# SOLVING OF CONSTRAINT SATISFACTION PROBLEM

**Ondřej Čekan, Marcela Šimková**
Doctoral Degree Programme (1,3), FIT BUT
E-mail: icekan@fit.vutbr.cz, isimkova@fit.vutbr.cz

Supervised by: Zdeněk Kotásek
E-mail: kotasek@fit.vutbr.cz

**Abstract**: The goal of this paper is to find a suitable solver for *Coverage Directed Test Generation* that is one of the functional verification techniques. In order to do this, we summarized information about the *Constraint Satisfaction Problem*. The problem consists of finding a solution (assignments for variables) that must satisfy certain constraints. Moreover, we created a survey about existing constraint solvers and compare their pros and cons. In the final part of the paper we propose a solution how to integrate an appropriate constraint solver into the process of test generation.

**Keywords**: Constraint solver, Constraint Satisfaction Problem, Coverage Directed Test Generation

## 1 INTRODUCTION

These days, more and more emphasis is given to the testing of the accuracy of the circuit's behavior. Today's integrated circuits are very large and complex, so the earlier techniques used for testing the correctness of hardware are not sufficient. Number of new techniques and tools that are intended to detect errors in the circuit are being developed. Errors can be caused by faults in the design or manufacture. In the foreground is a notion of functional verification which is used by large companies such as IBM, Cadence, Synopsys or Mentor Graphics [2].

The basis of functional verification is a reference model [8] which performs the function according to the specification and its output is then compared with the tested circuit. An important element described in this article is a generator of test vectors that generates inputs for the verified circuit. These must comply with certain constraints. The outputs of this component are essential to thoroughly test the circuit. It is profitable to generate test vectors automatically and accurately. The main principles of such generator is described in Section 2. Section 3 focuses on the *Constraint Satisfaction Problem* (CSP) whose purpose is to find values of variables that satisfy some restrictive conditions. Section 4 shows several clues how to solve the CSP. It also describes constraint solving and typical algorithms. Section 5 proposes our solution for generating test vectors and some concluding remarks.

## 2 COVERAGE DIRECTED TEST GENERATION

*Coverage Directed Test Generation* (CDTG) [1] [7] is one of the latest techniques for the verification of large designs. This method generates test vectors according to the defined conditions and limitations which are called *constraints*. The main challenge for generating test vectors is to achieve maximal coverage of circuit functions. As some features of the circuit may still remain unverified, it is necessary to specify additional constraints. Therefore, the CDTG guide us to create these constraints from the coverage analysis in order to achieve as largest coverage as possible. Thus, also the uncovered portion of the circuit can be verified as is shown in Figure 1.

Although various CDTG techniques are used in different technologies developed by different groups independently, they contain two common parts: Constraint model/language and Constraint solver. To describe the restrictive conditions, we can use a constraint model. To find the solution or solutions for these constraints, we can use constraint solver engine.

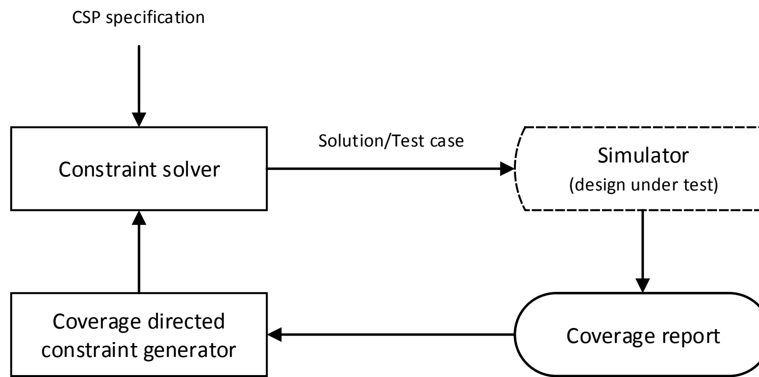


**Figure 1:** Coverage directed constraint random test generation.

By introducing CDTG we can gain two significant advantages. There is a possibility that the uncovered scenarios will be covered and a higher level of coverage will be achieved. The second advantage is that certain scenarios will be tested multiple times with different inputs.

Most problems in computer science that must satisfy certain constraints are special cases of the CSP or at least, they can be transformed into it.

## 3 CONSTRAINT SATISFACTION PROBLEM

Constraint Satisfaction Problem (CSP) [1] [4] [5] is a general mathematical problem defined as a set of variables that can take values from a finite and discrete domain and a set of constraints. The constraint is defined on a subset of variables and determines values from the domain that a variable can take. The result is a solution of one or all evaluations of variables so that the constraints are satisfied.

Among the typical examples of CSPs are N Queens problem, Map-Coloring problem (these two problems are described in the following text), Car sequencing problem, Magic Square, Social Golfers and more.

**The N Queens Problem**

The N Queens problem [4] is known from the chess game. On the playing board with dimensions N*x*N it is necessary to place the N chess queens so that diagonally, horizontally and vertically they do not jeopardize each other. The Queen can move in the same row, column or diagonal. The problem of the placement of the queens on the board, that have to fulfill certain restrictions, is the typical example of CSP.

**The Map-Coloring Problem**

The Map-Coloring problem [5] can also be solved as a CSP. The problem consists of assigning colors (from a domain) to each region on the map so that two adjacent regions do not have the same color. This problem can be transformed into the constraint graph, which is equivalent to the CSP. Each region of graph represents one variable and their mutual borders represent relationships and constraints between them.

## 4 CONSTRAINT SOLVER

As stated above, the solution to the CSP is assigning a value to each variable so that all imposed constraints are simultaneously satisfied. This raises the question whether there is a solution to a given CSP? This is the so-called NP-complete [3] decision problem, therefore, it cannot be decided in a deterministic polynomial time. An environment for solving the CSP is called Constraint Solver.

A scheme of a constraint solver is shown in Figure 2. It reflects the main principle of how the most solvers work. The first element only pre-processes a task of the CSP. The search element works on

the backtracking principle in the conjunction with the constraint propagation. Assigning a value to a variable is statical or is based on a heuristic and then a depth-first search or other searching algorithm can be performed. Backtracking is applied when a conflict in an assignment is detected. The simplify element contains a queue of constraints and performs their promotion. On the basis of this promotion, values are taken from the domain of variables.

There are several techniques that are used for solving the CSP hence several types of them are described in the next paragraph.

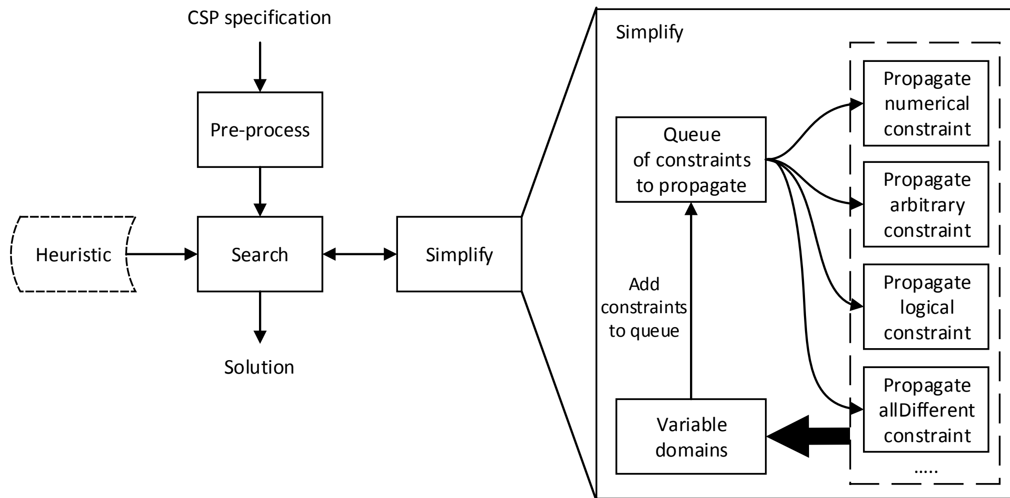


**Figure 2:** Scheme of a constraint solver.

**Generate-and-Test**

This method is the simplest possible way to solve the CSP. Generate-and-test [5] method systematically generates all possible combinations of values for the variables and then checks whether all constraints are satisfied. If they are, the solution was found, if not, it generates the next combination. The number of combinations that this solution can generate is equal to the size of the Cartesian product of the variable domains.

**Backtracking**

The second option which will be shown is the method called backtracking [5]. This method is known and used for decades. In contrast to the previous method, this method does not assign values to all variables directly but initializes variables sequentially and continuously verifies the validity of the restrictions. If any constraint is violated, assignments of variables are returned to the last valid instance that has another alternative assignment. Backtracking performs a depth-first search. Thanks to backtracking, it is possible to partially eliminate some of the violating passages thereby it reduce the subspace of the Cartesian product.

Although this method is better than the previous one, there is a problem with exponential time complexity for non-trivial problems. Therefore, there are other methods based on backtracking with some extensions and improvements known as intelligent backtracking or systematic backtracking.

**Propagating Constraints**

Another frequently used method for finding solution is the method based on the Propagating Constraints [5] [6]. The Propagating Constraints method shows another way to solve the CSP. This method is based on two principles. The first principle is the propagation which aims to reduce the search tree in a way that removes values that do not contribute to the solution. The second principle is to interleave enumeration (also called splitting or branching) that creates a new branch in the search

tree. Enumeration always creates two branches, one branch for a valid instance variables ($x = a$) and the other branch for an invalid instance ($x \neq a$). The second branch is used in the case of a constraint violation at the first branch and serves as an alternative way to represent backtracking.

**Hybrid Approaches**

There are many other techniques [6] that include various combinations of previous approaches and other innovative approaches that belong to the hybrid techniques.

### 4.1 EXISTING SOLVERS

This section shows some existing solvers which are open source and still under development. There is no space for implementation details, only the basic characteristics of selected solvers are demonstrated. This section is based on references [3] and [4].

**ECLiPSe**

ECLiPSe is one of the oldest environments for solving the CSP. It is programmed in Prolog, therefore, it does not reach high performance as similar systems programmed in modern programming languages. In contrast, it is possible to specify the problem better.

**Minion**

Minion is a constraint solver based on the Propagating Constraints approach. It is written in C++ language. The main advantages of Minion are speed and efficiency in solving many problems and it requires only one input file to run. In contrast to other solvers, Minion does not require additional piece of code.

## 5 THE GOALS OF THE FUTURE RESEARCH AND CONCLUSION

The main purpose of this survey is to compare existing constraint solvers and select the proper one for our future research. As was shown above, each solver is unique in some way. We intend to apply a constraint solver in the process of CDTG in functional verification of electromechanical applications in order to verify and increase their correctness and reliability. In particular, when functional verification is used for checking system properties, it is necessary to create as many testing scenarios (they consist of several variables) as possible in a restricted time. An important condition is that they need to cover all important system properties.

We propose CDTG with a probabilistic model as shown in Figure 3. The basic element is the Constraint Solver which will work with two types of constraints. The first are *design constraints* that specify restrictions on input values for a specific hardware system. A typical example of a design constraint is following: let's suppose a signal XY with 2-bit width. There are four values that XY can take: 00, 01, 10, 11. However, the combination 11 is invalid, because the system doesn't use it. Therefore, by a design constraint we can tell a generator not to produce invalid data for XY with value 11. In contrast to these type of constraints, we define *probabilistic constraints*. These constraints are generated from the probabilistic model. This model defines the probability that a specific value will be assigned to a specific input signal (in order to target coverage holes, some values are more prefered than others) based on the coverage report. If the probability of some value is zero or too low, the constraint solver can be optimized as it does not take some parts of the search tree into account. Figure 3 shows an example of a probabilistic model for Arithmetic Logic Unit (ALU). Input signal OP (operation) can be set to different values that represent different operations (ADD, SUB, MUL, etc.). As you can see from the picture, according to our method, every operation has its own probability that it will be generated in the next run of CDTG.

In our opinion, backtracking and constraint propagation cannot be significantly accelerated and moreover, there is no more space for further development because solvers on these approaches are already created. We expect that our solver will have a hybrid nature due to the probabilistic model (it will

take into account the probability in generating tests) and will combine features of existing solvers and probabilistic methods.

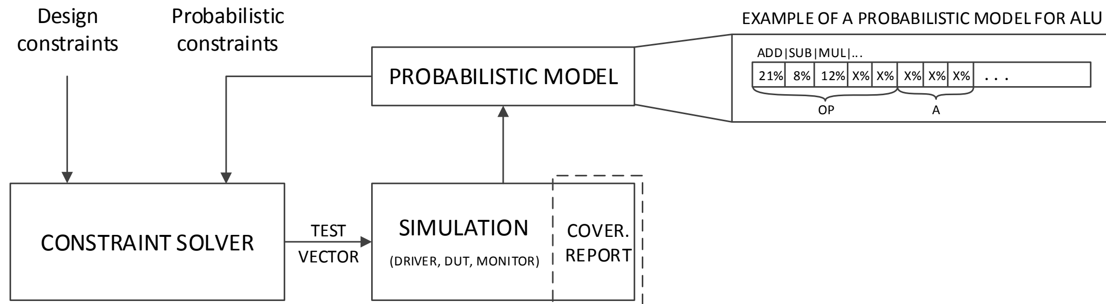


**Figure 3:** A scheme of the CDTG with a probabilistic model.

The current trend in research is developing a solver that is able to find a solution for complex problems with regard to the computation time. To meet these requirements, hybrid models are appropriate. A key priority for effective functional verification is developing the constraint solver for CDTG which is able to achieve maximum coverage of features (functionality, code, structure, etc.) of a system.

We have introduced a new method for CDTG that utilizes a probabilistic model. To the best of our knowledge, there doesn't exist any similar work. This article represents a survey at the beginning of Ph.D. studies, therefore, it is rather theoretical and shows future steps in our research.

## ACKNOWLEDGEMENT


This work was supported by the following projects: BUT project FIT-S-14-2297, National COST LD12036, research project No.MSM 0021630528, project IT4Innovations Centre of Excellence (ED1. 1.00/02.0070), COST Action project "Manufacturable and Dependable Multicore Architectures at Nanoscale", and internal grant "FIT-S-11-1".


## REFERENCES


[1] George, M., Ait Mohamed, O.: Performance analysis of constraint solvers for coverage directed test generation. In: Microelectronics (ICM), 2011 International Conference on, pp. 1–5 (2011). DOI 10.1109/ICM.2011.6177404

[2] Graphics, M.: Verification academy - the most comprehensive resource for verification training (2013). URL www.verificationacademy.com

[3] Jefferson, C., et al.: The Minion Manual, Minion Version 0.8.1 (2009). [online, available at http://minion.sourceforge.net/files/Manual081.pdf; accessed 06-August-2009]

[4] Kotthoff, L.: Constraint Solvers: An Empirical Evaluation of Design Decisions. ArXiv e-prints (2010)

[5] Kumar, V.: Algorithms for constraint satisfaction problems: A survey. AI MAGAZINE **13**(1), 32–44 (1992)

[6] Monfroy, E., Castro, C., Crawford, B.: Using local search for guiding enumeration in constraint solving. In: J. Euzenat, J. Domingue (eds.) Artificial Intelligence: Methodology, Systems, and Applications, *Lecture Notes in Computer Science*, vol. 4183, pp. 56–65. Springer Berlin Heidelberg (2006). DOI 10.1007/11861461_8. URL http://dx.doi.org/10.1007/11861461_8

[7] Shen, H., Wang, P., Chen, Y., Guo, Q., Zhang, H.: Designing an effective constraint solver in coverage directed test generation. In: Embedded Software and Systems, 2009. ICESS '09. International Conference on, pp. 388–395 (2009). DOI 10.1109/ICESS.2009.39

[8] Tasiran, S., Keutzer, K.: Coverage metrics for functional validation of hardware designs. Design Test of Computers, IEEE **18**(4), 36–45 (2001). DOI 10.1109/54.936247