Classification Trees for Random Tests and Functional Coverage

Alexander Krupp, Wolfgang Mueller
Paderborn University/C-LAB, Paderborn, Germany

1. Introduction

This article presents the classification tree method for functional verification to close the gap from the specification of a test plan to SystemVerilog [2] testbench generation. Our method supports the systematic development of test configurations and is based on the classification tree method for embedded systems (CTM/ES) [1] extending CTM/ES for random test generation as well as for functional coverage and property specification. We support the structured coding of assertions and constraints by a two-step method: (i) creation of the classification tree (ii) creation of (sample) abstract test sequences. For SystemVerilog testbench generation, we introduce a mapping to SystemVerilog random tests, assertions, and functional coverage specifications. As our method is derived from the CTM/ES, it is also compliant to the V-method and thus applies to IEC61508-conformant development of electronic safety related systems. The remainder of this paper gives an overview of the classification tree method (CTM) before presenting our extension for functional verification.

2. Classification Tree Method (CTM)

The Classification Tree Method (CTM) was developed in the early nineties at Daimler-Benz AG for the structured representation of test cases. Most recently they were extended for Embedded Systems (CTM/ES) [1]. In classification trees, potential inputs to a *system under test (SUT)* are defined as a tree with composition, classification, and class nodes. The development of classification trees and the associated combination tables is supported by the classification tree method (CTM). In CTM/ES, classifications are derived from the interface of the system under test (Fig. 1) and classes are given by values or intervals. The combination table of the CTM/ES defines abstract test sequences with time annotated test steps, the so-called synchronisation points referring to classes, i.e., to values or intervals. Meanwhile, several editors supporting CTM/ES support became available, like Razorcat's CTE, which is integrated into MTest from dSPACE, a test automation environment for Model/Hardware-In-the-Loop simulation [3].

3. CTM for Functional Verification

In a first step, we create the classification tree for the testbench starting from the interface description of the design under test (DUT). Based on a existing interface we create a classification tree, e.g., for an interface ACC we create a tree for a testbench with ACC_TB as a root node. For this, we only consider in and $inout$ ports and create one classification node for each of them. In our small examples, the classifications are taken as direct ancestors of the root node and we arrive at a tree with *desired_speed*, *tracking_distance*, *desired_distance*, *tracking*, etc. as classifications (see Fig.1).

In a next step, the user manually creates classes for each classification. The creation of classes for test values and test intervals is due to the requirement specification or functional specification, respectively. For the classification *desired_speed*, for instance, this is measured in meters per second with the classes $[-5:-1]$ for backward, 0 for stopping, and other intervals and test points like $[1:4]$ for driving forward (cf. Fig. 1). Based on that basic structure of the classification tree, we continue to define random test and functional coverages, which are defined as tree annotations and as tables referring to the existing classes.

Constraints and Weights for Random Tests. We define constraints for random generation as boxed tables. Those tables annotate the tree at the bottom and refer to the different classes as given in Fig. 1. Each boxed table defines a compound constraint, where each line in the table represents a single constraint. A constraint is defined by an optional square and circles on the line both referring to specific classes (i.e., values/intervals of classifications). A circle on a line selects a class with a specific weight. When no value is given, the default weight is 1. In Fig. 1, for example, the first line defines the weights 20 and 80 for the classes of $tracking$. Conditional constraints are defined with a square and circles, where the square specifies the condition and circles specify which inputs are considered for distribution of random values, where each point can be annotated by a weight again. E.g., a square for 1 of $tracking$ in our example defines that the line with the specific random distribution only applies for $tracking = 1$.

In Fig. 1, the table has $Constraint1$ with simple depen-

3-9810801-0-6/DATE06 © 2006 EDAA

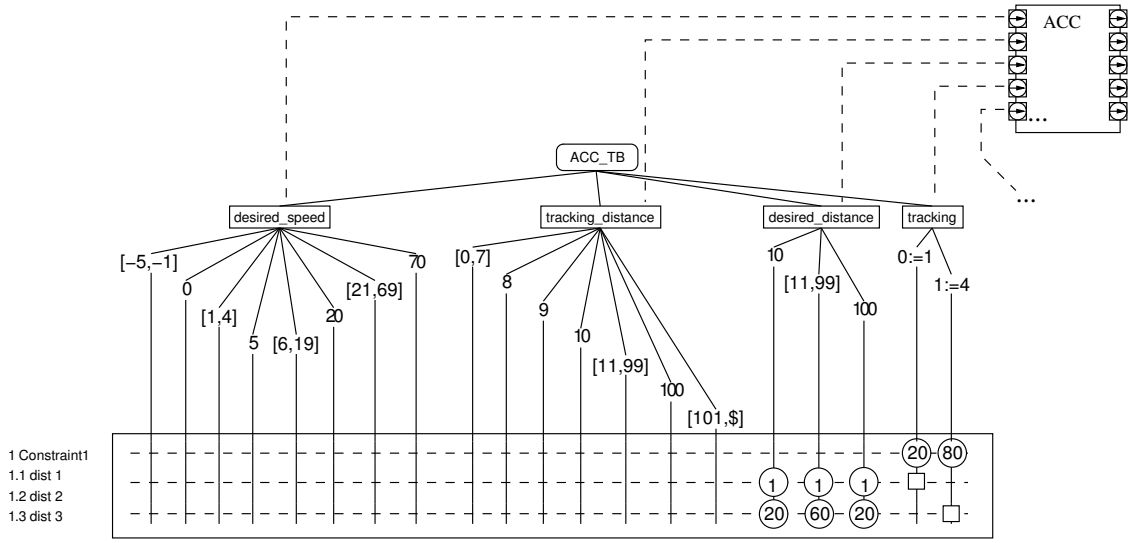

Figure 1. Example: Adaptive Cruise Controller

dencies between $tracking$ and $desired\_distance$. Random test generation shall be executed with weights depending on the different classes for $tracking$, where the corresponding SystemVerilog code sketches the mapping:

```
rand CT_value tracking, desired_distance;
...
constraint Constraint1 {
   tracking.value == {0 := 20, 1 := 80};
   tracking.value == 1 -> (
      desired_distance.value inside {
         10:=20,[11:99]/=60,100:=20
      };
   );
   ...
}
```

For the generation of randomized variables, classes like $tracking$ and $desired\_distance$ are given as instantiations of type CT_value, which is a struct containing a value and a transition. For $tracking$, for instance, weight is defined with a distribution of 20% and 80%, respectively. For $desired\_distance$, a different distribution is chosen, depending on the value of $tracking$: if $tracking = 0$, $desired\_distance$ is set to 10 or 100 with a weight of 20% each, and it is set to a value from the interval $[11, 99]$ with a weight of 60% for the interval. If $tracking = 1$, $desired\_distance$ is set with equal weights to the corner cases and the interval, with $33.\overline{3}\%$ each.

Functional Coverage and Bins. SystemVerilog offers `covergroup`, and `coverpoint` with `bins` for the definition of functional coverage. The previously described subtrees naturally map to such definitions without any major modification. I.e., classifications refer to a set of coverpoints and each class refers to a bin. As an example, take the following SystemVerilog code, which directly corresponds to the tree in Fig. 1.

```
covergroup ACC_TB @(posedge clk);
   ...
   tr0: coverpoint tracking
        {bins tr1 [1] = {0}; option.weight = 1;};
   ...
endgroup;
```

Here, the coverpoint *tr0* for input variable *tracking* contains a bin for the time 0 with weight 1 as an annotation.

Creation of Abstract Test Sequences. The definition of abstract test sequences is accomplished by a set of combination tables, one for each test sequence. For random test generation, we can associate a classification tree test sequence with productions of a SystemVerilog random sequence grammar. The different test sequences then show up as different alternatives of a production in that grammar. In addition, each test sequence is given by a production rule with the sequence of synchronization points. For each synchronization point, we assign the values/intervals as well as the transition type and randomize over the interval. Additionally, we can easily apply test sequences for the generation of transition coverage specifications without further modification. Then, a covergroup is triggered by the synchronisation point event and has a coverpoint for a class with a bin, which covers the specific abstract test sequence.

References

[1] Mirko Conrad. *Modell-basierter Test eingebetteter Software im Automobil*. Deutscher Universitäts-Verlag, Wiesbaden, 2004.

[2] IEEE. *IEEE Std.1800-2005 - Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language*, November 2005.

[3] Klaus Lamberg and Michael Beine. Test methods and tools in model-based function development. In *ASIM, Fachtagung Simulations- und Testmethoden für Software in Fahrzeugsystemen*, March 2005.