SYSTEMATIC TESTBENCH SPECIFICATION FOR CONSTRAINED RANDOMIZED TEST AND FUNCTIONAL COVERAGE

Alexander Krupp and Wolfgang Mueller
Paderborn University/C-LAB, Fuerstenallee 11, D-33102 Paderborn, Germany
E-mail: {Alexander.Krupp,Wolfgang.Mueller}@c-lab.de

Abstract— **Functional Verification is well-accepted for Electronic System Level (ESL) based designs and is supported by a variety of standardized Hardware Verification Languages like PSL, e, and SystemVerilog. In this article, we present the classification tree method for functional verification (CTM/FV) as a novel method to close the gap from the verification plan to the specification of randomized tests and functional coverage for test configurations. CTM/FV is introduced based on graphical means from which we automatically generate SystemVerilog code as a testbench for constraint-based randomized tests and functional coverage, where concepts are outlined by the automotive example of an adaptive cruise controller.**

I. Introduction

With increasing complexity, electronic systems design became a verification-centric activity. In this context basic concepts of formal verification [8] and simulation [7] were unified and advanced under the umbrella of Assertion Based Verification (ABV) [13]. Today, we can observe significant progress in tools managing and processing assertions and testbench features (i.e., randomized tests and functional coverage) in order to improve quality of designs and efficiency of ESL testbenches. Tools like *Questa*, *vManager*, *Specman* offer verification management and code generation for hardware verification languages (HVL) like PSL, e, and SystemVerilog [10]. They provide advanced support for assertion-based simulation, formal verification, functional coverage specification, and constraint-based random test generation. Though those tools already provide great help in test configuration management, we currently observe a big gap from the verification plan to the specification of assertions and testbench features since there is no systematic method for test configuration development.

In this article, we introduce the classification tree method for functional verification (CTM/FV) as a novel method to support the systematic development of test configurations. CTM/FV is based on the classification tree method for embedded systems (CTM_{EMB}) [1] with extensions for random test generation as well as for functional coverage and general property specification. We introduce CTM/FV as a methodology to fill the gap from the verification plan to the coding of constraint-based randomized tests and functional coverage as a two-step method: (i) creation of the classification tree (ii) creation of (sample) abstract test sequences. To give CTM/FV specifications a well-defined semantics and to demonstrate its applicability for testbench generation, we present CTM/FV with a mapping to SystemVerilog. This covers the automatic generation of random tests, and functional coverage expressions. As CTM/FV is derived from CTM_{EMB}, it is also compliant to the IEC61508 standard for the development of electronic safety related systems.

The remainder of this paper is structured as follows. The next section introduces related work before we give short overviews of what we need from SystemVerilog and the Classification Method for Embedded Systems (CTM_{EMB}). Thereafter, we present our methodology for random test generation and functional coverage specification and their mapping to SystemVerilog.

II. Related Work

C-based languages like SystemC and SystemVerilog are well accepted for electronic systems description and verification. In simulation, classical gate level designs considered a toggle coverage based on stuck-at fault models as a quality metrics [7]. At higher levels of abstraction, different implicit code coverage metrics are already applied for several years like statement/edge, condition, decision, and MCD (modified condition decision) coverage [2]. However, those measurements are often not sufficient because they just give information that different parts of the code are exercised and always presume an existing golden design. Therefore, complementary principles of assertion-based and functional verification with explicit specification of design-specific assertions and testbench features like functional coverage and constraints for randomized tests became really popular [13].

Several so-called hardware verification languages (HVLs) for functional specification are available for functional verification, like *PSL* [4], *e* [6], and the *SystemVerilog* subset [5]. For tool support, Mentor Graphics, for instance, provides the *Questa Verification Platform* [15], which supports coverage-driven, and constrained-random verification. Cadence offers *vManager* for the management of functional verification specification and execution, and *Specman* for testbench automation [16]. The latter supports constraint-driven test generation, functional coverage analysis, and assertion checking. The management features of vManager aim at the automatic scheduling of computing resources for verification. Both tools link to several *Electronic Design Automation (EDA)* simulators and operate on all major existing description languages such as Verilog, VHDL, and SystemC.

When applied correctly, functional coverage complements and surpasses code coverage measurements. Explicit functional coverage definitions are derived from the specification, give meaning to coverpoints, and, moreover, they enable detection of omissions in the

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

code. The main disadvantage is that there is currently no method to systematically guide the user through the activity of annotating the model with randomized tests, coverage statements, and assertions. To overcome this, we apply the Classification Tree Method (CTM), which was developed for structured test developments in software systems [3]. However, there was only little work investigating CTM beyond the initial application, such as Singh et al. who combined Z with CTM [12]. As a significant advance for automotive systems design, Conrad recently introduced an extension of CTM for embedded software labeled CTM_{EMB} [1]. Several tool suites with CTM_{EMB} support became available, e.g. MTest from dSPACE [14], a test automation environment for Model/Hardware-In-the-Loop simulation, which integrates the Razorcat Classification Tree Editor (CTE).

In [9] we already presented initial ideas for the application of classification trees for functional verification. Based on these ideas, we introduce CTM/FV as a complete method for randomized test and functional coverage specification with SystemVerilog code generation combining tree and test case specification. CTM/FV is introduced to complement existing approaches for the specification of verification plans by the means of CTM/FV-based testbench development and thus fills the gap from the verification plan to code generation. Though we present CTM/FV in combination with SystemC and for SystemVerilog code generation, CTM/FV principles apply for e language specifications and other modelling languages as well.

III. SystemVerilog

Verilog and VHDL have been the two dominant hardware description languages for simulation and synthesis over the last 15 years. To increase its market share, Verilog was initially published as IEEE Std. 1364 in 1997. Later, Accellera advanced Verilog to SystemVerilog 3.1 in 2002 and to SystemVerilog 3.1a, which became an IEEE Standard [5] in 2005. SystemVerilog IEEE Std.1800-2005 introduces many features for real object-oriented and communication-centric designs like classes, interfaces, and interprocess communication synchronization. Moreover, SystemVerilog supports the specification of random tests and properties, where the latter can be used for verification as an assumption, an assertion, or a coverage specification given by, e.g., tool directives *assert*, *assume*, *expect*, and *cover*. An assert statement enforces a property for a checker, an assume property can be considered as a hypothesis to prove the property, and the expect statement waits for a property evaluation. Coverage statements specify an explicit coverage metric by means of covergroups with coverpoints and bins for variables as given by the following example.

```
bit [9:0] v_a;
enum { red, green, blue } color;
covergroup cg @(posedge clk);
    coverpoint v_a {
        bins a = { [0:63],64};
        bins b ... ;
        ...
        bins others[] = default;
    }
    coverpoint color;
endgroup
```

The example shows a covergroup, which is triggered on a clock. The covergroup contains a first coverpoint, which associates a variable v_a (a 10-bit integer) with a number of *bins*. Each bin matches a value range and keeps a hit state for the variable value. A hit is determined once a matching value for a bin has occurred. Here, bin *a* is hit by an integer value between 0..63, and by 64. The bin *others* declared with the *default* keyword is hit by values unmatched by any other bin. The number of bins determines the size of the coverage space for a coverpoint: a coverage of 40% for a coverpoint consisting of 5 bins means, that 2 bins have been hit. A second coverpoint associates to a state variable *color*. It implicitly declares three bins, one for each possible value of the variable. The coverage value for a covergroup is the average calculated across all of its coverpoints. Overall coverage is calculated in a similar way from coverage of all covergroup instances. Coverage calculation can be influenced through optional parameters on covergroups and coverpoints. Besides covering a single domain, a coverpoint can cover several domains (cross coverage), and sequences of transitions.

A very powerful SystemVerilog feature is the specification of constraints for randomized test generation, which refer to data structures randomized as objects that contain random variables and user-defined constraints. Constraints, for instance, can be efficiently used to specify corner cases. The following example instantiates a class *CX* and randomizes its member x, such that it conforms to the constraint $x > 0$.

```
class CX
     rand bit[7:0] x; ...
endclass
...
CX cx = new;
success = cx.randomize() with {x>0;};
```

In the remainder of this paper, we focus on the generation of testbench features, i.e., cover statements and random tests, from our CTM extensions, which are introduced in the next section. Those parts and the other property specifications represent an independent part of SystemVerilog, which can be easily bound to other languages so that we can easily use it in combination with our SystemC simulation models.

IV. CTM

Classification Trees were developed at Daimler-Benz AG in the early nineties for the systematic specification of test cases [3]. In classification trees, potential inputs to a *system under test* are defined as a tree with composition, classification, and class nodes. A simple example for a brake system is given in Fig. 1). Starting from the root node, the testbench is partitioned by *composition* nodes into the 'environment', 'driver', and 'vehicle'. A composition can be hierarchically structured into further compositions, which are finally partitioned into *classifications* representing concrete input

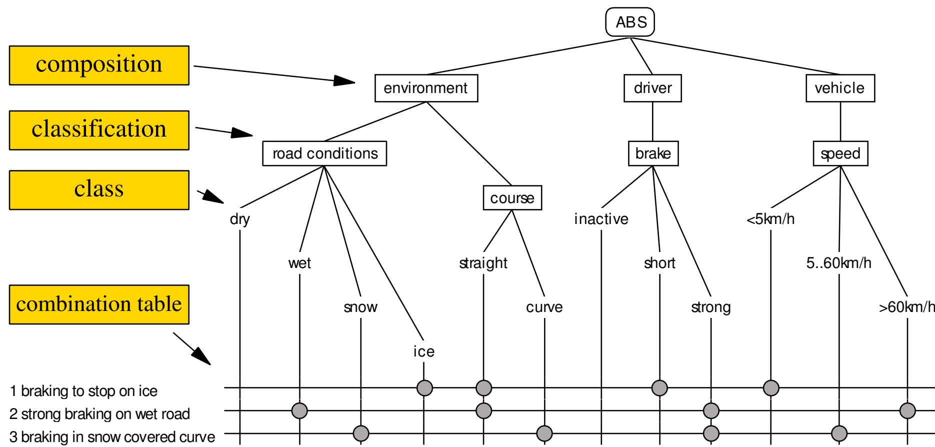

Fig. 1: Classification Tree Example: Adaptive Cruise Control

domains. Our example, for instance, has a classification for 'road conditions' under the composition 'environment'. Each classification finally has classes, which represent different subdomains of test data. For 'road conditions' classification, the corresponding classes are 'dry', 'wet', 'snow', and 'ice'. Classes of the classification tree make up the columns of a *combination table* where horizontal lines represent test cases made up from classes. The development of classification trees and associated combination tables is supported by the classification tree method (CTM) [3], which in turn is based on the category-partition method [11].

For embedded systems testing, Conrad [1] has extended the classification tree method to the Classification Tree Method for Embedded Systems ($\mathrm{CTM_{EMB}}$). In $\mathrm{CTM_{EMB}}$, classifications are derived from the interface of the system under test (Fig. 2, upper half). Classes are given by values or intervals, derived from the specification. The combination table defines abstract test sequences with time annotated test steps, called synchronisation points, which refer to classes, i.e., to value intervals, as shown in figure 3. Interpolation functions such as step, ramp, and sine are assigned to transitions between two synchronisation points, where different functions are indicated by different line styles. A concrete test sequence is then derived by value instantiation from classes, and, finally by interpolation and discretization to fit a sampling rate.

V. CTM/FV

The following subsections describe the Classification Tree Method for Formal Verification CTM/FV, and demonstrates its relation to SystemVerilog coverage and constraints for randomized test in the definition of a verification plan. The approach is illustrated by means of an Adaptive Cruise Controller as a design example from the automotive domain. An ACC is a radar-based system, which keeps either a desired speed, or the distance to an obstacle in front.

A typical verification plan relates design features to the design specification, defines verification strategies, and, e.g. for testbench generation includes an executable and machine readable part, which describes test configuration, testbench infrastructure, and test completion criteria [17]. CTM/FV as an extension of $\mathrm{CTM_{EMB}}$ assists in the structured definition of test configuration and test coverage criteria, up to the point where automatic generation of a hardware verification language (like SystemVerilog) is possible. This closes a gap in the definition of the verification plan to relieve of manual coding of assertions, random constraints, and coverage statements. Application of CTM/FV involves two development steps: the creation of a Classification Tree, and the creation of Abstract Test Sequences, constraints and coverage information.

For our outline, we presume that the interface definition of the ACC is available in any system description language like VHDL, SystemVerilog, or SystemC so that we just have to bind the design under test to the SystemVerilog testbench. The following example shows a binding of the ACC to a SystemVerilog coverage definition (ACC_COV).

```
bind ACC ACC_COV acc_cov_bind
  (.clk(clk), .desired_speed(desired_speed),...);
```

In order to arrive at a complete testbench, it has to be noted here, that the user has to decide on the sampling rate of the testbench at an earlier phase.

A. Creation of Classification Tree

Considering the example of an Adaptive Cruise Control (ACC), the following SystemC code fragment may sketch an interface definition.

```
SC_MODULE(ACC)
{
    sc_in<int>   desired_speed;
    sc_in<int>   tracking_distance;
    sc_in<int>   desired_distance;
    sc_in<bool>     tracking;
    ...
};
```

Based on that interface we semi-automatically create a classification tree for the ACC testbench (*ACC_TB*) with *ACC_TB* as root node. We just consider *sc_in* and *sc_inout* ports of the interface and create one classification node for each of them. In complex designs

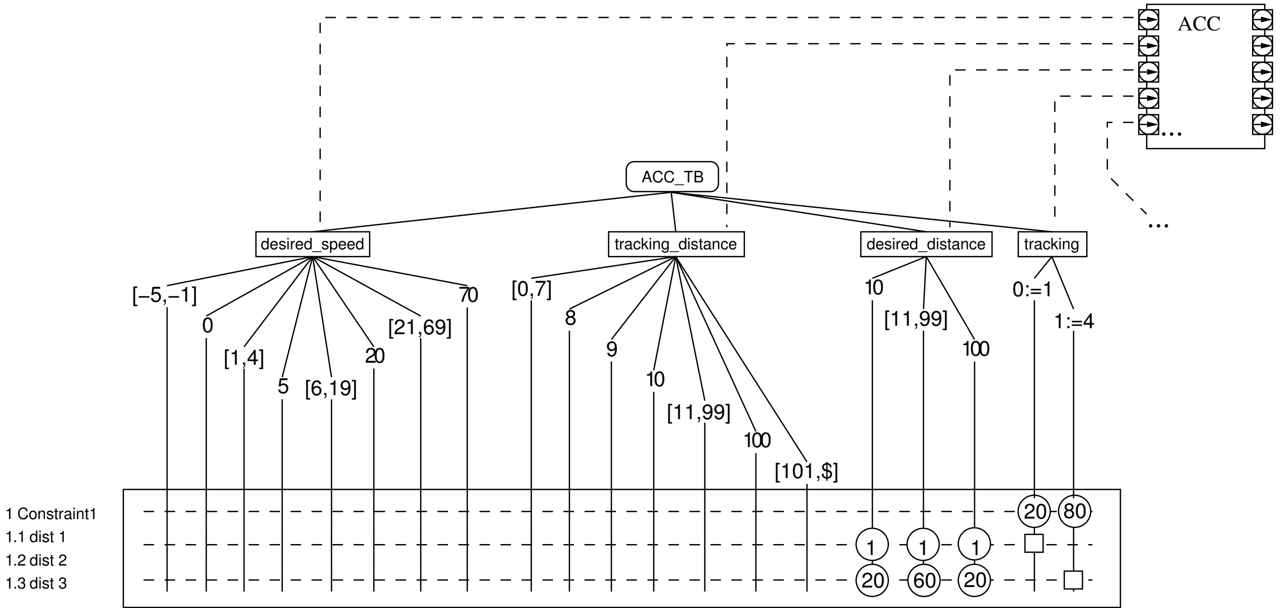

Fig. 2: Adaptive Cruise Controller with CTM/FV constraints and coverage

with several interfaces, intermediate nodes (i.e., compositions) are inserted for each interface. However, in our small example, the classifications are taken as direct descendants of the root node and we arrive at a tree with classifications *desired_speed*, *tracking_distance*, *desired_distance*, *tracking*, etc. (see Fig. 2).

In a next step, the user manually creates classes for each classification. The creation of classes for test values and test intervals mostly depends on the design specification, design features, and intended coverage, and requires the designer's expertise. 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 (see Fig. 2 for other examples). Based on that basic structure of the classification tree, we continue to define random test and functional coverage, which are defined as tree annotations and as tables referring to the existing classes.

A.1 Constraints and Weights for Random Tests

CTM/FV defines constraints for random test case generation as boxed tables in place of the combination table. They refer to the different classes as given in Fig. 2. Each boxed table defines a constraint compound, with each line in the table representing a single constraint. A single constraint is defined by placing annotated points and an optional square on the line. Points on a line define weights for the class. When no value is given, the default weight is 1. In Fig. 2, for example, the first line defines the weights 20 and 80 for $tracking = 0$, and $tracking = 1$, respectively. Conditional constraints are given by a square and points, where the square specifies the condition and points specify which inputs are to be constrained. E.g., the square for $tracking = 1$ in our example defines and implication, such that *desired_distance* is distributed with weights $\{20, 60, 20\}$ across associated classes $\{10, [11, 99], 100\}$. Therefore the boxed table defines *Constraint*1 with simple dependencies between *tracking* and *desired_speed*. Random test generation shall be executed with weights depending on the different class instantiations for *tracking*, where the corresponding SystemVerilog code gives the precise semantics:

```
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
      };
   );
   tracking.value == 0 -> (
      desired_distance.value inside {
         10:=1,[11:99]/=1,100:=1
      };
   );
}
```

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*, 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 distribution of 20% each, and it is set to a value from the interval $[11, 99]$ with a distribution of 60% for the interval. If $tracking = 1$, *desired_distance* is set with equal distribution to the corner cases and the interval, with $33.\overline{3}\%$ each.

A.2 Functional Coverage and Bins.

Hardware verification languages provide means for the definition of an individual coverage metric. SystemVerilog has covergroups, coverpoints, and bins for the definition of functional coverage. CTM/FV 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. 2.

```
covergroup ACC_TB @(posedge clk);
   ...
   dd1: coverpoint desired_distance
        {bins dd1[1] = {10}; };
   dd2: coverpoint desired_distance
        {bins dd2[1] = {[11:99]}; };
   dd3: coverpoint desired_distance
        {bins dd3[1] = {100}; };
   tr0: coverpoint tracking
        {bins tr1[1] = {0}; option.weight = 1;};
   tr1: coverpoint tracking
        {bins tr2[1] = {1}; option.weight = 4;};
endgroup;
```

Here, the coverpoint *dd2* for input variable *desired_distance* contains a bin for the interval $[11 : 99]$, for instance. In SystemVerilog, covergroups and coverpoints may have options like *weight* or *goal*. In our classification tree, this just needs an annotation for classes. In the example, the two classes of *tracking* are annotated with weights: $0 := 1$ and $1 := 4$, which directly correspond to coverpoints $tr1$ and $tr2$ in the code.

A similar approach has to be taken when considering tests with floating-point values, as common with Matlab/Simulink models. The classification tree in figure 3 shows classes, which encompass the continuous value range from -5.0 to 70.0 by means of intervals between values. For coverage computation we introduce a tolerance e in order to better cover the corner cases between intervals and values. For floating point test cases, a class 5 is represented as $[5.0 - e : 5.0 + e]$ and the interval $]0.0 : 5.0[$ becomes $]e : 5.0 - e[$. This mapping reflects the intended use of bins and coverage specifications, since floating point-based calculation hardly ever matches an exact value. Please note, that coverage of floating-point values is not supported by IEEE 1800-2005[5]. We apply a notation similar to SystemVerilog for floating-point coverage:

```
coverpoint desired_speed {
  bins ds0 = { [-5.000:-0.001[};
  bins ds1 = { [-0.001: 0.001] };
  bins ds2 = { ] 0.001: 4.999[ };
  bins ds3 = { [ 4.999: 5.001] };
  ...
}
```

The syntax allows floating-point numbers and open intervals. A coverage tolerance of $e = 0.001$ is generated here during the translation step from the classification tree. The generator will have to adapt the tolerance relative to the order of magnitude of the covered value, of course.

For cross coverage definitions and their representations in the classification tree we can use a similar representation as for the definition of random test constraints. In contrast, we apply rectangles marking classifications with annotations rather than squares and points.

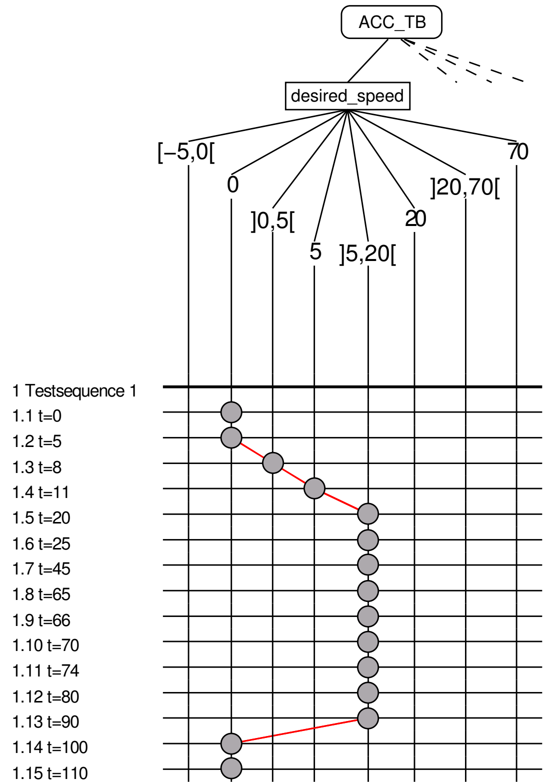

Fig. 3: Sample Test Sequence for an Adaptive Cruise Controller

As an example, the following SystemVerilog code shows cross coverage statement over *tracking_distance*, *desired_distance*, and *tracking* for the previously introduced *ACC_TB* covergroup. The line in the classification tree table has three annotated rectangles for the three classifications and corresponds to the following code.

```
tdxds: cross tracking_distance,
             desired_distance,
             tracking {
   bins cr_tr1[]  = binsof(tr1);
   bins cr_tr0[1] = binsof(tr0);
}
```

B. Creation of Abstract Test Sequences

In $\mathrm{CTM_{EMB}}$, the definition of abstract test sequences is accomplished by a set of combination tables, one for each test sequence. Fig. 3 gives the example of one fraction of a test sequence for the classification *desired_speed*. Selected test points on each line (i.e., each synchronisation point) refer to the different classes. For realistic tests, we assign an interpolation function to each transition between two synchronisation points of one classification. An interpolation function can be of type (*step*, *linear*, *sine*), which is represented by different line styles. When no line is given, the *step* interpolation is assigned as a default.

The sample test sequence in Fig. 3 starts at $t = 0$ with *desired_speed* $= 0$. Then it changes to $]0, 5[$

at $t = 8$, to 5 at $t = 11$, and to $]5, 20[$ at $t = 20$, where it remains until $t = 50$ before returning to 0 at $t = 100$. Note that the individual synchronisation points are coarse-grained and refer to fine-grained time points of the sampling rate. Recall from Section 3 that the test sequence is considered as abstract since not a concrete value for each interval has been selected yet. The next paragraph outlines how the definition of those test sequences can support automatic generation of randomized test and transition coverage specification.

B.1 Test Sequences for Random Tests.

For random test generation, we associate a CTM/FV test sequence with SystemVerilog for a random sequence production specification. The different test sequences then show up as different alternatives of a production in the 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. We finally have to apply an interpolation since the synchronisation points of the abstract test sequence are considered as sparse w.r.t. the sampling rate.

Given the sample test sequence in Fig. 3, we can automatically generate the following SystemVerilog random sequence.

```
randsequence( ts )
  ts   : ts1 | ts2 | ... | ... ;
  ts1  : ts1_1 ts1_2 ts1_3 ts1_4 ts1_5
         ts1_6 ts1_7 ts1_8 ts1_9 ts1_10
         ts1_11 ts1_12 ts1_13 ts1_14 ts1_15;
 ts1_1 :
    { desired_speed.value = 0;
      tracking_distance.value = 100;
            ...
      desired_speed.transition = STEP;
            ...
      interpolate(desired_speed,...);
          };
 ts1_2 :
    { desired_speed.randomize() with {
        desired_speed.value inside {[1:4]} &&
        desired_speed.transition = LINEAR
      };
            ...
      interpolate(desired_speed,...);
     };
  ts1_3 : ...
  ...
endsequence
```

In this example, different test sequences are given as $ts1$, $ts2$ etc. where $ts1$ refers to the sample test sequence of Fig. 3.

B.2 Transition Coverage.

Test sequences also seamlessly apply for the generation of transition coverage specifications without further modification. The following example directly refers to test sequence $ts1$ in Fig. 3.

```
covergroup ts1 @(ts1_trigger);
   coverpoint desired_speed {
      bins ds_ts1 = (0 => 0 => [1:4] => 5
                   => [6:19] [* 9] => 0 => 0);
   };
endgroup;
```

Here, the corresponding SystemVerilog covergroup is triggered by the synchronisation point event *ts1_trigger* and has a coverpoint for the variable *desired_speed* and a single bin, which covers the abstract test sequence $ts1$. It is easy to see from this example, that the coverage of all sample test sequences for a classification tree can be generated and gives intuitive means for the efficient specifications of transition coverages.

VI. Conclusions and Outlook

In this article, we introduced the classification tree method for functional verification (CTM/FV) as a novel method to support the systematic development test configurations. Though we have introduced CTM/FV just for randomized test and functional coverage specification for SystemC models and automatic SystemVerilog code generation, our method supports different HVLs such as the *e* language and the specification and documentation of general properties, i.e., assertions and assumption as well. Additionally, for a wider application, we already have solutions to extend CTM/FV and SystemVerilog for the specification of intervals based on float values.

References

[1] M. Conrad. *Modell-basierter Test eingebetteter Software im Automobil.* Dt. Universitäts-Verlag, Wiesbaden, 2004.
[2] C. Ghezzi, M. Jazayeri and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall PTR, 1991.
[3] M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Softw. Test., Verif., Reliab.*, 3(2):63–82, 1993.
[4] IEEE. *IEEE Std 1850-2005 - Standard for Property Specification Language (PSL)*, September 2005.
[5] IEEE. *IEEE Std.1800-2005 - Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language*, November 2005.
[6] IEEE. *IEEE Std.1647-2006 - Standard for the Functional Verification Language 'e'*, March 2006.
[7] N.K. Jha and S.Gupta. *Testing of Digital Systems.* Cambridge University Press, 2003.
[8] Th. Kropf. *Introduction to formal hardware verification.* Springer, 1999.
[9] A. Krupp and W. Mueller. Classification Trees for Random Test and Functional Coverage. In *Design, Automation and Test in Europe (DATE 2006)*, Munich, Germany, March 2006
[10] W.H.K. Lam. *Hardware Design Verification.* Prentice Hall PTR, 2005.
[11] Th.J. Ostrand and M.J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Commun. ACM*, 31(6):676–686, 1988.
[12] H. Singh, M. Conrad, and S. Sadeghipour. Test Case Design Based on Z and the Classification-Tree Method. In *ICFEM '97*, Washington, DC, USA, 1997. IEEE Computer Society.
[13] Synopsys. *Assertion-Based Verification - White Paper.* March 2002. www.synopsys.com/simulation.
[14] http://www.dspace.de/ww/en/gmb/home/products/sw/expsoft/mtest.cfm, Dec 2006
[15] http://www.mentor.com/products/fv/abv/questa_afv, Dec 2006
[16] http://www.cadence.com/products/functional_ver/vmanager/, Dec 2006
[17] Cadence Design Systems. *Reducing Block, Chip, and System Design Risk with a "Plan-To-Closure" Verification Approach*, White Paper, San Jose, CA, USA, 2006.