Designing an Effective Constraint Solver in Coverage Directed Test Generation

Haihua Shen, Pengyu Wang, Yunji Chen, Qi Guo, Heng Zhang
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{shenhh, wangpengyu, cyj, guoqi,zhangheng}@ict.cn.cn

Abstract

As the complexity of processors grows, the bottleneck of verification remains in generating suitable test programs that meet coverage metrics automatically. Coverage directed test generation is a technique to automate the feedback from coverage analysis to test generation. It is very important to solve the constraint satisfaction problem for a flexible coverage directed test generator with complex variables. In this paper, we propose an effective constraint solver which combines constraint satisfaction problem’s algorithms and coverage directed test generation to address the challenges that arise from the practical verification problem. We implement the constraint solver in our practical coverage directed test generation platform, which has been used in the verification of an embedded processor. The efficiency of our approach has been demonstrated by the practical results.

1. Introduction

Design verification comprises a large portion of the effort in designing a multimillion-gate processor. Simulation-based verification tries to uncover errors of design by detecting circuits’ faulty behavior when deterministic or pseudo-random simulation vectors are applied.

Random test generation (RTG) technologies have been developed for about twenty years. Now the most commonly used RTG in verification of complex microprocessors is biased random test generation [1]. Designers and verification engineers usually specify biases manually to hit areas or specific tasks in the design that are not covered well. The analysis of coverage reports, and their translation to a set of biases always require significant engineering efforts. Coverage directed test generation (CDG) is a technique to automate the feedback from coverage analysis to test generation. Considerable effort is invested in CDG to find good ways to automate the process of effective bias generation. Although there are different features in different technologies developed by different study group individually, CDG must have two inherent technologies: 1) constraint models or language which is used to describe the relationship between different instruction fields and variables according to specifications, internal data structure and design architecture; 2) constraint solver engine, a mechanism for automatic construction of solutions that achieves consistency over complex constraints.

In this paper, we develop a set of solution algorithms based on traditional constraint satisfaction problem (CSP) algorithms along with a set of constraint models to improve the performance of CDG. Although it presents a set of new techniques, the primary motivation of our work is not to developed advanced technologies for pure constraint satisfaction problems, but to take advanced technologies into practice to solve the problem in CDG and to break our verification “bottleneck”.

The remainder of this paper is as follows. In Section 2, we briefly present the background of CSP in CDG. We then describe our constraint model design and our improved algorithms to solve the constraint satisfaction problem in CDG in Section 3, 4. In Section 5 we describe the framework of our CDG platform in which our effective constraint solver is planted. Finally, we present experimental results in Section 6, and give some concluding remarks in Section 7.

2. Background

2.1. General constraint satisfaction problems

In computer science, many problems can be considered as constraint satisfaction problems (CSP) described in constrain-based formulations. Generally speaking, a constraint satisfaction problem consists of a finite set of variables and a set of constraints. Each variable is associated with a set of possible values, known as its domain. A constraint is a relation defined on some subset of these variables and denotes valid combinations of their values. A solution to a constraint satisfaction problem is an assignment of a value to each
variable from its domain, such that all the constraints are satisfied [2]. Constraint satisfaction problems can be divided into Boolean constraint problems, limitation constraint problems, numerical value constraint problems, and combination constraint problems by domain properties. Many references refer to different technologies for solving constraint satisfaction problems, which is known to be a NP hard problem. There is an international workshop named CSLP which is held annually only for the study of Constraint Solving and Language Processing. It is difficult to present all the results of extensive research and practice for constraint satisfaction problems in plenty of references. In this paper, we do not focus on general technologies for pure constraint satisfaction problems, but focus on technologies that solve the CSP problem in coverage directed test generation practice.

### 2.2. Constraint satisfaction problems in CDG

CSP has new properties when combined with new real applications, although it is a problem that has been studied by many groups in the world for many years. In this section we describe a number of properties of the CSP technologies, especially particular to CDG that probably raises new challenges.

CSP in CDG has three properties: large variable domains, well-distributed random sampling of the solution space and complex constraint topology [2].

The architectural and microarchitectural resources and their content must be described in detail in CDG. In modern complex design, especially advanced processors, these resources, such as operators, registers, memory, etc, can take a very large number of values. So the variables describing these resources have very large domains. It is impossible to examine all these large domains and their combination exhaustively with full coverage. People can divide the whole large test space into many subspaces and thoroughly verify some subspaces with high priorities by directed test generation. And CDG is another important way to cover the unspecified space. According to probability theory, the best way to generate test vectors is to uniformly sample the derived test space because the well-distributed random sampling of large variable domains in CSP may enlarge the possibility to find bugs in design under test.

It is hard to solve the problem that how to cover large variable domains and generate well-distributed random samples of the solution space. But compare with the complex constraint topology, the problem is trivial. Variables and constraints are commonly simple and fixed in traditional constraint satisfaction problems. However, many real-world problems, such as CDG, are difficult to be modeled in simple and clear constraints with well-defined variables. There are plenty of constraints that affect different portion of different variables in different topology in random test generation. Furthermore, the constraints in the complex topology may be constructed to a network with different hierarchy, priority, even contradiction. All these features in CDG bring new challenges to constraint satisfaction problems.

There are several algorithms that have been developed to solve the CSP in constraint random spaces of CDG.

Yuan presented an algorithm based on Binary Decision Diagrams (BDD). BDDs are data structures commonly used to represent Boolean functions. The advantage of the algorithm is that it can generate solutions in well-distributed way. The disadvantage is that BDD is less efficient than others when handling many variables. It runs in the exponential growth model at time and computer memory. This algorithm is used in SystemC verification libs.

Mahesh A. Iyer built a system named RACE based on Boundary Model Checking (BMC). A constraint network was constructed in RACE by analyzing the constraint conditions. Then the BMC was used to solve the satisfaction problem in the constraint network. RACE has been used in a commercial product. But it is difficult for RACE to get a solution space in uniformly sampling way because it is hard to predicate the solution space in BDD.

Bucket Elimination is a unifying algorithmic framework that generalizes dynamic programming to accommodate many complex problem solving and reasoning activities. Algorithms such as directional resolution for propositional satisfiability; adaptive consistency for constraint satisfaction; Fourier and Gaussian elimination for linear equalities and inequalities; and dynamic programming for combinatorial optimization can be all accommodated within this framework [5][6]. Bucket Elimination algorithm (BEA) has well-proportioned solution for complex CSP. But it runs in the exponential growth model at computer memory that limits the size of problem to some extent. It is important to present improved methods to minimize the computer memory usage in this algorithm.

It has been mentioned above that there are plenty of variables with large value domains and complex constraints in the CSP of CDG. It is hard to enumerate all the possible scenarios that can lead to bug discovery. But it is not so difficult to get a group of solutions to satisfy the constraint network because of the large solution space. The challenge is that how to sample
solutions of variables uniformly to ensure well-distributed test scenarios for high quality bug discovery.

We develop a Flexible Small Buckets Algorithm (FSBA) which has strong ability to get well-proportional solutions is suited to the properties such as large search spaces, complex constraint topology associate with CDG, along with a set of improved algorithms for memory saving and continuous variables handling, etc. The following sections discuss the constraint model in CDG and our FSBA algorithms.

3. Constraint model

Constraint model construction is essential to both general random test generation and CDG in real application. Generally speaking, a constraint solver includes constraint models, which describes all large domains of variables and their relationship, and a set of algorithms that solve the constraints problem according to the constructed constraint model.

Constraint models in CDG can be divided into two groups: general constraints and special constraints. General constraints describe variables and their relationships in the same way as general CSP. Special constraints describe special construction and verification knowledge of CDG, such as instruction sets, memory hierarchy, instruction alignment issues, branching problems near segment boundaries, etc. In this section, we discuss detailed general and special constraint models in CDG. Algorithms, which solve the constraints problem, will be described in the next section.

3.1. Special constraints

Special constraints define constraint models in various forms according to design specifications including architecture, microarchitecture, instruction specifications, etc. It is necessary to describe all these knowledge and specifications correctly, completely and unambiguously.

Formal specifications, based on using multiple constraints to collectively define the behavior of test generation, promise to meet all requirements. Our formal specification focuses on domains of instruction set and resources, which describe memory hierarchy, branching problems, segment boundaries, and so on. Figure 1 shows the special constraint models defined in CDG.

**Instruction define** defines all the instructions and their combinations according to instruction specifications. Both Instruction group and Instruction sequence are used to define a group of instructions. Instruction group define a group of instructions which can include another instruction group in recursive way without complex constraints. But Instruction sequence

![Figure 1 Special constraint models](image)
defines a group of instructions according to strict constraints, which are defined in resource forms. Instructions can be generated in configurable styles from pseudorandom groups with different ratios to totally directed sequence with strict constraints according to requirements from coverage driven engine in CDG.

Separated by formats, allowed instructions are divided into several groups. I type includes a kind of instruction with only two src operands and dst operand in registers. M type includes instructions that access memory resources. J type & B type are all instructions whose target is to change program counters. The difference between them is that instructions in B type have branch conditions and those in J type have not. Function I/M/J/B (instruction var) is defined to find if the instruction var is in I/M/J/B type. The constraints should be written according to the following reasoning:

♦ \( \text{prec}(I \cap \neg M \cap \neg J \cap \neg B) \rightarrow Rsrc1 \lor Rsrc2 \lor \neg Rdst \)

♦ \( \text{prec}(\neg I \cap M \cap \neg J \cap \neg B) \rightarrow (Rbase \lor ((Rsrc2 \lor \neg Rdst) \land \neg (Rsrc2 \land \neg Rdst))) \lor (\text{Mstr} \lor \text{Mend}) \)

♦ \( \text{prec}(\neg I \cap M \cap J \cap \neg B) \rightarrow \text{Target} \)

♦ \( \text{prec}(\neg I \cap M \cap J \cap B) \rightarrow Rsrc1 \lor Rsrc2 \lor \neg \text{Toffset} \)

Besides I/M/J/B types, there are other instructions with various formats. We group them into one type.

There are many resources that must be translated into constraints in CDG (see figure1 in light grey). Constraints on the registers can be written in simple forms. Compared with registers, constraints on memory are more complex. Several forms are defined to describe complex variables in fields like memory.

The constraint models are powerful enough to specify the requirement of generation from random to totally direction according to CDG. Now an instruction can be described based on define forms above. For an example, we can describe an instruction in following syntax:

Instruction_define | instruction_group | instruction_sequence [operation variable constraint1] ...[operation variable constraint \(*\)times]

Although abiding by the constraint models may seem restrictive, it promises many benefits. For one, the specification is easier to maintain. Constraints can be added or removed and independently modified. It is also believed that it is easier to write and debug. Since most existing languages already written as a list of rules with variables and constraints, the translation to this type of specification requires less effort and results in fewer opportunities for errors.

### 3.2. General constraints

Constraint models also describe variables and their relationships for general CSP. Several forms are defined to describe complex variables in all kinds of fields.

**Varialbe define** is used to define variables with many different value scopes. It is written as figure2 (a). The scope of variables can be continuous, incontinuous or enumerate. Figure2 (b) shows constraint models of different variable fields. **Constraint define** is used to define relationships among variables such as Greater than, Less than, Equal to, etc. **Constraint define** can be used in the whole generation model. It is defined as figure2 (c).

![Figure 2 General constraint models](image)

**Table 1** Relations supported by the constraint model

<table>
<thead>
<tr>
<th>sign</th>
<th>arguments</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>arg1:var</td>
<td>arg1 == arg2</td>
</tr>
<tr>
<td>ne</td>
<td>arg1:var</td>
<td>arg1 (\neq) arg2</td>
</tr>
<tr>
<td>gt or &gt;</td>
<td>arg1:var</td>
<td>arg1 &gt; arg2</td>
</tr>
<tr>
<td>lt or &lt;</td>
<td>arg1:var</td>
<td>arg1 &lt; arg2</td>
</tr>
<tr>
<td>ge or &gt;=</td>
<td>arg1:var</td>
<td>arg1 (\geq) arg2</td>
</tr>
</tbody>
</table>
### 4. Flexible small buckets algorithms

#### 4.1. Flexible small buckets algorithm for CDG

Generally speaking, the constraint satisfaction problem can be represented as a triplet \((X,D,C)\). Let \(X=\{X_1, \ldots, X_n\}\) be as set of random variables over multivalued domains \(D=\{D_1, \ldots, D_n\}\). To simplify the problem, \(X\) is a set of discrete variables. \(C=\{C_1, \ldots, C_m\}\) is a set of real-valued cost functions. Each function \(C_i\) is defined over a subset of variables \(S_i\). Suppose \(s \in \bigcup S_i\), if \(s\) satisfy the constraint, then \(C_i(s) = 1\). Else \(C_i(s) = 0\). Given a calculative function \(N^i_1 = \{X_1, X_2, \ldots, X_{i-1}\}\), the value of \(N\) is equal to the number of assignments \((X_1, X_2, \ldots, X_n)\) to \(X=(X_1, X_2, \ldots, X_n)\) that make the cost functions true.\[5\]

Our Flexible Small Buckets Algorithm (FSBA) for CDG is described here:

**Input:** A belief network \(BN = \{P_1, \ldots, P_n\}\); an ordering of the variables, \(d = X_{i_1}, \ldots, X_{i_n}\); evidence \(e\).

**Output:** The belief in \(X_{i_1} = x_{i_1}\).

1. Choose an ordering of the variables, \(X_1, X_2, \ldots, X_n\); and construct the ordering of buckets, \(B_1, B_2, \ldots, B_n\).
2. Generate an ordered partition of the conditional probability matrices, \(bucket_1, \ldots, bucket_m\), where \(bucket\) contains all matrices whose highest variable is \(X_i\). Put each constraint variable in its bucket.
3. For \(p \leq n\) downto 1, do:
   a) If there are calculative functions \(N_1, N_2, \ldots, N_i\) in bucket \(B_p\), then divide \(B_p\) into \(m\) small buckets \(B_{p1}, B_{p2}, \ldots, B_{pm}\).
   b) Let \(S_{pi}\) be the set of all variables in bucket \(B_{pi}\) except for \(X_{i_p}\). For all small buckets \(B_{pi}\), generate new calculative function \(N^p_i(s)\), where \(s\) is the vector defined on \(S_{pi}\).
   c) Put \(N^p_i(s)\) into the bucket that has the highest subscript in \(S_{pi}\).

The complexity (time and space) of the algorithm is \(O(n \times \exp(\Omega^*(d)))\), where \(n\) is the number of variables and \(\Omega^*(d)\) is the induced width of the ordered moral graph.

The algorithm neglects the original condition that all sub-variables must be equal in many other algorithms. It sacrifices accuracy to some extent because of the enlarged solution space. So we must put some constraints on new calculative functions to improve our algorithm.

At first, the small buckets are still generated by original calculative function \(N^{p*}_1, N^{p*}_2, \ldots, N^{p*}_m\) according to the original formula (1) in the original algorithm:

\[
N^p_i(s) = \sum_{x \in X_p} \prod_{v \in N_p} V(s,x) \tag{1}
\]

Then we calculate the final \(N^p_i\) according to the following formulas (2)(3).

\[
\rho_i(x) = \frac{\sum_{j \neq i} \min(\max N^{p*}_j(s),1)}{\sum_{p \in m} \min N^{p*}_j(s)} \tag{2}
\]

\[
N^p_i(s) = \sum_{x \in X_p} \rho_i(x) \prod_{v \in N_p} V(s,x) \tag{3}
\]

The new factor in formula (2) is defined for estimating the compatibility between small buckets when assigning values to \(X_i\) in bucket \(B_{pi}\). It represents the equality of sub-variables approximately.

#### 4.2. Sequence selection algorithms for variables

It is known that the size of the induced width varies with various variable orderings and each ordering has a different performance guarantee. The times of small buckets split and the cost of computer memory space can be balanced for high performance if we analyze constraints of variables before they are taken into the algorithm. We develop an algorithm to arrange the sequence of variables for CDG as following:

**SEQ algorithm**

1) For each \(X_i \in X\), let \(D_i\) be the set of variables to which \(X_i\) relates to.
2) For each constraint of \(X_i\). Let \(S\) be the set of variables to which the constraint relates to.
3) For each \(D_i \in D\), do
   \[D_i \leftarrow D_i \cup S - X_i\]
4) After calculate \(D_i\) for all variables, sort variables according to \(|D_i|\) from max to min.
The SEQ algorithm is based on the policy of traditional greedy algorithms. The ordering of the variables after SEQ algorithm is used in step1 of Flexible Small Buckets Algorithm in section 4.1.

4.3. Improved algorithm for continuous variables

To simplify the problem, constraints are all defined as a set of discrete variables in algorithms above. But constraints in real applications are always continuous. In fact, constraints of CDG are all continuous in integer field except for register number. The complexity of calculative function is time and space exponential in the induced large fields of continuous variables. To reach continuous variables to algorithms above, we must find a way to translate the continuous variable into discrete one.

The domains of a continuous variable can be represented as a triplet \((l, h, w)\), where \(l\) is the low limit of the domain, \(h\) is the high limit of the domain and \(w\) is the weight. The domain of a continuous variable may be one or more. For an example, \((0 \ 127)\ (128 \ 255)\) can be used to represent the domain of a byte. We have assumed that \(X = \{X_1, ..., X_n\}\) is a set of random variables over multivalued domains \(D = \{D_1, ..., D_n\}\). If \(X\) is a set of discrete variables, the improved algorithm for continuous variables is as following:

**Improved algorithm for continuous variables:**

1) For each continuous random variable \(X_i \in X\), do
   a) Choose \(t\) values from its domains randomly according to \((l, h, w)\) which gives the low limit, high limit and weight.
   b) Arrange \(t\) values chosen in a) to be the new temporary discrete domain of random variable.

2) Use the improved split bucket elimination algorithm to solve the CSP problem.

3) If there is no solution or the number of solutions equals to a ceiling predefined, then return 1. Else break.

The performance of the algorithm depends on whether the large continuous variable domains are sampling uniformly. \(t\) is the key parameter close to the performance. The larger \(t\) is, the more time and space the algorithm spends. On the contrary, there may be no solution on a tiny domain because of complex contradiction if \(t\) is too small. So we must choose the \(t\) carefully.

Step3 of improved algorithm for continuous variables ensure the variety of assignment values of variables. According to random policy, plenty of well-distributed random sampling of large variable domains may find more bugs in design under test. The algorithm will turn to step1 to redo the sampling if the number of solutions has been accumulated to a predefined ceil value \(n\). Constructing the ordering of buckets frequently may lower the performance if the predefined \(n\) is too small. We find in practice that the algorithm in CDG can get well performance if \(n\) is 1 or 2 times of \(t\).

5. Coverage directed generation platform

An effective constraint solver based on constraint models and algorithms described in this paper has been taken into practice on CDG, a real coverage directed verification platform for series of Godson processors.

Since biased random test generation technique has been one of the most commonly used verification technique in industry practice. The CDG scheme employs the ability to efficiently analyze and use the random test generation technique which has been developed for many years.

Figure 3 illustrates the structure of CDG platform and the verification environment we implement, which uses CDG task driver with genetic algorithms to close the loop of coverage analysis and test generation on the basis of CRPG, a biased RTG platform developed before [7][8]. The CDG platform is used to verify an embedded processor based on improved Godson1 with 900K logic gates. We run the verification environment on clusters with 10 nodes, each has a dual Pentium4 CPU. Hundred thousands of functional coverage points have been defined by designers and verification engineers using SystemVerilog. Practical results show that the CDG platform with our effective constraint solver can solve the constraint satisfaction problem in CDG effectively and improve the coverage progress significantly (see next section).

![Figure 3 The framework of CDG](image)
6. Experimental results and applications

Experimental results have been presented by considering two key aspects. One considers the performance of proposed flexible small buckets algorithms directly. The other aspect is to show how much the CDG platform is improved so that the performance of constraint solver can be demonstrated indirectly.

Finding a uniformly distributed random solution is the most important goal of random test generation related algorithms. The more equivalent the numbers of solutions on different domains are, the more uniformly the algorithm samples the derived test space. Given a constraint network with m solutions, each solution can be generated by $t_i$ times. The whole number of solution vectors generated is n. The difference of distributions is described by formula (4).

$$D = \frac{m}{n} \sqrt{\frac{\sum_{i=1}^{m} (t_i - \frac{n}{m})^2}{m}}$$  \hspace{1cm} (4)

It is definitely that the less D is, the better distributed the solutions are. Figure 4 is the sample distributions of different random CSP algorithms. To judge the algorithms objectively, the size of problem is limited so as to explore the whole solution space. The variables in constraints have four domains between 9 and 12. The number of constraints is between 15 and 25. It is obvious that the whole distributions are uniform more and more along with the increment of solutions.

![Figure 4 Sample distributions of different random CSP algorithms](image)

In Figure 4, SCV is the algorithm developed by Yuan based on BDD. SV is the random CSP algorithm which bases on RACE system in SystemVerilog. MBE0 is the original bucket elimination algorithm. MBE1, which is based on MBE0, is an improved algorithm developed by Dechter in 2002. We find in experience that performance of MBE1 is not as good as the author analyzed because it has not given well relationship of equivalent sub-variables. The performance of MBE1 is close to that of MBE0 in our experiment. The performance of SV is not good because it never predicts the solution space according to its design principles. FSBA, our flexible small buckets algorithm, has the good performance near to SCV, which have the most uniform distribution performance in all the algorithms in experiment. The cost of highest performance is the most memory space it sacrifices. Figure 5 shows the memory space different random CSP algorithms spend. The performance of SV is not described in Figure5 because it is difficult to extract the CSP engine from the SystemVerilog simulator and estimate the memory space of SV alone. However, it can be proved in theory that SV spend more memory space than BE because the solution space of BDD is limited by the path-width which is always more than induced-width [3]. The memory space of FSBA has not got more advantages than that of NAIVE, which is the roughest algorithm to solve random CSP. In fact, FSBA can deal with larger size of problem with more complex constraints and relationship among variables than NAIVE by arranging different granularities of small buckets. Compared with other algorithms, FSBA get the best tradeoff between performance and memory space cost.

![Figure 5 Memory space different random CSP algorithms spend](image)

We have implemented a constraint solver using FSBA algorithms in our coverage directed generation platform------CDG, which has been taken into practice in the verification of an practical embedded processor and finds several high quality bugs. The main goals of CDG are to improve the coverage progress rate, to help reaching uncovered tasks, and to increase the hitting rates to hard-to-reach coverage cases. Figure6 illustrates the comparison of coverage progress between our original random test generation platform (CRPG) and CDG. It is clearly that CDG can increase the coverage of tasks rarely covered before, decrease
the tasks uncovered and improve the coverage progress. CDG finds many bugs, some of which are high quality bugs found after Linux booting. Considering the complexity of the processor, we are satisfied with the quality we achieve at first tape release. The first prototypes are capable of booting the operating system and running applications correctly. For more details of CDG, please refer to [8].

![Figure 6. Coverage progress of random CRPG and CDG](image)

7. Conclusions

This paper highlights the need for flexible constraint solver which combines traditional constraint satisfaction problem’s algorithms and coverage directed test generation to address the challenges that arise from the practical verification problem. A novel constraint solver, which includes a set of constraint models and flexible small buckets algorithms (FSBA) is proposed and implemented in a practical coverage directed generation platform-----CDG, which has been taken into practice for the verification of a Godson embedded processor and finds several high quality bugs. Experimental results demonstrate that the proposed flexible small buckets algorithms (FSBA) have better performance than traditional algorithms when dealing with large solution space to meet uniformly well-distributed samples. Practical results also show the efficiency of CDG platform. Our future work focuses on exploring more techniques other than constraint solvers that may increase the capabilities of CDG process.

8. Acknowledgment

The author would like to extend thanks to Chaoji Li for his valuable contributions to this work. This work is supported by the National Basic Research Program 973 (No.2005CB321600), the National High Technology Development 863 Program of China (No.2007AA01Z112 & No.2008AA110901), the National Natural Science Foundation of China (No.60603049) and the Beijing Natural Science Foundation (No. 4072024).

9. References:


