# Generation of Evenly Distributed Input Stimuli By Domain Clustering

Jomu George Mani Paret and Otmane Ait Mohamed

Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
{jo_pare,ait}@ece.concordia.ca

**Technical Report**

July, 2014

**Abstract**

Constrained Random Verification (CRV) is becoming the mainstream methodology for the functional verification of complex System on Chip (SoC) designs. In order to achieve verification closure, CRV tools have to produce a large number of solutions, evenly distributed, in the search space. To attain this requirement, we propose a technique which analyzes the solution space by using consistency algorithm and splits the domain of variables into clusters. The proposed technique helps us to generate input stimuli which are evenly distributed in search space. Unlike other techniques, the proposed technique does not remove solutions with low probability from the search space. The experimental results show that the proposed technique helps to improve the evenness of input stimuli distribution, of a state-of-art CRV tool.

## 1 Introduction

As semiconductor technology improves, System on Chip (SoC) designs are becoming popular. SoC platforms usually consist of various design components dedicated to specified application domains. In order to ensure the functional correctness of a SoC, finding and fixing the design errors at early design phases is important.

Functional verification is the process used to ensure, whether the design satisfies the requirements specified in the specification. It is widely recognized as the bottleneck of the hardware design cycle because of the growing demand for better performance and shorter time to market. In current industrial practice, simulation based verification techniques play a major role in the functional verification of hardware designs. The functional verification starts with a verification plan that enumerates the verification scenarios. Verification engineers then convert the verification scenarios into constraints. The constraint solver associated with the verification tool generate solutions for the constraints. These solutions are then used as input stimuli for the verification of the design. This method is known as Constraint Random Verification (CRV).

Experience shows that, as design complexity increases, many bugs remain undetected even though considerable resources and time have been devoted to design verification. Because of the elusive nature of hardware bugs, we need a large number of input stimuli to cover the scenarios specified in the verification plan. Generating a large number of input stimuli which are evenly distributed is a big problem [10]. In this paper, we propose a technique based on clustering of variable domain to generate evenly distributed solutions that helps to attain higher coverage.

## 2 Related works

In order to ensure verification closure, the distribution of the generated stimuli should be even. That is, the generated input stimuli should be uniformly distributed in the search space. Also the input stimuli generation must be fast. In this section, we will address some of the existing constraint solving techniques which focus on the above two objectives and point out their disadvantages.

The acceptance and rejection (A&R) technique [4] applies random samples to produce feasible input stimuli. It ensures uniform or user-specified distribution. But the input stimuli generation speed would be slow when constraints cannot be easily solved. Formal solution generators [12] like SAT solvers can solve general constraints very fast. But they sacrifice the evenness of distribution. Another approach to increase the success ratio for the A&R technique, called RACE [5], is to apply interval propagation to reduce ranges of variables before sampling. The interval propagation procedure requires a large number of iterations for complete range-reduction on complicated constraints. Such runtime overheads cannot be neglected since it re-computes ranges while generating each solution.

In order to increase the evenness of distribution and the speed of solution generation, the weighted Binary Decision Diagram (BDD) technique [11] converts the constraints into a single BDD structure. The probability information is annotated on the BDD edges. By biased top-down traversal on the diagram, this approach guarantees the evenness of bit-level signal distributions and fast production of random input stimuli. However, it suffers from memory explosions for complex constraints during BDD constructions.

Monte Carlo Markov Chain (MCMC) based methods [9], reach the desired distribution after a large number of state transitions. It is hard and inefficient to determine the prob-

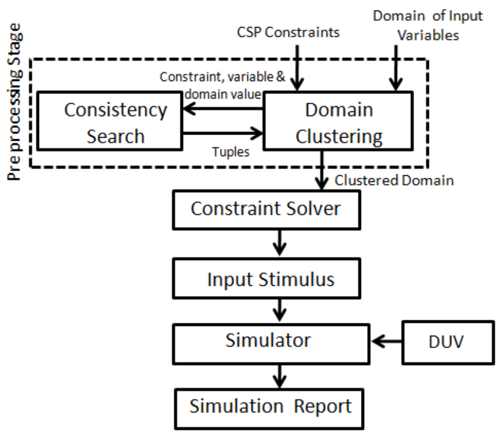

Fig. 1: Proposed framework

abilities required to move from current state to the next state, for non-continuous solution space. As a result, it converges to the target distribution slowly.

Range-Splitting heuristic and Solution-Density Estimation technique (RSSDE) [10] can be used to partition the search space in order to have even distribution of input stimuli. The range-splitting heuristic prunes subspaces which have very low probability to contain a solution. By removing subspace with low solution density, the solution densities in other subspaces are substantially enhanced.

All the above mentioned techniques except RSSDE, focus either to improve the solution generation speed or to improve the evenness of solution distribution, but not both. In RSSDE, both speed and distribution of solution generation are given focus. But in RSSDE, the eliminated sub space may contain solutions which can trigger corner cases in verification. In this paper we propose a new technique in which we focus on the above two objectives and will only remove subspaces with no solutions.

## 3 Proposed Preprocessing Framework

Most of the hardware manufacturers use the CRV methodology to produce input stimuli for verification. One of the important component of CRV tools is the constraint solver. Since CSPs arising from CRV are different from typical CSPs [3] general purpose constraint solvers are not suitable for CRV. Hence EDA tools use internally developed constraint solvers. Our main objective is to enhance the solution generation capability of constraint solvers associated with EDA tools.

In most of the stimuli generation scenarios, large number of solution has to be generated. In a CSP, the solutions are clustered together in the search space [8]. Hence partitioning the search space into clusters and generating solutions from the partitions can improve the evenness of the solutions generated by the solver. In this paper we propose a search space partitioning technique based on consistency search. Our proposed framework is presented in Fig.1.

The CSP problem and the domain of the input variables are given to the domain clustering block. This block then generate partition tuples (A tuple is an ordered list which contains values for all the variables in the constraint and a partition tuple is a tuple which contains values for all the variables in the CSP) based on the tuples returned by consistency search block. The consistency search block uses the consistency algorithm [7] to generate tuples. Then the partition tuples are used to cluster the variable domain. The generation of partition tuples and domain clusters are explained in detail in Section5. The clustered variable domain and the CSP constraints are given to the constraint solver and the input stimuli are generated. These input stimuli are used as inputs for the verification of the DUV. The simulation report is then generated by the simulator.

## 4 Background Information:Consistency Search

Consistency techniques [7] are constraint solving algorithms that reduce the search space by removing, variable values that cannot be part of any solution. For each constraint($C_i$) in the CSP, for each variable ($v_j$) in the constraint $C_i$, for each domain value of the variable $v_j$, the algorithm will try to find a tuple which satisfies the constraint. If there is a tuple which satisfies the constraint, then that tuple will be returned by consistency search. If there is a tuple which satisfies the domain value assignment and the constraint, then that domain value can be part of solution. Hence for each domain value, the algorithm needs to find only one tuple which satisfies the constraint.

To illustrate the idea discussed above, let us consider the following CSP network N with 3 constraints *C1 (a+b+c=5)*, *C2 (b+d+e=6)* and *C3 (e+f+g+h=6)* over the variables *a, b, c, d, e, f, g* and *h*. Each of the variable may hold a value between 1 and 3 inclusive, except for variable *d*. It is between 1 and 4 inclusive. The algorithm will first select the constraint $C1$, then the variable $a$ in the constraint $C1$ and assign the first value 1 to the variable $a$. Once the algorithm finds a tuple which satisfies the constraint $C1$ where variable $a = 1$, then that tuple is returned by the algorithm. For example, the tuple (1,1,3) satisfies the constraint $C1$ with the variable assignment $a = 1$. This process is repeated for all the domain values, variables and constraints in the CSP.

## 5 Domain Clustering

### 5.1 Preliminaries & Notations

*1.* A tuple $\tau$ on an ordered set of variables is an ordered list which contains values for all the variables. $Var(\tau)$ represents the ordered set of variables in the tuple $\tau$.

*2.* A constraint $C_i$ on an ordered set of variables gives the list of allowed tuples for the set of variables. $Var(C_i)$ represents the set of variables in the constraint $C_i$. $\tau_{Ci}$ represents a tuple which satisfies the constraint $Ci$. $\tau_{Ci}[m]$ represent $m$ tuples which satisfies the constraint $Ci$.

*3.* A constraint network is defined as a triple CN = $\langle Var, \mathcal{C}, Dom \rangle$ where:

$Var$ is a set of variables $\{x_1, \ldots, x_j\}$;

$\mathcal{C}$ is a set of constraints between variables $\{C_1, \ldots, C_k\}$;

$Dom$ is a finite set of domain values for the variables $\{Dom(x_1), \ldots, Dom(x_j)\}$.

*4.* $\Gamma[m]$ is the list of $m$ tuples which satisfies the highest arity (number of variable in a constraint) constraint, generated by consistency search.

*5.* $C_H$ represents the highest arity constraint from the CSP.

*6.* $\tau_{CH}$ represents tuple which satisfies the constraint $C_H$.

*7.* $\tau_{CHN}[n]$ represent first n tuples which satisfies the constraint $C_H$ from the list of tuples $\Gamma[m]$ $(m > n)$.

One way to cluster search space is to generate all possible solutions and find $n$ solutions which are far apart. These $n$ solutions are the center of the clusters and are used for partitioning. Even though this method gives best results, it is computationally expensive. In this paper we propose a methodology to cluster the search space using the tuples generated by consistency search. We will also prove that the proposed partition technique is equivalent to partitioning of solutions.

The clustering of variable domain into $n$ groups (where $n$ is a number defined by the user based on the verification scenario and time for simulation) can be divided into the following three steps:

### 5.2 Step 1: Selection of n tuples

**Algorithm 1 : Selection of n tuples**

```
1: Selection of n tuples (in:n=4, in:$\Gamma[m]$): $\tau_{CHN}[n]$
2: find $C_H$, $\tau_{CH}$ and $\tau_{CHN}[n]$
3: for i=0 to n-1 do
4:    for j=0 to n-1 do
5:       if i $\neq$ j then
6:          if j > i then
7:             HAM[i][j] = hamming distance between $\tau_{CHN}[i]$ and $\tau_{CHN}[j]$
8:          else
9:             HAM[i][j] = HAM[j][i]
10:         end if
11:      end if
12:   end for
13:   HAM[i][n] = $\sum_{j=0}^{n-1} HAM[i][j]$
14: end for
15: $HAM_T = \sum_{i=0}^{n} HAM[i][n]$
16: while tuple in $\tau_{CH}$ which is not yet considered $\neq$ nil do
17:    $\tau_{new}$ = tuple in $\tau_{CH}$ which is not yet considered
18:    $\tau_{low}$ = tuple with the lowest HAM[i][n] value
19:    $HAM_{new}$ = sum of hamming distances between $\tau_{new}$ and tuples in $\tau_{CHN}$ except $\tau_{low}$
20:    if $HAM_{new} > HAM_T$ then
21:       replace $\tau_{low}$ with $\tau_{new}$
22:    end if
23: end while
```

Table 1: Tuple after Consistency Search

| Constraint | e | f | g | h |
|---|---|---|---|---|
| C3 | 1 | 1 | 1 | 3 |
| | 2 | 1 | 1 | 2 |
| | 3 | 1 | 1 | 1 |
| | 1 | 2 | 1 | 2 |
| | 1 | 3 | 1 | 1 |
| | 1 | 1 | 2 | 2 |
| | 1 | 1 | 3 | 1 |

Initially, constraint with the highest arity is selected. For each variable ($v$) in the constraint $C_H$, for each domain value $b$ of the variable $v$, the algorithm will try to find a tuple which satisfies the constraint where the variable $v$ is assigned the value $b$. Then $n$ tuples

Table 2: 4 tuples selected from Sclast for C3

| Group | e | f | g | h |
|---|---|---|---|---|
| 1 | 2 | 1 | 1 | 2 |
| 2 | 3 | 1 | 1 | 1 |
| 3 | 1 | 2 | 1 | 2 |
| 4 | 1 | 3 | 1 | 1 |

which satisfies the highest arity constraint has to be selected from the generated tuples. The selected $n$ tuples should be far way (different) from each other. Selection of $n$ values which are far away from each other is a hard problem to solve [2]. There are several heuristics developed for the above. We used *hamming distance* heuristics, to find tuples which are far away from each other. The pseudo code for the selection of n tuples in shown in Algorithm 1.

For the example discussed in Section 4, constraint $C3$ is the highest arity constraint and the tuple generated for the constraint $C3$ is shown in the TABLE 1. If $n$ is set to 4 we need to select 4 tuples which satisfies $C3$ from the list and are faraway from each other. The selected tuples are shown in the TABLE 2.

### 5.3 Step 2: Generation of n partition tuples

**Algorithm 2 : Generation of n partition tuples**

```
1: Generation of n partition tuples (**in**$\tau_{CHN}[n]$, **in:**list of constraints - $C_H$, **in:**$\Gamma[m]$): $\tau_{CHN}[n]$
2: **while** constraints to be considered $\neq$ nil **do**
3:   $C_{l2}$ = highest arity constraint which is not yet considered and has the highest number of variables in common with $\tau_{CHN}[n]$
4:   **for** i=1 to n **do**
5:     Update $Var(\tau_{CHN}[i])$ such that $Var(\tau_{CHN}[i]) = Var(\tau_{CHN}[i]) \bigcup Var(C_{l2})$
6:     $comvar = Var(\tau_{CHN}[i]) \bigcap Var(C_{l2})$
7:     $comval$ = value of variable(s) $comvar$ in tuple $\tau_{CHN}[i]$
8:     $\tau_{Cl2}$ = tuple returned by consistency search that satisfies the constraint $C_{l2}$ and domain value of variable(s) $comvar$ is equal to $comval$
9:     **if** $\tau_{Cl2}$ = nil **then**
10:      $comval$ = next lexicographic higher value
11:      Go to step 8
12:    **else**
13:      Update the domain value of variables $Var(tau_{Cl2})$ in $\tau_{CHN}[i]$ with the domain values in $\tau_{Cl2}$
14:    **end if**
15:  **end for**
16: **end while**
```

Partition tuples are tuples which contain all the variables in the CSP. In order to make partition tuples, the highest arity constraint, which is not yet considered and has the highest number of variables in common with $n$ tuples ($\tau_{CHN}[n]$) generated earlier, is selected.

Then the $n$ tuples are modified as follows. For each tuple, the domain value of variables which are present in both the selected constraint $C_{l2}$ and $\tau_{CHN}[n]$ are determined. This domain value(s), variable(s) and the constraint is given to the consistency search block. If the consistency search does not return a tuple, then the next higher lexicographic value for domain value is chosen and used for consistency search. If the consistency search returns a tuple, then that tuple is used to update the domain value of variables in the constraint $C_{l2}$. For example in the above CSP, $C2$ is the next highest arity constraint and variable common to $C2$ and $\tau_{CHN}[n]$ is $e$. In the first tuple (2,1,1,2) variable e is equal to 2. So we do consistency search for the constraint $C2$ with $e = 2$. The tuple (3,1,2) which satisfies the constraint $C2$ and assignment $e = 2$, is returned by the consistency search. This tuple is then used to update the values of variables $b, d$ and $e$.

If variables present in both the tuple has different values then the highest domain value among them is assigned to the variable. Inorder to understand why the largest value is chosen, consider a variable $v_m$, which is assigned values $d_i$ and $d_j$ in the tuple returned by consistency search for constraint $c_i$ and $c_j$ resply. Also assume $d_i < d_j$. In the partition tuple variable $v_m$ is assigned the value $d_j$. This is because, during consistency search, tuples are generated in lexicographic order starting from the lowest value. So if for constraint $c_j$ the variable $v_m$ is assigned the value $d_j$ that means the value $d_i$ was found to be inconsistent. Hence $v_1 = d_i$ cannot satisfy the constraint $c_j$. If a variable value is inconsistent with a constraint, then it cannot be part of the solution for the CSP. The objective of the algorithm is to find clusters of solutions in the search space and partitions the search space based on the clusters. Hence for the partition tuple variable $v_m$ is assigned the value $d_j$.

This process is repeated until all the constraints in the CSP are considered. The pseudo code for the generation of $n$ partition tuples in shown in Algorithm 2. After this process, the $n$ partition tuples generated for the above CSP are as shown in the TABLE 3.

## 5.4 Step 3: Partitioning of variable domain

In this step, initially the partition tuples generated (in step 2) are arranged in lexicographic order. Then for each tuple, the domain values will be compared with their neighboring tuples, starting from the left most variable in the tuples. The leftmost variable which has a different value when compared with neighboring tuples is the partition point. If more than one tuple has the same variable value at partition point, then for those tuples we continue comparing towards the right until the variable has different values in neighboring tuples. This will be the partition point for those tuples.

The intuitive idea behind the algorithm is that partitioning of the tuples is equivalent to partitioning of the solutions of the CSP. If the arity of the largest arity constraint is nearly equal to the number of variables in the CSP, then the tuples generated by consistency search are approximately equal to the solutions of the CSP. So partitioning of the tuples is equivalent to partitioning of the solutions of the CSP. Another possibility is that the arity of the highest arity constraint is smaller than the number of variables in the CSP. Then the algorithm updates the other variable values. While updating, if a variable is having different values for different tuples, then the resultant partition tuples are different from each other. This results in good partition of the domain values. While updating, a variable can have same value for different tuples. The algorithm is using partial solutions to update variable values. Hence in actual solution those variable values may remain the same. Then those variables don't have much impact on the evenness of the solution. We can consider those variables as constant. The resultant tuples, ignoring the variables with constant values, will be different from each other and leads to good partitioning.

In TABLE 3 the first leftmost variable which is different in the partition tuple is $b$. So this is the first point of domain partition. There are two partition tuple which has the same

Table 3: 4 partition tuples

| Group | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 1 | 1 | 2 | 1 | 1 | 2 |
| 2 | 1 | 2 | 2 | 1 | 3 | 1 | 1 | 1 |
| 3 | 1 | 1 | 3 | 4 | 1 | 2 | 1 | 2 |
| 4 | 1 | 1 | 3 | 4 | 1 | 3 | 1 | 1 |

Table 4: Variable domain for n clusters

| Group | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| 1 | 1-3 | 1 | 1-3 | 1-4 | 1-3 | 1-2 | 1-3 | 1-3 |
| 2 | 1-3 | 1 | 1-3 | 1-4 | 1-3 | 3 | 1-3 | 1-3 |
| 3 | 1-3 | 2 | 1-3 | 1-4 | 1-3 | 1-3 | 1-3 | 1-3 |
| 4 | 1-3 | 3 | 1-3 | 1-4 | 1-3 | 1-3 | 1-3 | 1-3 |

value for variable $b$. Hence for those two partition tuples we continue comparing the domain values. For the above two tuples variable $f$ has different values. Hence the domain of variable $f$ is divided into two groups. Fig. 2 shows the partition points.

For all other variables which are not part of the partition point, the domain values will be the values specified in the CSP. TABLE 4 gives the domain values of all the variables of the 4 groups used for solution generation. This partitioned domain values along with the CSP constraints are then given to the constraint solver.

**Lemma 1** *For a set $S$ euclidian points, if $T$ is the solution returned by the proposed algorithm and $T_{op}$ be the optimal solution, then $Cost(T) \leq 2 * Cost(T_{op})$ (lower the value of cost implies better distribution).*

*Proof.* Let $a$ is the maximum distance between a point $x$ ($x \epsilon S$) and $T$. Then cost of $T \approx a$. Let $x_0$ be the point in **S** which replaces a point in T in the optimized solution. Then $T \bigcup x_0$ consists of $n+1$ points which are all distance $\leq a$ apart. Two of the points must be having the same closest representative in $T_{op}$ since the cardinality of $T_{op}$ is $n$. In order to have both the point in the same cluster, the representative point must be at a distance $\leq a/2$. Similarly, considering all other points in $T_{op}$, the cost of $T_{op}$ is $\leq a/2$.

**Lemma 2** *For a set of $S$ euclidian points with $m$ attributes, the time complexity of the algorithm to return $n$ points which are far apart form each other is $l$ times faster than k-means clustering.*

*Proof.* In the proposed algorithm, initially $n$ points are taken from the set $S$. Then hamming distance between the $S$ points are calculated. The proposed algorithm generates the $n$

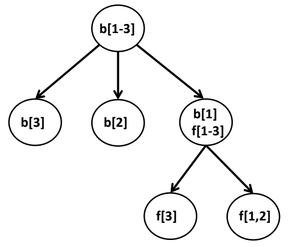

Fig. 2: Partitioning points

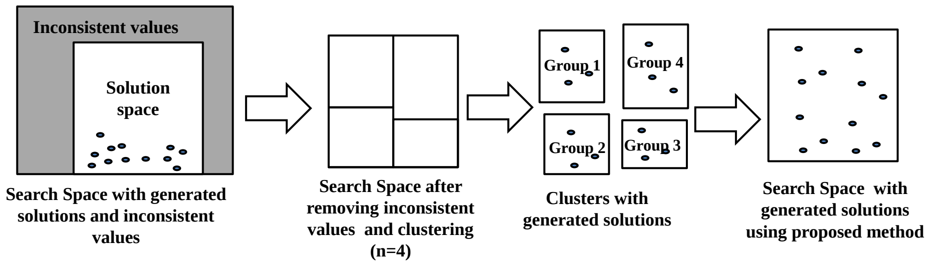
Inconsistent values
Solution space
Search Space with generated solutions and inconsistent values
Search Space after removing inconsistent values and clustering (n=4)
Group 1
Group 4
Group 2
Group 3
Clusters with generated solutions
Search Space with generated solutions using proposed method

Fig. 3: Search space with generated solutions

Table 5: Evenness Evaluation on Random Cases

| #vars | #cons | $\sigma_{DNP}$ | | | $\delta_{DNP}$ | | | $\delta_{KM}$(k=100) | | | $\delta_{KM}$(k=1000) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | M1 | M2[10] | M3 | M1 | M2[10] | M3 | M1 | M2[10] | M3 | M1 | M2[10] | M3 |
| 31 | 9 | 95.8 | 97.7 | 94.8 | 0.07 | 0.07 | 0.06 | 1396 | 1382 | 1407 | 1162 | 1187 | 1200 |
| 34 | 24 | 100.0 | 99.3 | 97.5 | 0.08 | 0.07 | 0.06 | 1372 | 1400 | 1422 | 1167 | 1191 | 1209 |
| 36 | 16 | 103.7 | 101.1 | 100.6 | 0.07 | 0.07 | 0.08 | 1446 | 1456 | 1460 | 1237 | 1249 | 1255 |
| 38 | 13 | 105.0 | 104.5 | 100.7 | 0.07 | 0.07 | 0.06 | 1565 | 1484 | 1499 | 1360 | 1378 | 1419 |
| 40 | 20 | 97.2 | 104.8 | 96.5 | 0.07 | 0.07 | 0.06 | 1446 | 1487 | 1519 | 1208 | 1277 | 1332 |

points in a single iteration. Hence the time complexity is $\mathbf{O}$(Smn). In k-means algorithm, the algorithm first divides the $S$ points into $n$ clusters. Time complexity for clustering is same as the proposed algorithm ($\mathbf{O}$(Smn)). But this process is repeated $l$ times to find the optimal cluster. Hence the proposed algorithm is $l$ times faster than k-means clustering. If the size of $S$ is large, the value of $l$ is also large.

## 6 Distribution Evaluation

Due to the unknown characteristics of solution space, it is difficult to prove evenness of the generated solutions. But, statistical analysis can give persuasive profiles about the evenness of the generated solutions. Therefore, we used three different statistical analysis to evaluate the distribution of solutions generated.

### 6.1 Evaluation Metric: Differentsoln

As mentioned earlier, our intention is to generate a large number of different solutions distributed evenly in search space. But using existing CRV tools, constraint random generation does not guarantee even distribution of solutions. In existing CRV tools, the randomization process does not give high solution generation productivity. For example, consider a simple constraint $0 \leq X \leq 10$, which is randomized 5 times and the solution generated are $3, 0, 0, 3, 0$. The solutions meet the constraint, but out of 11 possible solutions only 2 is generated. The generated solutions are not evenly distributed in search space. So we define a metric called *differentsoln* to determine the quality of the solutions generated. *Differentsoln* is defined as the number of different solutions generated by the solver. High value for *differentsoln* implies that the evenness of solution generation is higher.

### 6.2 Distance of Nearest Neighbor

The k-nearest neighbor algorithm is amongst the simplest of all machine learning algorithms. If $p_j$ is a point near to the point $p_i$, the shortest Euclidean distance between them is denoted as $d_{min}(pi)$. If the standard deviation of $d_{min}(pi)$ is smaller for a given data set, then those data set are evenly distributed. Standard deviation $\sigma_{DNP}$ is defined by

$\sigma_{DNP} = \sqrt{\frac{\sum_{i=1}^{Np}(d_{min}(pi)-\overline{d}_{min})^2}{Np}}$ where

$\overline{d}_{min}$= the average of all shortest distances

$Np$=number of points (solutions)

If the ratio between $\sigma_{DNP}$ and $d_{min}$ is smaller for a given data set, it implies that the distribution is more even. The above ratio is defined as a parameter called $\delta_{DNP}$ where $\delta_{DNP} = \frac{\sigma_{DNP}}{\overline{d}_{min}}$

### 6.3 K-Means Clustering

K-means is one of the simplest unsupervised learning algorithms. Given a set of n-dimensional data points, k-means clustering analysis, partition them into k clusters with the nearest mean. k-means defines a cost function to measure whether the data set is well clustered or not. Higher the value of cost function, more even will be the distribution. The cost function $\delta_{KM}$ is defined as

$\delta_{KM} = \sqrt{\frac{\sum_{j=1}^{k}\sum_{x \epsilon c_j}||x-z_j||^2}{Np}}$

where $c_j$ denotes the $j^{th}$ cluster and $z_j$ represents the centroid of the $j^{th}$ cluster.

K-Means and Distance of Nearest Neighbor analysis consider the correlation and distribution of data points while the discrete Fourier transform and Shannon's entropy only care the frequency of data points. Therefore, these measures give more persuasive distribution analysis [6].

## 7 Experimental Results

We used Weka[1], which is a data mining tool, for K-Means Clustering and Distance of Nearest Neighbor analysis. We used our framework with a state-of-the-art commercial tool, Synopsys VCS 2009.06. VCS 2009.06 is run on the SUN SPARC Enterprise M3000 server. It has a SPARC64 VII quad-core with 2.75 GHz and a memory of 8GB. The CSPs used has both arithmetic and logical constraints. The outputs of the CSPs were analyzed by the metrics defined in section 6.

To ensure the evenness of generated solutions, we used K-Means Clustering and Distance of Nearest Neighbor analysis. In TABLE 5, we list five cases shown in [10]. Columns 1 and 2 indicate the number of variables and constraints respectively. The domain of each variable contains 1024 values (0 to 1023). M2 represents the result obtained using the technique RSSDE [10]. M1 represents the result obtained using the CRV tool VCS for input stimuli generation and M3 represents the result obtained using domain clustering as a preprocessing step with VCS. The columns 9-11 are the results obtained when the number of centroids ($k$) is set to 100. Similarly columns 12-14 are the results obtained when $k$ is set to 1000. The number of different solutions generated is shown in columns 15 and 16. $10^6$ solutions were generated. We can see that the number of different solutions generated by the proposed methodology is nearly 8 times than the random generation method.

Ideally, if solutions are evenly distributed in search space, all the shortest distances with the corresponding nearest point should be identical. The difference between the distances should be very small. Hence lower the value of $\sigma_{DNP}$, better the distribution. $\delta_{DNP}$ is the ratio between $\sigma_{DNP}$ and $\overline{d}_{min}$. If the solutions are far apart from each other, then the value of $\overline{d}_{min}$ should be larger. Hence, when the value $\delta_{DNP}$ is smaller, the distribution of solutions is more even. In the case of K-Means Clustering, higher the cost, better the solution distribution.

From TABLE 5 we can see that the values of $\sigma_{DNP}$ and $\delta_{DNP}$ are smaller and the value of $\delta_{KM}$ is higher for the proposed method when compared to the other two techniques. Our technique helps to generate more evenly distributed solution with VCS.

Table 6: Coverage

| Scenarios | Time | | Coverage | |
|---|---|---|---|---|
| | M1 | M2 | M1 | M2 |
| 1 | 210 | 190 | 24.2 | 31.3 |
| 2 | 240 | 200 | 34.6 | 40.6 |
| 3 | 230 | 200 | 23.5 | 34.3 |
| 4 | 240 | 210 | 78.3 | 84.1 |
| 5 | 220 | 200 | 67.5 | 79.2 |

## 8 Case Study: CORTEX M0

In order to show the effect of the consistency search and domain clustering on coverage we used an ARM Cortex-M0. The Cortex-M0 processor is based on the ARMv6-M architecture. It has only 56 instructions. We chose the following 5 requirement of ARMv6-M core for our purpose:

1. Most 16-bit instructions can only access eight of the general purpose registers, R0-R7 known as the low registers.
2. A small number of 16-bit instructions can access the high registers, R8-R15.
3. Conditionally executed means that the instruction only has its normal effect on the programmers model operation and memory if the N, Z, C and V flags satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP.
4. Most of the instructions set the condition code flags, according to the result of the operation. If an instruction does not set a flag, the existing value of that flag, from a previous instruction, is preserved.
5. Shift and rotate instructions move each bit of a bitstring left or right by a specified number of bits.

These requirements are converted into various verification scenarios. The verification scenarios are then modeled using SystemVerilog constraints. These constraints are then used to generate the input stimuli required for verification. We run the experiment with and without domain clustering. For each scenario about 1200 instructions were generated. The TABLE 6 shows the coverage obtained. M1 represents the results obtained by constraint random test generation and M2 represents the results obtained by using the proposed domain clustering technique. From the experimental results we can see that using domain clustering we were able to attain higher coverage in almost the same time. In some cases the improvement in coverage is about 15%. This is because by dividing the search space into sub search space and generating solutions from the sub search space, increases the probability to generate solutions which are different from each other. The results show that by using the proposed methodology, the evenness of solution distribution can be increased.

## 9 Conclusion and Future works

In this paper, we presented a framework based on consistency search and domain clustering to improve the distribution of input stimulus generated by existing verification tools. Consistency search on the CSPs was able to reduce the domain of input variables. Domain clustering based on the results of consistency search helped to improve the distribution of generated input stimuli. Experiments showed that the proposed preprocessing stage helped to improve the distribution of input stimuli generated by VCS. Another remarkable advantage is that the proposed methodology can be easily integrated with other constraint random verification tools. In the proposed methodology the number of cluster is considered as an input parameter. In future, by analysis of the search space, we would like to determine number of clusters in the search space.

## References

[1] WEKA. `http://www.cs.waikato.ac.nz/ml/weka/`. Accessed: 16/6/2014.

[2] P. Baldi. Autoencoders, unsupervised learning, and deep architectures. *Journal of Machine Learning Research*, pages 37–50, 2012.

[3] E. Bin, R. Emek, G. Shurek, and A. Ziv. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386 –402, 2002.

[4] L. Devroye. Random variate generation for unimodal and monotone densities. *Computing*, 32(1):43–68, 1984.

[5] M. Iyer. Race a word-level atpg-based constraints solver system for smart random simulation. In *Proceedings of International Test Conference*, volume 1, pages 299–308, 2003.

[6] Z. Kong, S. Deng, J. Bian, and Y. Zhao. Even distribution evaluation in random stimulus generation. In *In Proceedings of 11th Joint Conference on Information Sciences*, 2008.

[7] J.G.M. Paret and O. Ait Mohamed. Coverage driven test generation and consistency algorithm. In *Declarative Programming and Knowledge Management*, Lecture Notes in Computer Science. 2014.

[8] A. J. Parkes. Clustering at the phase transition. In *Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, 1997.

[9] W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: exploiting random walk strategies. In *Proceedings of the 19th national conference on Artifical intelligence*, pages 670–676. AAAI Press, 2004.

[10] B. Wu and C. Huang. A robust general constrained random pattern generator for constraints with variable ordering. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 109–114, 2012.

[11] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using bdds. In *IEEE/ACM International Conference on Computer-Aided Design, 1999*.

[12] Y. Zhao, J. Bian, S. Deng, and Z. Kong. Random stimulus generation with self-tuning. In *13th International Conference on Computer Supported Cooperative Work in Design*, 2009.