C H A P T E R   4

TEST PLANNING FRAMEWORK AND GOAL TREE SYSTEM

This chapter describes the concepts of a test planning framework which focuses on functional validation of the numerical characteristics of DSP systems whose system mathematics is modeled. It shows how a goal tree can be used to represent a test plan, and addresses the development of the goal tree system which manipulates goal trees.

4.1 Test Planning Framework

A *test plan* is a document which organizes the system requirements in terms of how these requirements will be tested [1, 2]. It defines the elements required in test planning such as *test goals*, *test groups*, *test cases*, and *test oracles*.

The development of a test plan is a top-down partitioning process. By referring to the specifications, it starts with the desired test goal and breaks that goal into simpler subgoals. These subgoals are in turn subdivided until primitive goals are reached [3], where primitive goals are goals to be directly met by applying test groups.

A test case is a test scenario for single execution, in which the system requirements are fully specified. It maps to a VHDL configuration declaration of the test bench.

A test group defines a series of organized test cases which evaluate a certain primitive goal iteratively. It constrains all of the system requirements except one (termed the adjustable requirement) whose value is allowed to vary based on a given strategy. The length of a test group is the number of test cases defined in the test group. Some test groups have fixed lengths. For others, their lengths are dynamically determined based on the test results during simulation. A test group maps to a partially specified configuration declaration of the test bench [47, 48]. It is possible that two or more test groups are associated with a primitive goal.

A test oracle is a rule which describes what the correct response to input test cases should be [60]. It is used by the comparator of the test bench to determine the success or failure of a test.

4.2 Goal Tree Representation

A test plan is represented by a data structure called a goal tree $\mathcal{G}(\mathcal{N}, \mathcal{E})$ which consists of a node set $\mathcal{N}$ and an edge set $\mathcal{E}$, where the node set $\mathcal{N}$ is partitioned into *goal nodes G, test group nodes TG*, and *operator nodes O*, and the directed edge set $\mathcal{E}$ indicates the incidences between two nodes.

The goal nodes *G* represent the test goals (including subgoals and primitive goals) of a test plan. The root of a goal tree is a goal node which corresponds to the grand goal of a test plan.

The test group nodes *TG* represent the test groups defined by the primitive goals of a test plan. They are the leaves of a goal tree. A test group node is an oversized node which comprises a sequence of slots representing test cases.

A test case *T* is embraced in a test group node in the goal tree representation. It represents an individual test in the test plan.

An operator node *O* can be either a logical *AND* or *OR* operator. It decomposes a goal into subgoals, and helps a primitive goal define two or more test groups in and/or fashion. All children of an *AND* operator must be satisfied to fulfill the objective of its parent node. On the other hand, fulfillment of any child of an *OR* node suffices.

The goal tree representation of a test plan provides necessary information to configure the test bench. Each test defined in the goal tree maps to a configuration declaration of the structural test bench. Since the same test bench configuration may serve several goals, shared test groups may occur.

4.3 Primitive Goals and Test Strategies

The test planning framework focuses on functional testing for arithmetic operations of DSP models which primarily apply to real or complex numbers along the data paths. The primitive goals are classified as *confirmation goals* or *search goals*.

4.3.1 Confirmation Goals

Testing all values from the space of the adjustable requirement (called the *test space*) is usually impractical [50]. It is desirable to limit the number of test values [67]. 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. The following example illustrates the confirmation goal.

Example 1. To test the specification that "the low-pass filter $H(\Omega)$ has a passband Ω_c with magnitude $1-\delta_1$" in time domain, it is required that the magnitude response in the passband satisfy [12]:

$$1 \geq |H(\Omega)| \geq 1 - \delta_1,\ |\ \Omega\ | \leq \Omega_c$$

A confirmation goal can be set up so that the adjustable requirement is the frequency Ω of the input signal, the test space is $[0,\ \Omega_c]$[1], and the test oracle is $1 \geq |H(\Omega)| \geq 1 - \delta_1,\ \forall\ \Omega \in [0,\ \Omega_c]$. ♦

Four possible sampling strategies that can be used for confirmation goals are as follows.

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 [67, 68], i.e., a set of n points $\{P_1,\ P_2,\ \ldots,\ P_n\}$, where P_1 and P_n are the minimum and maximum values of the test space, respectively, and $P_{i+1} - P_i = (P_n - P_1) / (n$

[1] Only the non-negative frequencies are considered here since the magnitude response for a real filter is an even function, i.e., $|H_1(\Omega)| = |H_1(-\Omega)|$

– 1), i = 1, 2, …, n–1. Figure 4.1 depicts the selection of 4 samples using this test strategy.

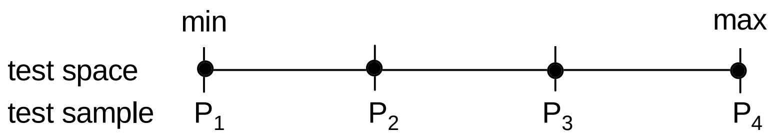


Figure 4.1 Even Sampling with Endpoints

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.

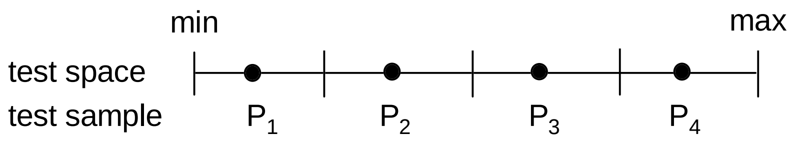


Figure 4.2 Even Sampling without Endpoints

3) **Random**. A sequence of random values are chosen from the test space according to a given probability distribution, for instance, uniform distribution or normal distribution [69].

4) **Enumeration**. All test values are enumerated individually.

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.

Table 4.1 Test Strategies and Control Variables

Test Strategy	**Control Variables**
even sampling with endpoints	test space, no. of tests
even sampling without endpoints	test space, no. of tests
random	test space, no. of tests, probability density function
enumeration	none
arithmetic search	test space, initial value, increment, direction
geometric search	test space, initial value, initial increment, direction
binary search	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

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. The following example illustrates the search goal.

Example 2. Continued from Example 1. Suppose that it has been ascertained that $H(\Omega)$ is a Butterworth low pass filter [12], whose magnitude response is monotonically decreasing for all frequencies. We can set up a search goal by searching for an input frequency Ω_δ such that $|H(\Omega_\delta)| = 1 - \delta_1$, and comparing Ω_δ with Ω_c. In this case, the test space becomes $[0, \infty)$, and the test oracle is $\Omega_\delta \geq \Omega_c$. ♦

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.

1) **Arithmetic Search**. An arithmetic sequence is chosen to form the test values, i.e., a sequence $a, a \pm d, a \pm 2d, a \pm 3d, \ldots$, 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 $O(len / d)$, where len is the length of the test space.

2) **Geometric Search**. An offset geometric sequence of ratio 2 is applied to choose the test values, i.e., a sequence $a, a \pm 2^0d, a \pm 2^1d, a \pm 2^2d, a \pm 2^3d$, 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 $O(log(len / d))$, where 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.

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 [7, 8].

```
        // Assume the MUT passes the test with the initial lower bound
        // and fails the test with the initial upper bound.
        // Initialization
        initialize LOWER, UPPER, and PRECISION
Loop:                                   // Binary Search
        PIVOT = (LOWER + UPPER) / 2
        if MUT passes the tester then
                LOWER = PIVOT           // UPPER = PIVOT in the other case
        else
                UPPER = PIVOT           // LOWER = PIVOT in the other case
        end if
        if |UPPER - LOWER| < PRECISION then
                exit binary search
        end if
        goto Loop
```

Figure 4.3 Algorithm for Binary Search

Figure 4.3 outlines the algorithm of the binary search strategy, where LOWER and UPPER stand for the lower and upper bounds of the extreme value of the adjustable requirement, PRECISION is the precision, and PIVOT is the test value. The algorithm below assumes that the MUT passes the test with the lower bound and fails the test with the upper bound. If the MUT fails the test with the lower bound and passes the test with the upper bound, the algorithm needs to be modified slightly as underlined in Figure 4.3.

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.

The control variables used in the search strategies have been listed in Table 4.1.

4.4 Advantages of the Test Planning Framework

The test planning framework offers several advantages.

1) The hierarchical structure of a goal tree helps understand the large number of requirements to be tested, and helps define test goals and test cases.

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. The efficiency of the whole testing process is therefore significantly enhanced.

3) Use of the *OR* operator allows coexistence of equally effective test goals and test groups defined in the test plan. An associated pointer to the selected child indicates the branch to be taken when traversing the goal tree during simulation.

4) The test plan can be easily modified by pruning and grafting the goal tree.

4.5 Goal Tree System

As discussed in the previous sections of this chapter, test plans can be represented by goal trees. It is desirable to have a mechanism which supports the creation and processing of machine-readable goal trees to enhance the test planning process.

Figure 4.4 shows the goal tree system which manipulates goal trees and controls the testing process. One enters information on test plans into the goal tree editor, and goal trees are constructed accordingly and can be stored in the goal tree library. Existing goal trees can also be retrieved from the goal tree library and modification can be made if needed. The default primary requirements values are received from the specification repository. The test vector database provides the user with the available test files for a file I/O strategy, which will be explained shortly. The goal tree system engine traverses the goal trees depth-first and, whenever a test group node is reached, produces a set of values for the primary requirements which are sent to the requirements interface. It also commands the test bench generator to create configuration declarations for test benches and initiates a new test by simulation. Based on the simulation results, the goal tree system engine determines whether a new test should be initiated for this test group. If not, this test group is terminated and the tree traversal process continues.

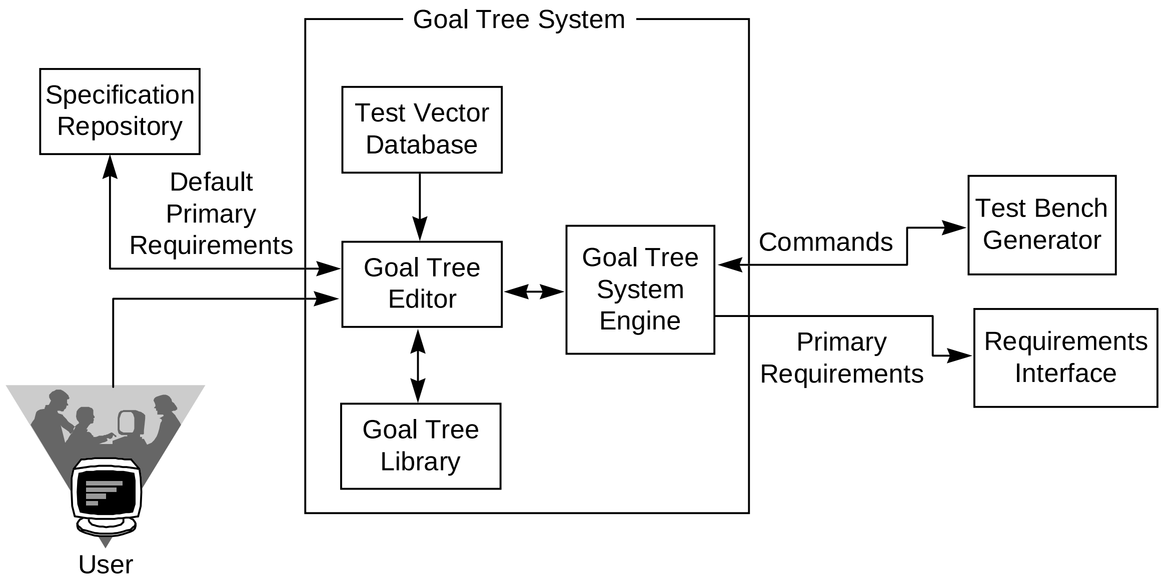


Figure 4.4 Goal Tree System

4.5.1 Goal Tree Traversal

Figure 4.5 illustrates how the goal tree system traverses a portion of a goal tree depth-first using four types of pointers a node has: the pointer to the first child node, the pointer to the parent node, the pointer to the right sibling node, and the pointer to the selected child, applicable if the node of interest is an *OR* node. When an *AND* node (say, Node A) is reached, all of its children (Nodes B, C, and D) have to be traversed. On the other hand, when an *OR* node (say, Node C) is reached, the subtree rooted at the selected child (Node G) is traversed, and those rooted at the other children (Nodes F and H) are skipped. Designation of the selected children for *OR* nodes can be done easily from the Node Property Editor which is described in 5.5.4.

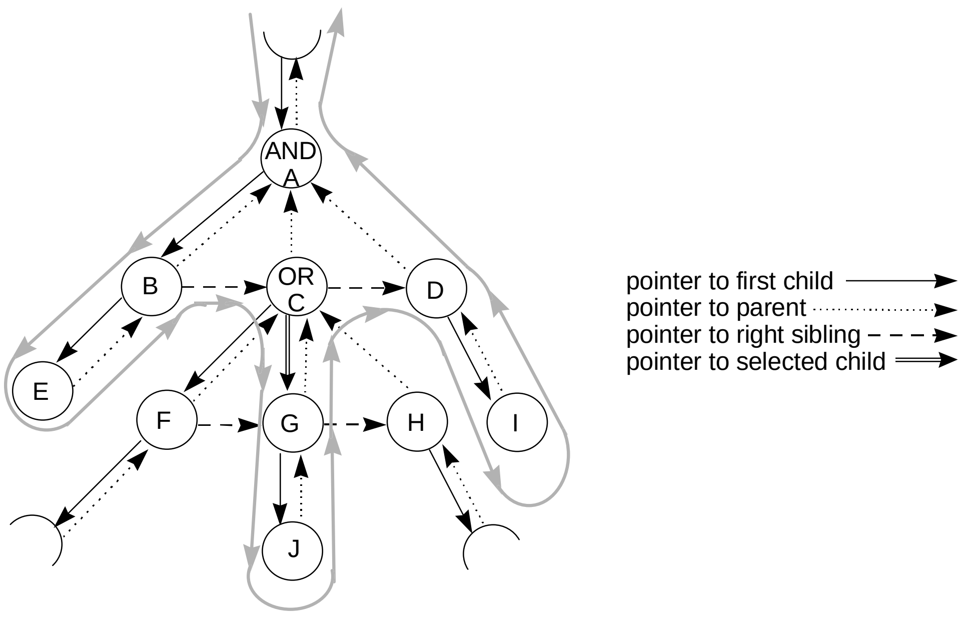


Figure 4.5 Traversal of a Goal Tree

4.5.2 Reuse of Test Vectors

In the test planning framework, a test group defines a sequence of tests for which the test vectors are generated. This involves iteratively simulating the stimulus generator and is thus expensive. It would save simulation time if the generated test vectors can be reused whenever and wherever applicable. The test planning interface in the RASSP test bench generator did not have a mechanism to accommodate this feature. To cope with this, a new test strategy called file I/O has been added to the test planning framework in addition to the sampling strategies and search strategies discussed in Section 4.3. The file I/O strategy allows a test group to sequence through a list of enumerated test files. A difference from the other test strategies is that the test vectors are not generated by simulating the stimulus generator. They are read from the test files through file I/O and sent to the model under test. Another difference is that other test strategies vary the values of the adjustable requirements from test to test, but the file I/O strategy has no control over the system requirements.

The file I/O strategy offers the following advantages:

1) Sharing test groups with other primitive goals. If one test group can be applied to two primitive goals in a goal tree, the primitive goal which is traversed later can simply define a test group using the file I/O strategy so as to reuse the test vectors generated earlier for the other primitive goal.

2) Replaying test vectors generated for the same primitive goals. A test plan needs to be executed many times to correct the design errors between the development phase and testing phase. Also a test plan needs to be run periodically in the maintenance phase. Both facts suggest that a copy of the test

vectors be generated and saved once and reused afterwards. Therefore, we can associate in *OR* fashion each normal test group[2] with a counterpart test group where the test vectors generated for the normal test group are listed using file I/O strategy. In this way, when the normal test groups are selected for traversal, the test vectors are generated by simulation; otherwise, the pregenerated test vectors are reused.

3) In addition to reusing all test vectors generated for a test group, the file I/O strategy also allows one to collect and reuse individual test vector files generated in different test groups.

The file I/O strategy has been implemented in the goal tree system. The SAR goal tree, which will be discussed in Section 4.6, fully utilizes these advantages. The file I/O strategy proved to be extremely useful in the post testing quality analysis, which is detailed in Section 6.3.

4.5.3 Goal Tree Editor

A graphical user interface called the goal tree editor has been implemented in the X Window System using the xlib, Xt intrinsics, and Motif libraries [61] on the Solaris platform as shown in Figure 4.6. It provides the user with commands for tree structure construction, node property editing, goal tree syntax checking, and goal tree library

[2] Test group that does not use the file I/O strategy

maintenance. The commands can be easily invoked by clicking the mouse on the associated buttons or the nested pull-down menus. Listed below are the commands that the goal tree editor provides for manipulating goal trees.

1) 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 subtree rooted at the current node from the goal tree.

   Cut: Remove the current node or the subtree rooted at the current node from the goal tree and place it on the clipboard.

   Copy: Copy the current node or the subtree 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.

   Insert file: Insert a goal tree file into the current goal tree to the right, to the left, or as the rightmost child of the current node.

   Collapse: Collapse the subtree rooted at the current node into a super node.

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

2) Commands for node property editing

   These will be discussed in the next section.

3) Commands for goal tree syntax checking

   Node property checking: Check the intranode properties of a node or all nodes in the tree.

   Tree structure checking: Check the internode property for the whole goal tree.

4) Commands for goal tree library maintenance

   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.

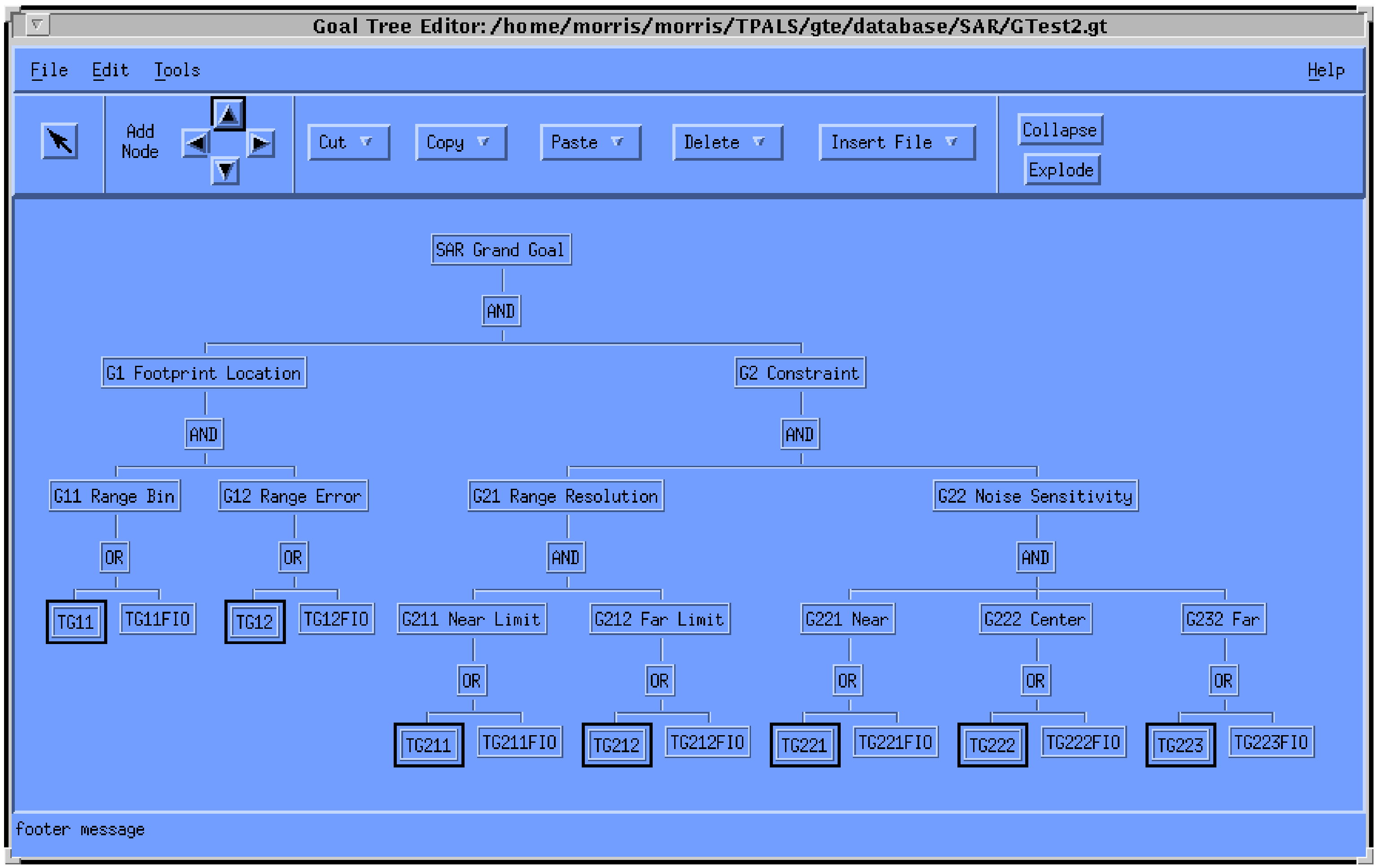


Figure 4.6 Goal Tree Editor

4.5.4 Node Property Editor

Figure 4.7 shows a Motif based X Window graphical user interface called the Node Property Editor. When the user doubly clicks on a node of the goal tree from the goal tree editor, the Node Property Editor pops up, from which the 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.

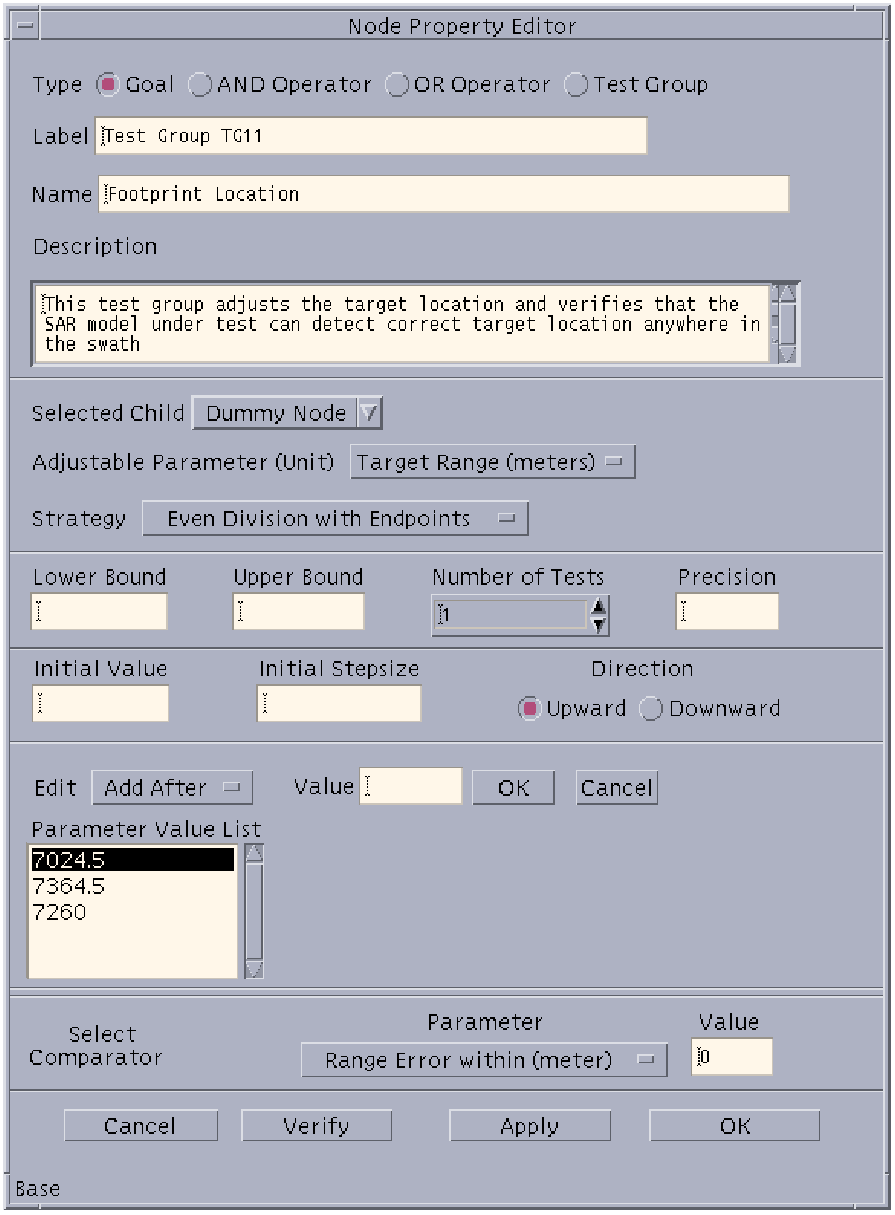
Type: The node type can be set to a goal node, an AND operator node, an OR operator node, or a test group node as discussed in Section 4.2.

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

Selected child: This menu shows up if the node of interest is an OR operator. The children of the node are dynamically listed and the user can designate its selected child.

The following menus or input fields show up and are applicable if the node of interest is of the test group type.

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.



Node Property Editor

Type [✓ Goal] AND Operator OR Operator Test Group

Label Test Group TG11

Name Footprint Location

Description

This test group adjusts the target location and verifies that the SAR model under test can detect correct target location anywhere in the swath

Selected Child Dummy Node

Adjustable Parameter (Unit) Target Range (meters)

Strategy Even Division with Endpoints

Lower Bound	Upper Bound	Number of Tests	Precision
		1	

Initial Value	Initial Stepsize	Direction
		[✓ Upward] Downward

Edit Add After Value OK Cancel

Parameter Value List

```
7024.5
7364.5
7260
```

Select Comparator

Parameter Range Error within (meter)

Value 0

Figure 4.7 Node Property Editor

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 parameter and **Acceptable error**: The user can select a comparator parameter and provide an acceptable error according to the test oracles. The comparator parameter is the metric against which the MUT response and the expect response are compared. The comparator is so configured that if the difference is greater than the acceptable error, the test is a failure. Otherwise, it is a success. Let's take SAR as an example. To test the mapping from target location to range bin which is an integer number, the comparator parameter is set to the *range bin number* and the acceptable error is set to 0, which means that the range bin number should be an exact match. To test the footprint location of the target, the comparator parameter is set to the *detected range in meters* and the acceptable error is set to a real number, say, 0.15 meters.

4.6 SAR Test Plan

Figure 4.8 shows a test plan for SAR using the goal tree representation. This goal tree breaks the grand goal G into two smaller subgoals: the footprint location goal G_1 and the constraint goal G_2. The footprint location goal G_1 is further divided into the footprint range bin primitive goal G_{11} which evaluates whether the MUT gives the correct footprint location in terms of range bin number, and the footprint range error primitive goal G_{12} which evaluates the error of footprint range in meters produced by the MUT.

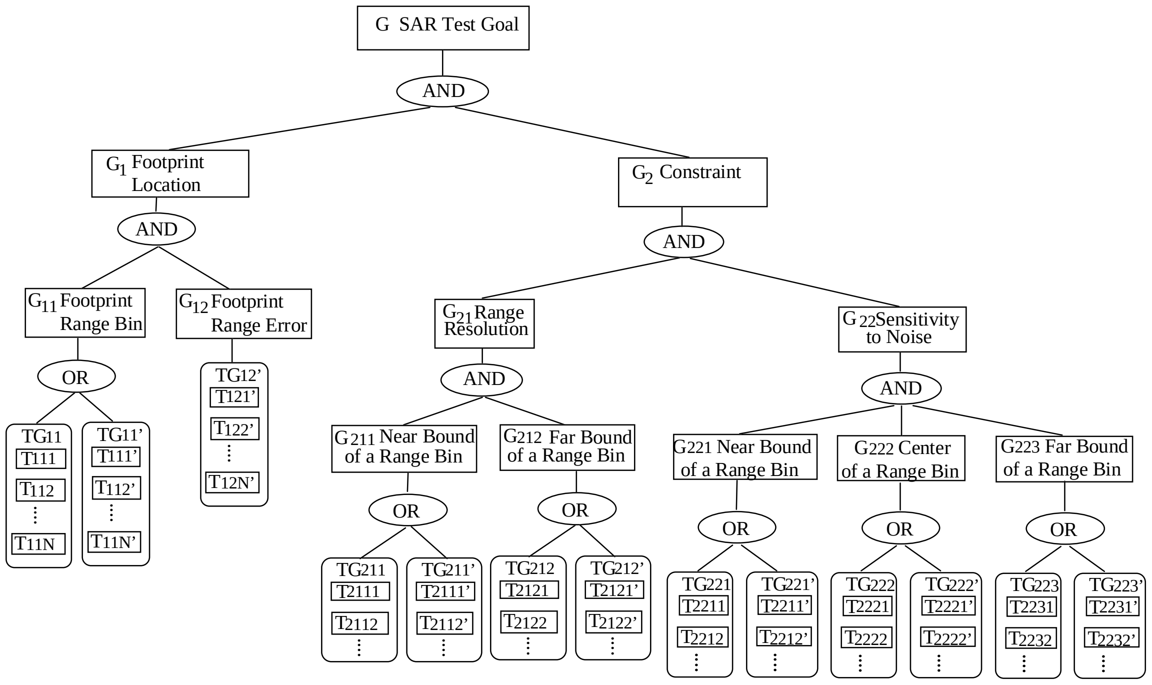


Figure 4.8 SAR Goal Tree

The constraint goal G_2 is decomposed into the range resolution subgoal G_{21} and the noise sensitivity primitive goal G_{22}. The range resolution goal G_{21} is composed of two primitive goals G_{211} and G_{212} which together search for the resolution of range compression processing in meters. The noise sensitivity goal is divided into three primitive goals G_{221}, G_{222}, and G_{223}, which search for the maximum Gaussian noise that the MUT can tolerate at the center, the leftmost, and the right most locations of a range bin, respectively, where the three locations are found in G_{21}.

Table 4.2 summarizes the features of the primitive goals in the SAR test plan including their test groups, adjustable requirements, test strategies, numbers of test cases, and comparator parameters. Except G_{12} that is supported by a file I/O test group (i.e., TG_{12}'), all other primitive goals are supported by normal test groups (e.g., TG_{11} for G_{11}) and their file I/O counterparts (e.g., TG_{11}'). G_{12} simply borrows test files generated in TG_{11}. The comparator of the test bench can be configured into two types. The first comparator compares the MUT response and the gold response with respect to the range bin numbers of the targets, while the second comparator compares with respect to the target range in meters.

Chapter 6 will detail the SAR goal tree and provide the simulation results.

Table 4.2 SAR Primitive Goals

Primitive Goal	**Test Group**	**Adjustable Requirement**	**Test Strategy**	**Number of Test Cases**	**Comparator Parameter**
G_{11}	TG_{11}	target location	even sampling with endpoints	fixed, given by test plan	range bin number
	TG_{11}'	target location	file I/O	fixed, given by test plan	range bin number
G_{12}	TG_{12}'	same as TG_{11}'			target range in meters
G_{211}	TG_{211}	target location	geometric-binary search	$O(\log(\text{swath width} \times \text{sampling rate}))$	range bin number
	TG_{211}'	target location	file I/O	$O(\log(\text{swath width} \times \text{sampling rate}))$	range bin number

G_{212}	TG_{212}	target location	geometric-binary search	O(log(swath width $\times$ sampling rate))	range bin number
	TG_{212}'	target location	file I/O	O(log(swath width $\times$ sampling rate))	range bin number
G_{221}	TG_{221}	noise level	geometric-binary search	O(log(1 / precision))	target range in meters
	TG_{221}'	noise level	file I/O	O(log(1 / precision))	target range in meters
G_{222}	TG_{222}	noise level	geometric-binary search	O(log(1 / precision))	target range in meters
	TG_{222}'	noise level	file I/O	O(log(1 / precision))	target range in meters
G_{223}	TG_{223}	noise level	geometric-binary search	O(log(1 / precision))	target range in meters
	TG_{223}'	noise level	file I/O	O(log(1 / precision))	target range in meters