CGA: COMBINING CLUSTER ANALYSIS WITH GENETIC ALGORITHM FOR REGRESSION SUITE REDUCTION OF MICROPROCESSORS

Liucheng Guo, Jiangfang Yi, Liang Zhang, Xiaoyin Wang and Dong Tong
School of Electronics Engineering and Computer Science, Peking University, Beijing, P.R. China
Email: {guoliucheng, yijiangfang, zhangliang, wangxiaoyin, tongdong}@mprc.pku.edu.cn

ABSTRACT
Regression testing plays an important role in the simulation-based functional verification of microprocessors. Regression suite is maintained in the entire verification phase with an increase of the scale. However, the executing cost is always high when running the entire suite on a RTL-level simulator. Regression suite reduction (called RSR for short) is presented to reduce the executing cost of the regression suite without debasing the quality of the functional verification. For this two-objective RSR of microprocessors, we present a heuristic algorithm which mainly combines cluster analysis with genetic algorithm (called CGA for short). The experiments on some regression suites at different scales for a microprocessor have shown the efficiency and feasibility of CGA. CGA can effectively reduce about 90% of the executing cost without decreasing the functional coverage in an acceptable runtime.

I. INTRODUCTION
Functional verification is one of the major bottlenecks in microprocessor design and takes above 50% of the design resources [1]. In current industrial practice, simulation-based functional verification is widely used and regression testing plays an important role. The regression suite is maintained in the entire verification phase and run after every hardware modification, every night or weekend, in order to check that no new bugs are introduced [2].

The tests in the regression suite come from many ways, such as handwritten tests, successive tests from the previous verification work and a massive amount of tests from the random generators. In different phases, these tests are added for different reasons, so the scale of the regression suite continues to grow and becomes a limiting factor in microprocessor verification [1]. Running the entire regression suite is costly and may take over a month [3], especially on the RTL-level simulator which is not efficient enough (less than 10 clock-cycles per second) [1].

Usually, the regression suite contains some redundancy, especially when the tests come from the random generators. For running the regression suite many times more efficiently, regression suite reduction is presented to eliminate the redundancy from the regression suite, without debasing the quality of functional verification which is measured by the functional coverage [4]. That is to say, RSR is a two-objective problem [5]. When all the executing cost of tests is equal, RSR is also an instance of the set cover problem which is known to be NP-complete [6].

In this paper, we present CGA combining the efficiency of cluster analysis and the capability of genetic algorithm to solve the two-objective RSR of microprocessors. CGA takes advantage of the capability of genetic algorithm as the basic algorithm, and creatively applies the cluster analysis to avoid the problems of genetic algorithm when applied to large-scale regression suites. CGA also uses the greedy algorithm to reduce unnecessary work and accelerate RSR. The experiments on some existing practical regression suites for a microprocessor have shown that CGA is more effective compared with the genetic algorithm and the greedy algorithm. CGA can generate a subset with the same functional coverage as the original regression suite, while the executing cost of the subset is only 10% of the original one’s. The experiments also show CGA is feasible even when the scale of the regression suite is large: the runtime of CGA is close to that of greedy algorithm and much less than that of genetic algorithm.

II. PRELIMINARY

<table>
<thead>
<tr>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>p_2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>p_3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p_4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>p_5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>p_6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Cost({t_1, t_2, t_3, t_4, t_5, t_6}) = {10, 30, 40, 80, 50, 120}

Figure 1: A coverage matrix and a cost array.

Fig. 1 provides a 5x6 coverage matrix and a cost array as the input of RSR. T = \{t_1, t_2 ... t_6\}, is a regression suite with n tests, while P = \{p_1, p_2 ... p_m\}, is a set of legal and interesting functional coverage points with m ones. The coverage
information is recorded by a 0-1 matrix \( M(P, T) \): "1" in row \( i \) and column \( j \) denotes that the \( j \)-th test \( (t_j) \) can cover the \( i \)-th coverage point \( (p_i) \), whereas "0" for the contrary. \( PD(t_i) \) is defined as the coverage degree to indicate the number of coverage points covered by \( t_i \), and \( Cost(t_i) \) indicates the executing cost of \( t_i \) spending on the RTL-level simulator, which is measured by clock cycles in this paper.

Then, \( P(T) \) is a set of coverage points covered by test suite \( T \) and \( PD(T) = |P(T)| \). Notably, \( PD(T) \) is not just the sum of \( PD(t_i) \) when \( t_i \in T \), because some tests may cover the same coverage points.

According to the descriptions above, we can present the definition of RSR: a process of finding a subset \( T_{\text{min}} \) from the original regression suite \( T_{\text{ori}} \) to minimize the cost function

\[
\text{Cost}(T_{\text{min}}) = \sum_{t_i \in T_{\text{min}}} \text{Cost}(t_i) \tag{1}
\]

subject to the constraint

\[
PD(T_{\text{min}}) = PD(T_{\text{ori}}) \tag{2}
\]

Besides, in the matrix \( M(P, T) \), there are some special coverage points, each one of which is covered only by one test. When such tests are not included in \( T_{\text{min}} \), those special coverage points cannot be covered. Those tests are called as the essential tests [7], which need not to be processed. In Fig. 1, \( p_2 \) is covered only by \( t_4 \), and then \( t_4 \) is in the essential test set \( T_{\text{ess}} \). Obviously,

\[
T_{\text{ess}} \subseteq T_{\text{min}} \tag{3}
\]

III. RELATED WORK

There are two major research directions for compressing regression suite. One is to generate a regression suite by the automatic test generator on-the-fly [2]. However, this approach is only possible when a powerful random test generator is available. Another research direction is regression suite reduction, which eliminates the redundancy from an existing regression suite and is the focus of this paper. The NP-completeness of RSR encourages the applications of the heuristics. The major ones applied include the greedy algorithm and the genetic algorithm.

Greedy algorithm was used to solve set cover problem in 1979 [8] and extended for single-objective RSR problem. Then there are some heuristics presented which can be regarded as the variations of the greedy algorithm [7][9][10].

To solve the two-objective RSR, a greedy algorithm ADD-DELTA is presented [10]. Ref. [5] also provides the additional greedy algorithm just like ADD-DELTA. It adds the \( t_i \) (1 ≤ \( i \) ≤ \( n \)) to \( T_{\text{min}} \) with the minimum \( \Delta \text{Cost}(T_{\text{ori}}) / \Delta PD(T_{\text{ori}}) \) every time, while \( \Delta \text{Cost}(T_{\text{ori}}) = \text{Cost}(t_i) \) and \( \Delta PD(T_{\text{ori}}) \) denotes the coverage points added. ADD-DELTA algorithm repeats the addition until \( PD(T_{\text{min}}) = PD(T_{\text{ori}}) \). In fact, however, greedy algorithms cannot ensure the quality of the solutions, Ref. [5] has proved that the additional greedy algorithm is not able to achieve the two objects. Experiments also show that none of the greedy algorithms is suitable for all the situations [11]. For example, in Fig. 1, when we use ADD-DELTA, \( T_{\text{min}} \) will be \( \{ t_1, t_2, t_3, t_4, t_5 \} \) and the executing cost is 210, but the best solution is \( \{ t_3, t_4, t_5 \} \) and the cost is just 170.

Genetic algorithm (called GA for short) can be understood as an intelligent probabilistic search algorithm and applied to a host of combinatorial optimization problems. Although may not ensure to get the best solution, GA can find approximate optimal solution with enough time, while the one by the greedy algorithm is certain and unchangeable.

GA has been applied to RSR recently [5][12][13][14]. According to the experiments [13][14], we can see GA is more suitable for small-scale problems. As a result, we choose it as the basic algorithm to process small-scale regression suites. When applied to large-scale one, however, GA has some serious problems: 1) chromosome becomes long due to the large scale of regression suite, which will increase the computational cost and decrease the convergence speed; 2) particularly, as the differences in the coverage and executing cost of tests are always large in a regression suite, GA may be impossible to find a feasible solution; 3) As an uncertain stochastic optimal algorithm, it is very hard for GA to estimate the theoretical time complexity [15]. Then GA may waste much time to try to achieve 100% of the original coverage. The experiments have shown the poor effect of GA for large-scale problems [15].

IV. CGA for RSR

For avoiding the above problems of GA, CGA creatively applies cluster analysis first to process the original regression suite, which can control the scale and avoid the problems caused by large differences in attributes of tests. Besides, CGA also uses greedy algorithm in some steps to reduce the unnecessary work and accelerate RSR. The steps of CGA are as follows:

Step 1. Pre-processing is done by \( \text{Simplify}(M, T_{\text{ess}}) \) to reduce the unnecessary work and add \( T_{\text{ess}} \) to \( T_{\text{min}} \). Then the updated \( T_{\text{ori}} \) and the simplified coverage matrix \( M \) are gained as the input;

Step 2. \( T_{\text{ori}} \) is grouped by cluster analysis according to the coverage degrees and executing cost of tests, supposing \( K \) groups are from \( T_{\text{i}} \) to \( T_{K} \);

Step 3. Greedy algorithm is used to select a group according to the coverage degrees and
executing cost of groups, assume \( T_i \), \( 1 \leq i \leq k \), then a new coverage sub-matrix \( M'(P(T_i), T_i) \) is generated;

**Step 4.** GA is used on the sub-matrix \( M' \) to get a reduced test suite \( T_{\text{red}} \). Then \( \text{Simplify}(M, T_{\text{red}}) \) is invoked again to update coverage matrix \( M \) and \( T_{\text{min}} \). We can get an updated coverage matrix \( M(P', T') \). Obviously, \( P' = P(T_{\text{red}}) \) and \( T' = T - T_{\text{red}} \).

**Step 5.** Check whether the number of rows of \( M \) is less than the threshold \( TH \). If it is not, go to step 1;

**Step 6.** Greedy algorithm is used to reduce \( T' \) to get a reduced test suite \( T_{\text{red}}' \) according to \( M(P', T') \), then add \( T_{\text{red}}' \) to \( T_{\text{min}} \).

Fig. 2 is the flowchart of CGA. The main parts of CGA which is labeled by (“A”, “B”, “C”, “D”) in the Fig. 2, will be introduced in following sections.

![Flowchart of CGA](image)

**A. Pre-processing with \( \text{Simplify}(M, T_{\text{ess}}) \)**

First of all, we can simplify the matrix \( M(P, T) \) to reduce the unnecessary work. According to the expression \( T_{\text{ess}} \subseteq T_{\text{min}} \), we do pre-processing with the \( \text{Simplify}(M, T_{\text{ess}}) \) like this:

1. Find every coverage point which is covered by only one test, then save all such tests to \( T_{\text{ess}} \);
2. Add \( T_{\text{ess}} \) to \( T_{\text{min}} \);
3. For every test \( t \) in \( T_{\text{ess}} \), delete \( P(t) \) from \( M \);
4. Check the updated coverage matrix \( M \). If there is no “1” in a column, namely a test does not cover any coverage points, it can be deleted from \( M \) since its original coverage points have been covered by \( T_{\text{min}} \). For example, in Fig. 1, if \( t_5 \) is added in \( T_{\text{min}} \) and then \( \{p_1, p_3, p_4, p_3\} \) are deleted from \( M \), \( \{t_1, t_2, t_3, t_5\} \) can be removed.

After the pre-processing with \( \text{Simplify}(M, T_{\text{ess}}) \), \( T_{\text{ori}} \) and coverage matrix \( M \) are suitable as the input.

**B. Grouping using cluster analysis**

There are some problems when GA is applied to the RSR of microprocessors. As the scale of the regression suite becomes large after long verification cycle, a very long chromosome is encoded, thereby enlarging the search space and reducing the speed of algorithm convergence.

Additionally, various new tests are added to the regression suite from different ways in different phases, some consisting of thousands of instructions and others only of several. The large differences in the coverage degrees and executing cost increase the difficulty to find a better solution. Using \( PD(T_i)/\text{Cost}(T_i) \) as the fitness function works well in some situations [14], but sometimes it fails to find a feasible solution. For instance, there are two individuals, \( Ind_1 \) and \( Ind_2 \). The fitness of \( Ind_1 \), \( PD(T_i)/\text{Cost}(T_i) = 10/5 = 2 \), is high, but the fitness of \( Ind_2 \), \( PD(T_i)/\text{Cost}(T_i) = 99/100 = 0.99 \), is low. Obviously, \( Ind_1 \) is a better solution according to the fitness, but in fact it has low coverage. \( Ind_2 \) has a high fitness just because of the lower executing cost. If individual \( Ind_1 \) enters into the next generation and \( Ind_2 \) is eliminated, the optimal solution gained by GA may not have a high coverage.

To avoid such problems, first we group the original regression suite with cluster analysis. Clustering is a process of grouping the data into classes or clusters, so that patterns in the same cluster are similar to each other, while patterns in different clusters are not [16]. For RSR of microprocessors, cluster analysis has three advantages: 1) the large-scale original regression suite is grouped into smaller suites processed one by one, which can control the size of chromosome and take advantage of the ability of GA when applied to a small-scale problem; 2) after getting some groups, we can decide which group is processed first. In CGA, we can use the greedy algorithm to choose the group with the highest coverage and lower executing cost; 3) grouping regression suite according to the coverage degrees and executing cost can ensure fewer differences in attributes within same group, avoiding the problems described above.

Regression suite of microprocessors is always in large scale and contains complex sharp of test space and many outliers. The algorithm chosen for cluster analysis should be effective when applied on large scale and not sensitive to the shape of regression suite space and outliers. CURE [16] can recognize arbitrarily shaped clusters and is robust
to the presence of outliers. CURE also has linear storage requirements and time complexity of $O(n^2)$ for low-dimensional data. Experiments have shown CURE is effective at finding 2-D spatial clusters [16], so we choose CURE algorithm to do cluster analysis in CGA.

Cluster analysis is based on the coverage degrees and executing cost of tests. Fig. 3 is a PD-Cost coordinate graph for the example described in Fig. 1. The X-axis denotes the PD($t_i$) while the Y-axis denotes the Cost($t_i$). From the figure, we can see that the measurement units and value between coverage degrees and executing cost are very different, which may affect the results of cluster analysis. So the attributes value of all the tests should be centralized and standardized before doing cluster analysis [17].

After getting some groups, we use a greedy algorithm [10] to choose the group with the highest coverage and lower executing cost, which has the potential benefit of decreasing the iterations in CGA and accelerating RSR. Having done many experiments, we decide that the more suitable size of each group for applying genetic algorithm was 50.

C. Genetic algorithm

Before using the genetic algorithm, the parameters of GA, such as representation of the chromosome, the genetic operators and the fitness function, should be carefully set according to the features of the problems. We use GAib [18] algorithm library and modify it to adapt well to RSR.

The first step of applying GA to a particular problem is to design a suitable representation for an individual’s chromosome. Each individual in the population is associated with a candidate for $T_{min}$. For $n$ tests in a regression suite, an $n$-bit binary string is used as the chromosome structure. If there are $u$ individuals in a population, the chromosome representation of individual Ind$_i$ is:

$$\text{Ind}_i = \langle b_1, b_2, \ldots, b_u \rangle,$$

where $b_k \in \{0, 1\}$, $1 \leq k \leq u$.

Each bit of the chromosome denotes whether a test is included in the candidate, test $t_i$ is in the candidate if $b_k = 1$. Fig. 4 shows the chromosome of an individual, the test suite mapped from it is \{t$_1$, t$_a$, $\ldots$, t$_n$\}, where $t_a$, $t_n$.

Then, according to the representation of the chromosome, the genetic operators can be specified. Binary mutation is used for the string of chromosome, which inverts some bits with a probability, namely changes $b_k$ from "1" to "0" or vice versa. This is to say, mutation can create the possibility that some tests appear or disappear in an individual. Besides, a simple one-point crossover operator is used to swap the segments of the two parent strings in CGA.

Since the coverage degrees and executing cost of tests in a group are almost equal after the cluster analysis, the fitness of individual Ind$_i$ can be calculated like this:

$$\text{Fitness} (\text{Ind}_i) = \frac{\text{PD}(T_i)}{\text{Cost} (T)}$$

We can see that a test with better fitness has higher coverage and lower executing cost.

According to the group size and many experiments, we can set the rate of crossover to 0.9, the ratio of mutation to 0.02 and the number of generations to 1000.

D. Post-processing with greedy algorithm

As an uncertain stochastic optimal algorithm, GA may not achieve 100% of the original coverage in a given time. So when the most of the coverage points are covered by $T_{min}$ and fewer are left, a deterministic algorithm can be used as the post-processing to achieve the two objects quickly.

CGA sets $TH$ as a threshold for this situation. When the rows of $M(P', T)$ is less than $TH$, namely $PD(T') < TH$, a greedy algorithm ADD-DELTA [10] is used for the remaining coverage points. Though the greedy algorithm has a weakness of not ensuring the quality, it can be ignored because few coverage points are left.

V. EXPERIMENTAL RESULTS

We do some experiments on three existing practical modules to compare CGA with ADD-DELTA greedy algorithm (AGR) [10] and genetic algorithm (GA) [14]. These modules are at different scales and all from the design of a superscalar microprocessor. $M1$ is the decoder module with about 100 coverage points; $M2$ is the renaming module with about 1000 ones; $M3$ is the pipeline module with about 10000 ones. Three groups of regression suites for them mainly contain the handwritten tests and a mass of random tests generated by a platform MicroGP [19], which
bases on the coverage model and can record the coverage information. In different phases or when the hardware design is modified, some new coverage points are added and tests are generated. We pick 10 regression suites from different phases for each module. Table 1 gives thirty regression suites for M1, M2 and M3. We can see that the scales of the regression suites for M3 are large and the total executing cost is huge. It may take much time to run such regression suites on the RTL-level simulator.

### Table 1: The regression suites for the three modules

<table>
<thead>
<tr>
<th>No.</th>
<th>Points</th>
<th>Tests</th>
<th>Cost (cycles)</th>
<th>No.</th>
<th>Points</th>
<th>Tests</th>
<th>Cost (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>1</td>
<td>70</td>
<td>45</td>
<td>7</td>
<td>89</td>
<td>1193</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>70</td>
<td>56</td>
<td>8</td>
<td>98</td>
<td>1286</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>80</td>
<td>68</td>
<td>9</td>
<td>103</td>
<td>1293</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>89</td>
<td>79</td>
<td>10</td>
<td>123</td>
<td>1528</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>98</td>
<td>89</td>
<td>11</td>
<td>134</td>
<td>2797</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>103</td>
<td>97</td>
<td>12</td>
<td>144</td>
<td>2861</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>123</td>
<td>108</td>
<td>13</td>
<td>189</td>
<td>3012</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>134</td>
<td>123</td>
<td>14</td>
<td>198</td>
<td>3121</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>144</td>
<td>137</td>
<td>15</td>
<td>208</td>
<td>3241</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>189</td>
<td>151</td>
<td>16</td>
<td>227</td>
<td>3361</td>
<td>23</td>
</tr>
</tbody>
</table>

In Table 1, every regression suite has a unique number, the amount of coverage points, the amount of tests and the total executing cost measured by clock cycles in RTL-level simulation. Then the coverage matrix $M$ and the cost array for every regression suite can be easily gained. For all the 30 regression suites, some horizontal comparisons are carried out on the aspects of the coverage gained by three algorithms, the effects of reduction and the runtime of algorithms.

### A. The coverage gained by three algorithms

The Y-axis in Fig. 5 denotes the coverage gained by each algorithm applied to the 30 regression suites which are shown on the X-axis. AGR can gain 100% coverage of the original regression suites, while CGA can also gain 100% since it uses greedy algorithm in the post-processing. As an uncertain optimal algorithm, GA can only gain about 75% of coverage even after many generations when applied on large scale. For comparing other aspects more clearly with the same coverage in the following experiments, we combine traditional GA with the greedy algorithm as the post-processing, and set the same threshold $TH$ as the one of CGA. When the most coverage points covered by $T_{\min}$ with GA and fewer left, greedy algorithm is used to achieve the two objects quickly. In this case, the main difference between GA and CGA is that CGA uses the cluster analysis and pre-processing.

### B. Effects of RSR by three algorithms

When all the three algorithms can gain 100% coverage, Fig. 6 compares the reduction effects by them. The Y-axis illustrates the percentages of $Cost(T_{\min})/Cost(T_{ori})$ for the 30 suites showed on the X-axis. Fig. 7 shows the average percentages of every module to compare the effects for different scales clearly. We can see that the executing cost of $T_{\min}$ generated by CGA is the least of all, which is about 10% of the original cost for different scales. For small-scale regression suites such as M1, GA can be effective, and that's why we choose GA as the basic algorithm. However, for large-scale suites of M2 and M3, GA has a very poor effect because of the long chromosome and the large differences in the attributes of tests. AGR can get about 85% reduction which is close to that of CGA, but in fact, CGA is much more efficient than AGA for RSR of microprocessors since the little reduction of executing cost can save much time in the RTL-level simulation.

![Figure 5: The coverage gained by three algorithms](image)

![Figure 6: The percentages of $Cost(T_{\min})/Cost(T_{ori})$ for RSR](image)

![Figure 7: The average percentages of $Cost(T_{\min})/Cost(T_{ori})$ of M1, M2 and M3](image)
C. Runtime of three algorithms

We compare the runtime of three algorithms when they all gain 100% coverage. Results are shown in Fig. 8, the Y-axis uses logarithm to denote the runtime in seconds while the X-axis shows the 30 regression suites. The AGR takes the least runtime because of its simplicity. As the scale of regression suite grows, GA takes increasing long time and becomes infeasible ultimately. CGA takes an acceptable run time which is about 10 seconds, since CGA uses cluster analysis to control the scale and reduces the computational complexity of GA significantly. CGA also uses the greedy algorithm repeatedly to reduce the unnecessary work and accelerate RSR.

VI. CONCLUSIONS

Functional verification is the major bottleneck in the microprocessor design and regression testing plays an important role. Regression suite always has some redundancy since the tests come from various approaches and are added in different phases. Regression suite reduction is presented for two objects: firstly, the executing cost should be minimized for running on the RTL-level simulator more feasibly; secondly, the functional coverage should be the same as the original suite to guarantee the quality of functional verification.

This paper presents a new heuristic algorithm CGA which combines cluster analysis and genetic algorithm to generate a reduced regression suite $T_{\text{min}}$. CGA can effectively decrease executing cost with the same coverage as the original suite. The experimental results of some existing practical regression suites for a microprocessor show that CGA achieves the best reduction effect compared with the greedy algorithm and the genetic algorithm.

There are some features about CGA: 1) CGA creatively applies the cluster analysis to RSR, which can control the scale of the suite and avoid the problems caused by large differences in the attributes of tests; 2) CGA takes advantage of the capability of genetic algorithm when the scale of sub-suite is small; 3) CGA also uses the greedy algorithm to reduce the unnecessary work and accelerate RSR. Besides, CGA is not aimed at deleting the tests from the regression suite; it just focuses on getting a suitable subset which can be more feasibly run for many times.

CGA has a strong applicability and can be applied to more fields, such as verification of other hardware design and software testing. However, CGA may need the attention of engineers to specialize the parameters.

REFERENCES


