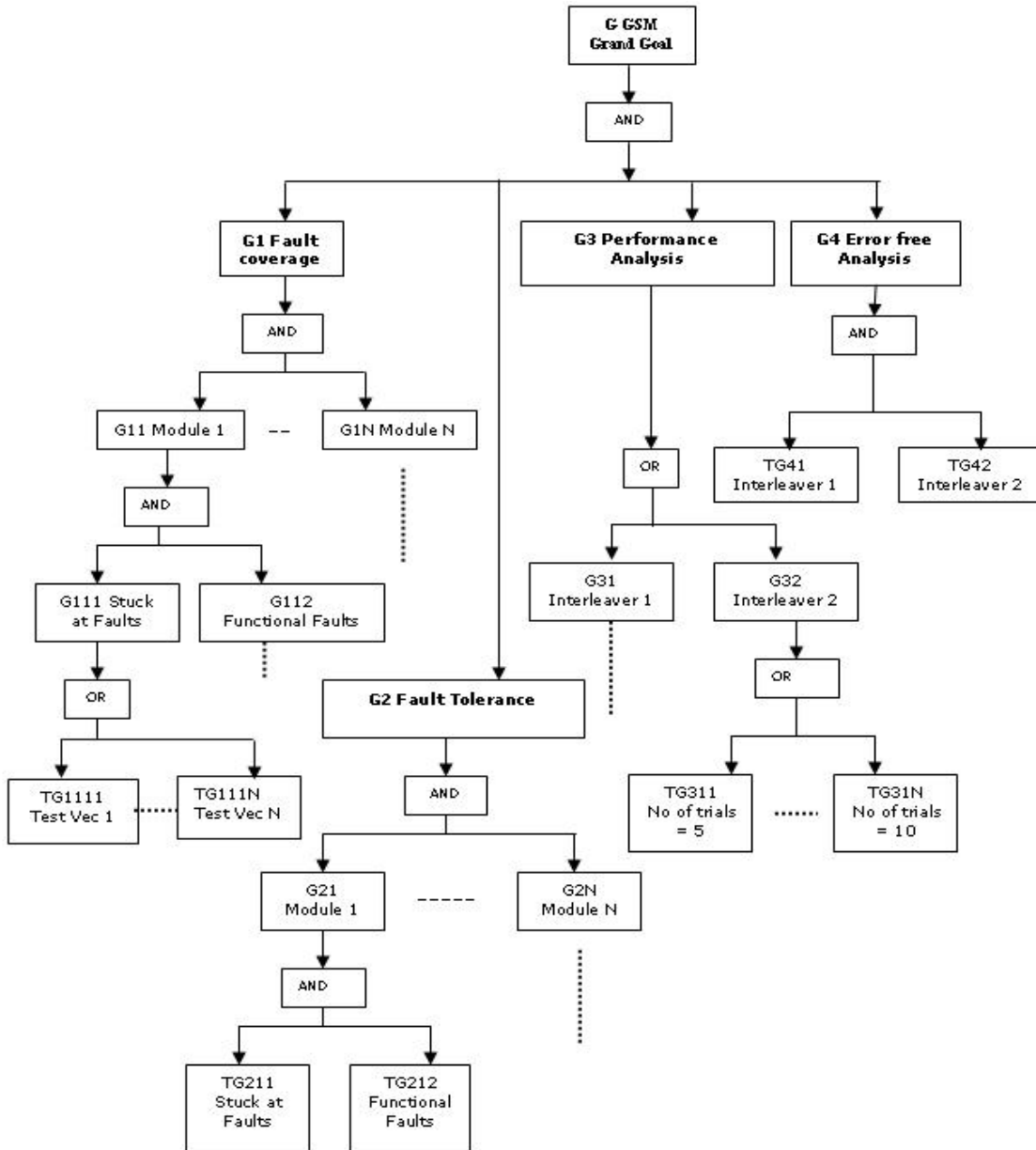


Figure 8.1: Goal Tree for VHDL Model of GSM System

Primitive Goal	Test Group	Adjustable Parameter(s)	Test Strategy	Number of Test cases
G111	TG1111	Length of the test bit vector	File Enumeration	variable
G111	TG1112	Length of the test bit vector	File Enumeration	variable
G21	TG211	Number of induced faults	Data Enumeration	variable
G21	TG212	Number of induced faults	Data Enumeration	variable
G31	TG311	Channel Error Probability	Geometric + Binary search	variable
G32	TG322	Channel Error Probability	Geometric + Binary search	variable
G4	TG41	Length of the test bit vector	File Enumeration	variable
G4	TG42	Length of the test bit vector, Pseudo random pattern	File Enumeration	variable

**Table 8.1 Description of the Goal Tree for the GSM System Modeled in VHDL**

Test Group	Search strategy	Test No.	Number of induced faults	Goal Fault coverage percentile	Model Fault coverage percentile	Result
TG1111	File Enumeration	1	180	75	71.25	Fail
		2	180	75	63.75	Fail
		3	180	75	81.5	Pass
TG1112	File Enumeration	1	180	75	52.75	Fail
		2	180	75	46.5	Fail
		3	180	75	72.75	Fail
		4	180	75	70	Fail
		5	180	75	76.75	Pass

**Table 8.2 Results for the Test Groups TG1111 and TG1112**

Test Group	Search strategy	Test No.	Number of induced faults	Tolerable System Error Probability in presence of Faults	Model System Error Probability in presence of Faults	Result
TG211 Stuck at Faults	Data Enumeration	1	5	0.008215	0.000012	Pass
		2	10	0.008215	0.000214	Pass
		3	15	0.008215	0.001946	Pass
		4	20	0.008215	0.007192	Pass
		5	25	0.008215	0.008826	Fail
TG212 Functional Faults	Data Enumeration	1	2	0.008215	0.002514	Pass
		2	4	0.008215	0.006936	Pass
		3	6	0.008215	0.138255	Fail
TG221 Stuck at Faults	Data Enumeration	1	5	0.008215	0.000539	Pass
		2	10	0.008215	0.001655	Pass
		3	15	0.008215	0.003320	Pass
		4	20	0.008215	0.009127	Fail
TG222 Functional Faults	Data Enumeration	1	2	0.008215	0.002754	Pass
		2	4	0.008215	0.013584	Fail

**Table 8.3 Results for the Test Groups TG21 and TG22**

Test Group	Search strategy	Test No.	Tolerable System Error Probability	Error Burst Length	Confidence Level %	Confidence Interval Precision %	Channel Error Probability	Model System Error Probability	Result
TG311	Geometric + Binary Search	1	0.007435	2-6	92	2	0.0001	0.000000	Pass
		2	0.007435	2-6	92	2	0.001	0.000000	Pass
		3	0.007435	2-6	92	2	0.01	0.000000	Pass
		4	0.007435	2-6	92	2	0.1	0.194970	Fail
		5	0.007435	2-6	92	2	0.055000	0.076035	Fail
		6	0.007435	2-6	92	2	0.032500	0.017160	Fail
		7	0.007435	2-6	92	2	0.021200	0.008279	Fail
		8	0.007435	2-6	92	2	0.015600	0.007988	Fail
		9	0.007435	2-6	92	2	0.012800	0.000592	Pass
		10	0.007435	2-6	92	2	0.014200	0.002071	Pass
		11	0.007435	2-6	92	2	0.014900	0.007628	Fail
		12	0.007435	2-6	92	2	0.014500	0.007509	Fail
		13	0.007435	2-6	92	2	0.014300	0.006213	Pass
		14	0.007435	2-6	92	2	0.014400	0.008133	Fail

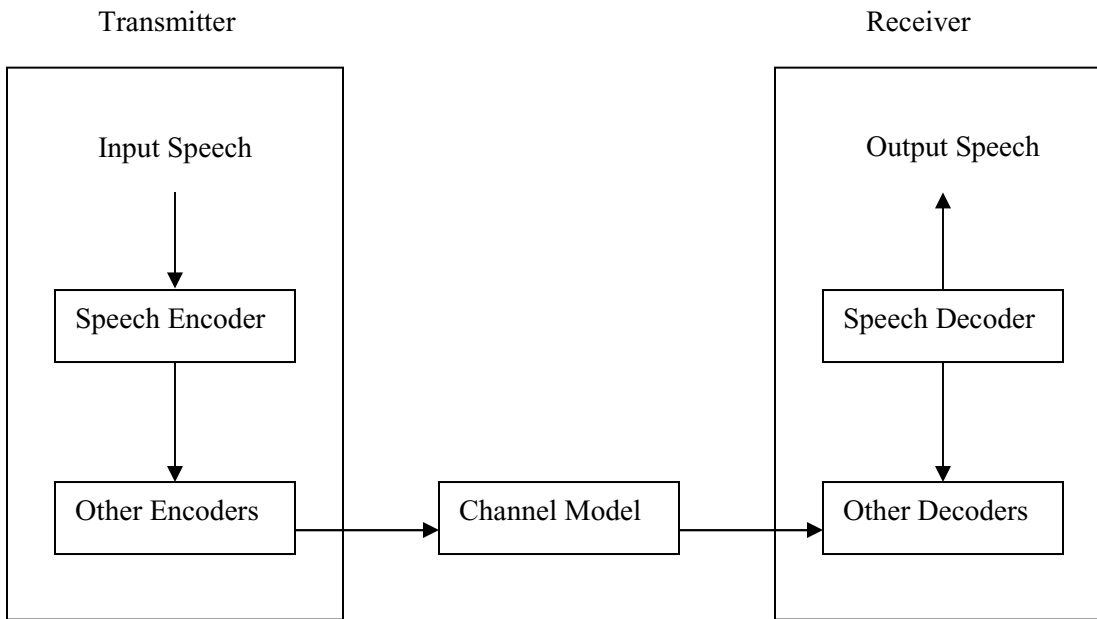
**Table 8.4 Results for the Test Group TG311**

## 9 GTS Application to a SystemC Model of a GSM System

### 9.1 Model Introduction

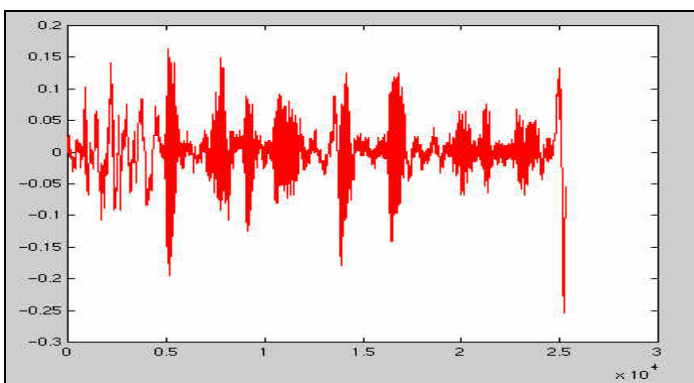
A SystemC based GSM model [10] was also tested with GTS to demonstrate the effectiveness of GTS across multiple hardware description languages. Figure 9.1 shows the high-level block diagram of the model under test (MUT). The MUT was a SystemC based model of GSM (Global System for Mobile communication). This consists of a series of encoders on the transmitter side, a model of the channel and a series of decoders on receiver side [10].

In this model, the input speech signal goes through a speech encoder, which converts it to a set of binary coefficients for a speech model.. This encoding is a lossy, but it makes sure that the deterioration on human perceptible frequencies is very minimal. The converted binary data then goes through a parity bit encoder, a convolutional encoder, an interleaving encoder, a packet format encoder and a differential encoder in that order. The resulting data is fed to a channel model. On the receiver side the data is decoded in reverse order to get the output speech signal.

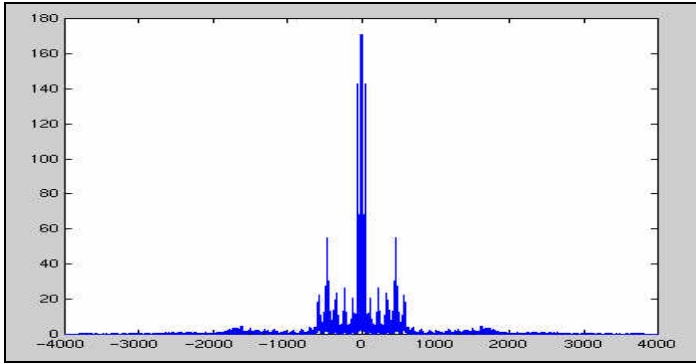


**Figure 9.1: High Level Block Diagram of GSM Model**

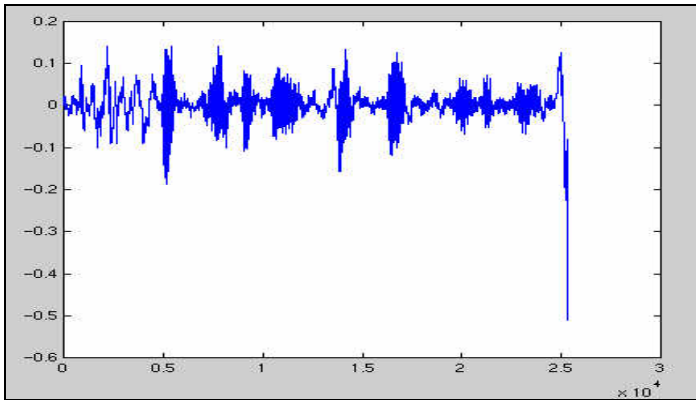
Plots 9.2-9.6 demonstrate the effectiveness of the GSM model in encoding, transmitting, receiving, and decoding a sound (.au) file of a male voice. Both time domain and frequency domain plots are shown.



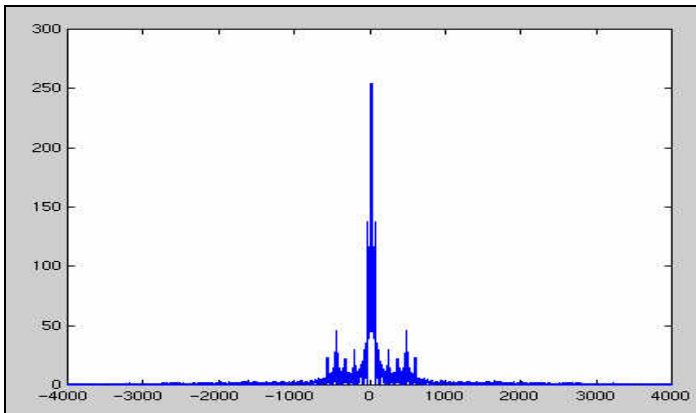
**Figure 9.2 Input File (Time Domain)**



**Figure 9.3 Input File (Freq. Domain)**



**Figure 9.4 Output File (Time Domain)**



**Figure 9.5 Output File (Freq. Domain)**

On magnifying these plots we can see some differences between input and output sound files in both time and frequency domains. So a comparator based on mean squared error between input and output files could be designed in both time and frequency domains. However, the tests shown in this section are based on time domain comparison because of its computational efficiency.

## 9.2 Tool Configuration and Goal Tree Construction

The first step in tool configuration is to specify the adjustable parameters of the GSM system model. This system has following adjustable parameters.

1. Input sound file
2. Output sound file
3. Channel error probability
4. Channel burst duration
5. Channel random seed

The test-bench of this model reads these parameters from different files. Therefore, in tool configuration all the adjustable parameters are listed as input parameters. Figure 9.6 shows the default *adjustable\_parameters.list* file for this model (the first letter is line number)

```
input_sound_file@input@song.au@input_sound.txt
output_sound_file@input@song_out.au@output_sound.txt
chan_error_prob@input@0@error_prob.txt
chan_burst_dur@input@1@chan_burst.txt
chan_random_seed@input@234@random_seed.txt
```

**Figure 9.6:** *adjustable\_parameters.list* File for GSM model

Figure 9.7 is one of the Goal Trees of GSM system and Table 9.1 gives the features of primitive goals. The system goal tree breaks the grand goal G into two smaller sub-goals: the model performance goal G1 and the error analysis goal G2. G2 is further divided into error probability primitive goal G21 which computes an acceptable upper limit on channel error probability in terms of the mean squared error between input and output sound files, and the error burst duration primitive goal G22 which evaluates the effect of variation of error burst duration for a fixed error probability.

As described above, the GSM model employs a speech encoder (Figure 9.1), which results in lossy compression. Therefore input and output sound files are expected to differ even with no channel errors. However, a correct implementation for speech encoder should make sure that the compression doesn't cause detectable deterioration of quality of sound. Test goal G1 is used for this purpose to test if speech encoder

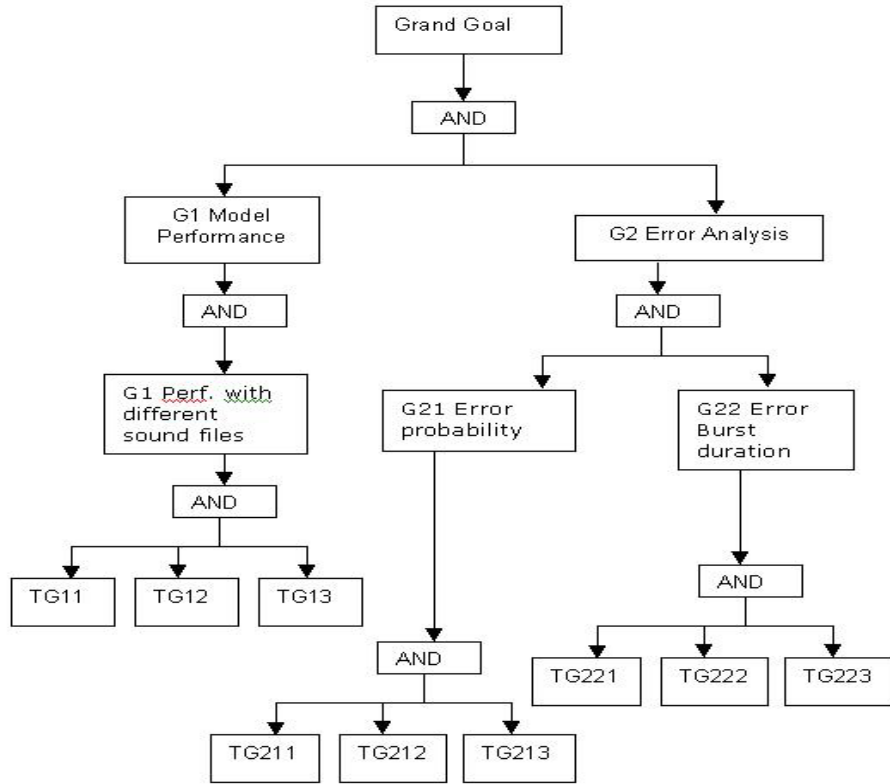
performance is acceptable in case of very low channel error probabilities. 4 different sound files are used for this purpose, they are

1. sin.au - which is a single frequency (1000 Hz) sine wave
2. sqr.au - which is a single frequency (1500 Hz) square wave
3. hello.au - which is a speech sample for a male voice
4. song.au - which is a speech sample for a female voice

The performance of speech encoder, for these 4 sound samples, was tested on low error probabilities of 0, 0.0001 and 0.0002.

G21 evaluates acceptable upper limit on channel error probability in terms of mean squared error between input and output sound files. Test groups TG211, TG212, TG213 find the range for acceptable mean squared error of 0.005, 0.03 and 0.05 respectively.

The error burst duration primitive goal G22 evaluates the effect of variation of error burst duration for a fixed error probability. The burst duration is the maximum duration of a burst. The actual burst length is determined probabilistically. Actual bits in error would depend on both channel error probability and error burst duration. Test groups TG221, TG222, TG223 conduct these tests for burst duration between 1 and 20 and for channel error probability 0, 0.0004 and 0.0007 respectively.



**Figure 9.7: A Goal Tree for SystemC Model of GSM System**

<b>Primitive Goal</b>	<b>Test Group</b>	<b>Adjustable Parameter</b>	<b>Test Strategy</b>	<b>Number of Test Cases</b>	<b>Comparator Parameter</b>
G1	TG11	Input sound file. Error probability used =0	File Enumeration	fixed (4), given by test plan	Mean squared error between input and output sound file (0.02)
G1	TG12	Input sound file. Error probability used =0.0001	File Enumeration	fixed (4), given by test plan	Mean squared error between input and output sound file (0.02)
G1	TG13	Input sound file. Error probability used =0.0002	File Enumeration	fixed (4), given by test plan	Mean squared error between input and output sound file (0.02)
G21	TG211	Channel error probability	Geometric + Binary search (initial value = 0, initial stepsize = 2, precision = 1, upwards)	Variable	Mean squared error between input and output sound file (0.005)
G21	TG212	Channel error probability	Geometric + Binary search (initial value = 0, initial stepsize = 2, precision = 1, upwards)	Variable	Mean squared error between input and output sound file (0.03)
G21	TG213	Channel error probability	Geometric + Binary search (initial value = 0, initial stepsize = 2, precision = 1, upwards)	Variable	Mean squared error between input and output sound file (0.05)
G22	TG221	Error burst duration for error probability = 0.0002	Even sampling with endpoints	fixed (5), given by test plan	Mean squared error between input and output sound file (0.02)
G22	TG222	Error burst duration for error probability = 0.0004	Even sampling with endpoints	fixed (5), given by test plan	Mean squared error between input and output sound file (0.02)
G22	TG223	Error burst duration for error probability = 0.0007	Even sampling with endpoints	fixed (5), given by test plan	Mean squared error between input and output sound file (0.02)

**Table 9.1: Description of Test Goals**

### 9.3 Test Results

Table 9.2 shows the test results for test-goal G1. The speech encoder performance is found to be reasonably satisfactory for most sound files. Since square wave is doesn't represent a real sound wave, the mean square error for this wave was very high.

Test No./ Input file	TG11 (Err Prob. 0)			TG12 (Err Prob. 0.0001)			TG13 (Err Prob. 0.0002)		
	Gold MSE	Model MSE	Result	Gold MSE	Model MSE	Result	Gold MSE	Model MSE	Result
1/sin.au	0.02	0	Pass	0.02	0	Pass	0.02	0	Pass
2/sqr.au	0.02	0.074545	Fail	0.02	0.207147	Fail	0.02	0.181958	Fail
3/song.au	0.02	0.000024	Pass	0.02	0.000036	Pass	0.02	0.000038	Pass
4/hello.au	0.02	0.000133	Pass	0.02	0.000305	Pass	0.02	0.000313	Pass
sin.au - 1KHz sine wave, sqr.au - 1.5KHz square wave, song.au - female voice, hello.au - male voice									

**Table 9.2: Test Results for Test Groups TG11, TG12 and TG13**

Table 9.3 shows the results for goal G21. As expected, the acceptable error probability increases with increase in tolerable mean squared error. The upper limit on tolerable error probability was found to be (0.0004-0.0005), (0.0013-0.0014) and (0.0018-0.0019) for acceptable mean squared error of 0.0005, 0.03 and 0.05 respectively.

Table 9.4 shows the results of analyzing the effect of increasing burst duration on model performance.

Test Group	Search Strategy	Test No	Error Probability	Goal MSE	Model MSE	Result
TG211	Geometric	1	0	0.005	2.4e-05	pass
		2	0.0003	0.005	7.6e-05	pass
		3	0.0009	0.005	0.019495	fail
	Binary	4	0.0006	0.005	0.007129	fail
		5	0.0004	0.005	0.00462	pass
		6	0.0005	0.005	0.005042	fail
TG212	Geometric	1	0	0.03	2.4e-05	pass
		2	0.0003	0.03	7.6e-05	pass
		3	0.0009	0.03	0.019495	pass
		4	0.0021	0.03	0.077816	fail
	Binary	5	0.0015	0.03	0.032448	fail
		6	0.0012	0.03	0.022237	pass
		7	0.0013	0.03	0.026066	pass
		8	0.0014	0.03	0.030043	fail
TG213	Geometric	1	0	0.05	2.4e-05	pass
		2	0.0003	0.05	7.6e-05	pass
		3	0.0009	0.05	0.019495	pass
		4	0.0021	0.05	0.077816	fail
	Binary	5	0.0015	0.05	0.032448	pass
		6	0.0018	0.05	0.044682	pass
		7	0.0019	0.05	0.065144	fail

**Table 9.3: Test Results for Test Groups TG211, TG212 and TG213**

Test No.	TG221 (Err Prob. 0.0002)			TG222 (Err Prob. 0.0004)			TG223 (Err Prob. 0.0007)		
	Burst Duration	Model MSE	Result	Burst Duration	Model MSE	Result	Burst Duration	Model MSE	Result
1	1	3.8e-05	Pass	1	0.00462	Pass	1	0.00832	Pass
2	5	3.8e-05	Pass	5	0.004619	Pass	5	0.014095	Pass
3	10	2.4e-05	Pass	10	2.5e-05	Pass	10	0.000455	Pass
4	15	4.4e-05	Pass	15	0.004621	Pass	15	0.015465	Pass
5	20	2.5e-05	Pass	20	2.5e-05	Pass	20	0.000462	Pass

**Table 9.4: Test Results for Test Groups TG221, TG222 and TG223 (Goal MSE -0.02)**

Looking at the test results from tables 9.2, we can conclude that performance of speech encoder is satisfactory for human voice. Also tables 9.3 indicate the deterioration of output voice signal with increase in error probability. However, table 9.4 shows that burst duration does not significantly affect the performance of the GSM system.

## 10 Conclusions and Future Work

The strategies presented in this thesis and their implementation using the Goal Tree System, demonstrate the feasibility of developing multi-language test automation tools. A high degree of language independence enables the use of same tool with models developed in multiple HDLs. This results in faster and cheaper adoption of new HDLs. Simulator independence allows one to separate model simulation and test planning and to use the most suitable simulator. The ideas presented in this thesis are generic enough and can be applied to other tools with some extensions.

Goal tree based test planning provides a natural way of organizing testing activities. This approach can be used in a wide variety of application domains. Strategies like code-coverage tests and fault analysis [8] can be used to test effectiveness of a test plan and to check corner cases.

There is a wide range of future research possibilities

- 1) The implementation of the language abstraction layer and the language independent layer in current GTS can be extended to support more data-types of VHDL and SystemC. Also they can be extended to support other HDLs like Verilog.
- 2) Extensions of search strategies and operator nodes (e.g. IF-THEN-ELSE nodes, conditional loop nodes) in GTS.
- 3) Enabling configuration of more than one adjustable parameter in a test group.
- 4) Integration of GTS with code-coverage and other relevant tools to provide evaluation of constructed goal trees.
- 5) Studying the applications of language abstraction approach on other types of test and design tools e.g. automatic test-pattern generators, test-bench generators.

## 11 Bibliography

[1] J.R. Armstrong and F.G. Gray, *VHDL Design Representation and Synthesis*, Prentice-Hall, Upper Saddle River, NJ, 2000.

[2] M.W. Lin, J.R. Armstrong and F.G. Gray, "A Goal Tree Based High-Level Test Planning System For DSP Real Number Models," *Proc. International Test Conference*, 1998, pp. 1000-1009.

[3] J. Bergeron, *Writing Test-benches: Functional Verification of HDL Models*, Kluwer Academic Publishers, Norwell, MA, 2000.

[4] J.R. Armstrong, F.G. Gray, and M.W. Lin, "VHDL Modeling and Model Testing for DSP Applications", *IEEE Transactions on Industrial Electronics*, Feb. 1999, pp. 13-22.

[5] L. Harte, R. Levine and G. Livingston, *GSM Super Phones*, McGraw Hill Publications, Hightstown, NJ, 1999.

[6] D. Schiff, R.B. D'Agostino, *Practical Engineering Statistics*, John Wiley & Sons Inc, New York, NY, 1996.

[7] K.M. Butler and M.R. Mercer, "The Influences of Fault Type and Topology on Fault Model Performance and the implications to Test and Testable Design", *Proc. 27<sup>th</sup> Design Automation Conference*, pp. 673-678, June 1990.

- [8] P.C. Ward and J.R. Armstrong. “Behavioral Fault Simulation in VHDL”, *Proc. 27<sup>th</sup> Design Automation Conference*, pp. 587-593, June 1990.
- [9] G.A. Shaw, *RASSP Benchmark 1 - Technical Description - Synthetic Aperture Radar Processor*, Lincoln Laboratory, MIT, January 1994.
- [10] A. Varma, “*Modeling and Synthesis with SystemC*”, masters thesis, Dept. Electrical and Computer Eng., Virginia Tech., Blacksburg, VA, 2001.
- [11] M.W. Lin, “*A Test Planning System for Functional Validation of VHDL DSP Models*”, doctoral dissertation, Dept. Electrical and Computer Eng., Virginia Tech., Blacksburg, VA, 1998.
- [12] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification*, Addison-Wesley Publication Company, Boston, MA, 1999.
- [13] P. Naughton and H. Schildt, *Java™ 2: The Complete Reference*, Osborne McGraw-Hill, New York, NY, 2000.

## **Vita**

Rajneesh Mahajan was born and brought up in the hilly state of Himachal Pradesh in India. He graduated with a Bachelor of Engineering degree in Electronics Engineering from the Visvesvaraya Regional College of Engineering, Nagpur in 1998. After working in Citicorp Overseas Software Limited for 2 years, he decided to pursue his higher studies at Virginia Tech in 2000. He graduated with a Master of Science degree in Computer Engineering from Virginia Tech in 2002. His hobbies are music and war history. His technical interests include hardware design, verification and computer networks.