A Functional Verification Methodology for an Improved Coverage of System-on-Chips

Jomu George Mani Paret

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

January 2015

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By: **Jomu George Mani Paret**

Entitled: **A Functional Verification Methodology for an Improved Coverage of System-on-Chips**

and submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Dr. Lyes Kadem (Chair)

_____ Dr. Malek Mouhoub

_____ Dr. Ferhat Khendek

_____ Dr. Benjamin C. M. Fung

_____ Dr. Samar Abdi

_____ Dr. Otmane Ait Mohamed

Approved by _____

Chair of the ECE Department

_____ 2015 _____

Dean of Engineering

ABSTRACT

A Functional Verification Methodology for an Improved Coverage of System-on-Chips

Jomu George Mani Paret

Concordia University, 2015

The increasing popularity of System-on-Chip (SoC) circuits results in many new design challenges. One major challenge is to ensure the functional correctness of such complicated circuits. Functional verification is a verification technique used to verify the functional correctness of SoCs. Coverage Directed Test Generation (CDTG) is an essential part of functional verification, where the objective is to generate input stimulus that maximize the coverage of a design. Coverage helps to determine how well the input stimulus verified the design under verification. CDTG techniques analyze coverage results and adapt the input stimulus generation process to improve the coverage. One of the important component of CDTG based tools is the constraint solver. The time efficiency of the verification process depends on the efficiency of the solver. But the constraint solvers associated with CDTG tools require large amount of memory and time to generate input stimuli for SoCs. The solvers cannot generate solutions which are evenly distributed in search space, in order to attain the required coverage.

The aim of this thesis is to provide a practical framework that enables the generation of evenly distributed input stimuli. A basic feature of the search space (data set) is that it contains k sub populations or clusters. Partitioning the search space into clusters and generating solutions from the partitions can improve the evenness

of the solutions generated by the solver. Hence one of our main contribution is a novel domain partitioning algorithm. The domain partitioning algorithm relies on solution generated by a consistency search algorithm developed for our purpose. The number of partitions (required by the domain partitioning algorithm) is determined by using an algorithm which can find the optimal number of clusters present in a data set. To demonstrate the effectiveness of our approach, we apply our methodology on Constraint Satisfaction Problems (CSPs) and some real life applications.

To My Wife, My Sister, My Mom and Dad.

ACKNOWLEDGEMENTS

First, I am deeply grateful to Dr. Otmane Ait Mohamed for his help, guidance and encouragement throughout my graduate studies. He has taught me all I know, of research in the field of hardware verification. He has taught me how to think about research problems and helped me make significant progress in skills that are essential for a researcher. Many thanks to the members of the thesis committee for the assistance they provided at all levels of the research project. I would like to thank my family: my parents, my wife, my sister, and her family, who have been an endless source of love, affection, support, and motivation for me. Many thanks to my good friends at the hardware verification group for their support, help and motivation. Finally, my greatest regards to the Almighty for bestowing upon me the courage to face the complexities of life and complete this project successfully.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

A&R	Acceptance and Rejection
AC	Arc Consistency
ACA	Ant Crawl Algorithm
AETG	Automatic Efficient Test Generator
AI	Artificial Intelligence
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BN	Bayesian Network
CATS	Constrained Array Test
CDTG	Coverage Directed Test Generation
CFG	Control Flow Graph
CRV	Constraint Random Verification
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DPLL	Davis Putnam Logemann Loveland
EDA	Electronic Design Automation
FP	Floating Point
FSM	Finite State Machine
GA	Genetic Algorithm
GAC	Generalized Arc Consistency
GACCC	GAC on Conjunction of Constraints
GP	Genetic Programming
ILP	Inductive Logic Programming

IPO	In-Parameter-Order
LCS	Learning Classifier Systems
MBTG	Model Based Test Generator
MM	Markov Models
PSL	Property Specification Language
RSSDE	Range Splitting heuristic & Solution Density Estimation
RTL	Register Transfer Level
SA	Simulated Annealing
SoC	System-on-Chip
SVA	SystemVerilog Assertion
TS	Tabu Search
XCS	eXtended Classifier System

Chapter 1

Introduction

1.1 Motivation

> *The number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented.*
>
> Gordon Moore [1965]

Due to the rapid advances in silicon manufacturing technologies, silicon capacity has been steadily doubling every 18 months as predicted by the Moore's Law [1] (Figure 1.1). As a result, a designer is able to implement a complete and complex system on a single chip. The technology is commonly known as System-on-Chip (SoC). As semiconductor technology continually improves, SoC designs are becoming more popular. SoC usually consists 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. The process of finding errors in a design is called verification. Due to the importance of ensuring a design's functional correctness, a great deal of effort has been devoted to design verification.

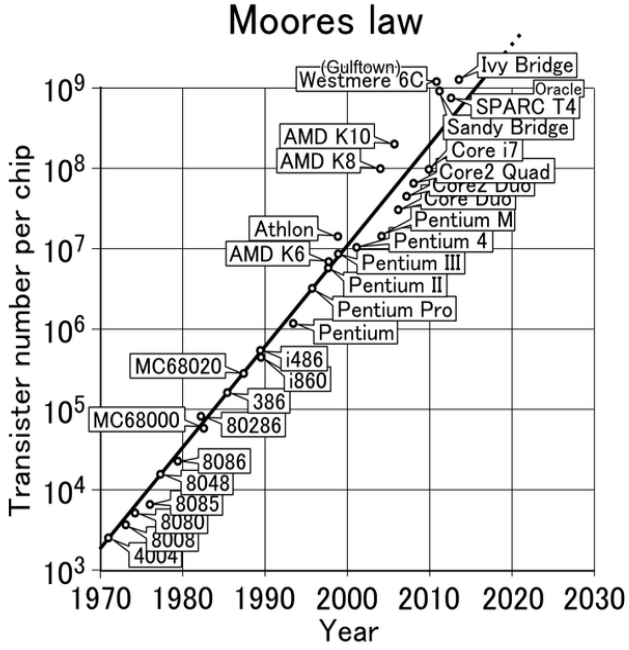

Figure 1.1: Moore's Law

However, as design complexity increases, experience shows that many bugs remain undetected even though considerable resources and time have been devoted to design verification.

As the statistics in industry surveys show, up to 70% of project resources have been devoted to functional verification [2]. Hence, the efficiency of functional verification has a significant impact on the speed with which designs can be put into production. Only 33% of SoC designs are correct on the first pass, and 75% of all design flaws are attributable to, logic or functional bugs due to shortcomings in functional verification [3]. The complications of functional verification stem from the sheer complexity of today's SoCs. Due to the increasing complexity of SoC design and the decreasing time-to-market, functional verification has become a major challenge in the design development cycle. The increase in logic bugs (design errors) is proportional

(sometimes exponential) to the increase in design complexity. The increase in design errors makes verification tasks more difficult.

The main task of functional verification is to compare the specification of a design with its observed behavior and to determine the equivalency between the specification and the actual design, and any differences are reported as bugs. There are several methodologies for tackling functional verification problem and they are divided into simulation methods and formal methods. Formal methods use mathematical expressions and mathematical reasoning to prove the correctness of the design, while in simulation methods, the design is represented functionally and logically by the semantic of a language which can be simulated to observe the behavior of the design. Formal verification focuses on systematic ways to prove or disprove the correctness of the design using mathematical formal methods. Mathematical expressions and symbols are used to express the properties of the design. Formal methods then use mathematical reasoning to prove or disprove the correctness of the properties regardless to the input values [4]. There are three main approaches for formal verification: Model Checking, Equivalence Checking, and Theorem Proving. Model checking and equivalence checking are exhaustive techniques and cannot be used for large design due to state-space explosion problem. This problem is partially solved by introducing Symbolic Model Checking [5]. On the other hand, Theorem Proving can be used to verify larger designs but it is not very practical due to considerable human efforts and the expertise needed [6]. Hence the viable method for the verification of large designs is simulation methods.

Simulation methods can be further divided into several methodologies such as

simulation based verification, assertion based verification, and coverage based verification [7]. In simulation based verification, a dedicated test bench (input stimulus) is built to functionally verify the design by providing meaningful scenarios. On the other hand, assertion based verification is used to catch the place where errors occur, where assertions are written either in a hardware description language or specialized assertion language (e.g., Property Specification Language (PSL) or SystemVerilog Assertion (SVA)). The concept of coverage based verification requires the definition of coverage metrics which are used to assess the progress of the verification cycle and to identify functionalities of the design that have not been verified.

When generating an input stimuli, an empirical evaluation of an existing stimuli is required to direct the generation process and to provide a goal for completion. Coverage is a measure of the completeness of a set of input stimuli, which are applied to the design. The concept of coverage based verification requires the definition of coverage metrics that provide quantitative measures of the verification process. A coverage metric defines a set of goals which must be satisfied during simulation. The most widely used metrics are: code coverage, toggle coverage, path coverage, Finite State Machine (FSM) coverage, and functional coverage. Each metric provides specific aspects about the completeness of the verification process. Even though none of these metrics are sufficient to prove a design is error free, they are helpful in pointing out areas of the design that have not been verified.

In all the above simulation methods, a dedicated input stimulus is generated to verify the design functionality by providing meaningful scenarios. The success of verification methods depend heavily on the quality of the input stimuli in use. There

are three ways to generate the input stimuli:

1. **Directed Stimuli Generation:** In directed stimuli generation, the verification engineer writes input stimuli that are biased to stress specific aspects of the design. But most of the directed stimuli are currently manually written, which is time consuming and error-prone.

2. **Random Stimuli Generation:** Random stimulus generator is used to explore unexercised areas of the design. In this method the input stimuli are generated randomly. In random stimuli generation, unobserved scenarios will be generated and certain scenarios can be easily verified multiple times with different input stimuli. But the verification engineers have no control over the generated input stimuli. Hence certain scenarios which are of interest for the verification engineer may not be generated.

3. **Constraint Random Stimuli Generation:** The number of input stimuli valid for a particular design is limited. All the valid input stimuli are not of interest since the verification engineers are concentrating on certain scenarios. In Constraint Random Verification (CRV) method the conditions for valid input stimuli and conditions for the scenarios are specified. Solving the constraints will give the required input stimulus. The different constraints used in CRV are:
   (a) Constraints based on design specifications given by the design engineer.
   (b) Constraints based on expert knowledge of the verification engineer.
   (c) Constraints based on verification scenarios chosen by the verification engineer.

With the directed approach, the amount of time required to generate new tests is relatively constant, so the verified functionality improves roughly linearly over time.

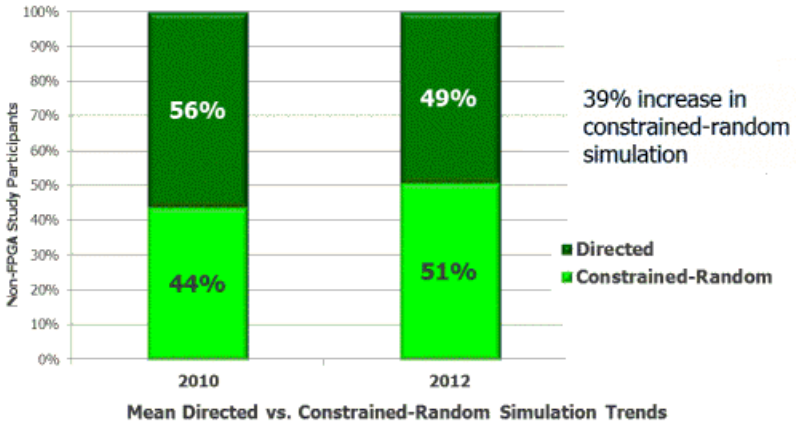

Figure 1.2: CRV vs Directed Stimuli Generation

With a constraint random verification, there is an up-front cost that must be invested before the first test can be run. This initial up-front cost is to build a verification environment in which the relevant portions of the test are parameterized and constrained, such that, future tests can be easily derived. By using randomization to generate the input stimuli required for scenarios that are created, input stimuli which more likely to hit corner cases are generated and thereby find more design bugs. Such tests are also much more likely to hit coverage points, accelerating verification closure.

Hence out of the many stimulus generation methods that have been developed, constraint random stimuli generation is the most commonly used method for the verification of complex designs. Figure 1.2 shows the gaining importance of CRV in present day verification [8]. Coverage tools are used side by side with constraint random test generator in order to assess the progress of the verification plan during the verification cycle. Coverage analysis serve two critical purposes in the verification process. The first is serve as an indicator of when verification is thorough enough to tape out. Coverage provides more than a simple yes/no answer; incremental improvement

in coverage metrics helps to assess verification progress and thoroughness, leading to the point at which the verification engineer has the confidence to tape out the design. Second purpose is to identify holes in the process by pointing to areas of the design that have not yet been sufficiently verified. This allows for the modification of the directives (constraints) for the stimulus generators and the targeting of areas of the design that are not covered [9]. This process of adapting the directives of stimulus generator according to the coverage reports is called Coverage Directed Test Generation (CDTG). It is a time consuming and exhausting process, but it is essential for the completion of the verification cycle.

In order to understand the complexity of CDTG, let us look at one example involving verification of the Floating Point (FP) unit present in microprocessors using a CDTG tool. This is an industrial problem described in detail in [10][11]. FP unit is known to exhibit an exceptionally wide array of corner cases, making its verification a difficult challenge. Input stimulus generation for FP unit verification involves targeting corner cases, which can often be solved only through complex constraint solving. Hence the main task of the constraint solver is to generate a set of input stimuli that comprises a representative sample of the entire search space, taking into account the many corner cases. Consider a FP unit with two input operands. This potentially yields 400 (20^2) cases that must be covered, assuming 20 major FP instruction types (e.g. +/-zero, +/-min denorm,). With four floating point instructions (addition, subtraction, division and multiplication) there is about 1600 cases to be covered. The probability that a CDTG tool will generate a sequence that covers a particular combination is very low [12]. Hence a CDTG tool will take many hours to generate the input stimuli required to attain the required coverage. So the main

motivation of this research is to attain the required coverage in less time.

1.2 Thesis Contributions

The verification of SoC design is arguably the biggest challenge for designers. On the other hand, designers are still constrained in using the traditional time consuming simulation methods for verification. A new verification methodology for SoC design is needed. The new methodology has to be able to reduce the amount of time required for input stimuli generation and to attain required coverage. A basic feature of the search space (data set) is that it contains k number of sub populations or clusters. Partitioning the search space into clusters and generating solutions from the partitions can improve the evenness of the solutions generated by the solver. Hence to achieve the above goals we proposed a methodology based on domain partitioning. The contributions of this dissertation are as follows:

1. **A survey of the related work**. We summarized some of the important existing work on CRV, automated CDTG and state of the art in the area of evenly distributed stimuli generation.

2. **Methodology based on domain partitioning**. We proposed a methodology based on domain partitioning to improve the evenness of the solutions generated by the solver.

3. **Estimation of number of clusters in a data set**. We formulated a method for the estimation of number of clusters in the input domain of a design.

4. **Domain partitioning algorithm**. We developed a fast domain partitioning algorithm which helps to generate more evenly distributed CSP solutions. We

also defined some metrics which helps to determine the evenness of the solutions generated.

5. **Consistency search algorithm**. We presented an optimized consistency search algorithm suitable for our purpose. Since variable ordering scheme used in consistency search has an impact on the search speed, a suitable variable ordering scheme was formulated.

6. **Implementation**. Our frame work called DPCGEN was implemented in C++ and it was successfully applied on CSPs and some real case studies.

1.3 Thesis Organization

This dissertation presents our methodology for generation of evenly distributed input stimuli and the required algorithms. The dissertation is organized as follows:

Chapter 2: Related Work: A survey of the important existing work on CRV, automated CDTG and state of the art in the area of evenly distributed stimuli generation is presented. It is followed by a discussion of domain partitioning techniques used in software testing.

Chapter 3: Preliminaries: This chapter briefly discusses the necessary background on CSPs, consistency search and SystemVerilog constraints.

Chapter 4: Proposed Methodology based on Domain Partitioning: The existing methodology for CDTG is presented in this chapter. It is followed by the proposed methodology for generating evenly distributed stimuli generation for faster coverage.

Chapter 5: Consistency Search Algorithm: The proposed consistency algorithm is presented along with the variable ordering scheme for the algorithm. We

also evaluated the performance of the proposed algorithm.

Chapter 6: Estimation of number of clusters in a data set: The necessary steps to determine the number of clusters in a data set is provided in this chapter.

Chapter 7: Domain Partitioning Algorithm: In this chapter we presented the proposed domain partitioning algorithm.

Chapter 8: Applications and Implementation: We used the proposed methodology in the study of real life applications and proved that the proposed methodology is able to attain higher coverage in less amount of time than the existing CDTG method.

Chapter 9: Conclusions and Future Work: In this final chapter we provided a summary of the thesis with a reflection about the achievements made and several future research directions.

Chapter 2

Related Work

One direction of research in CDTG is to improve the input stimuli generated by the CRV tools. Research is going on to develop effective constraint solver for CRV tools. All high-end hardware manufacturers use CSPs to produce input stimuli. Some manufacturers of less complex designs rely on Electronic Design Automation (EDA) tool vendors (e.g. Cadence, Mentor Graphics and Synopsys) for their stimulus generation needs. Those EDA tools, in turn, are based on internally developed constraint solvers. Others,to solve such as Intel, adapt external of-the-shelf solvers to the stimulus generation problem.

Some manufacturers such as IBM rely on proprietary constraint solvers developed in-house to solve this problem. But development of random test generators are often hindered because of the following difficulties:

1. Complexity: Computer architectures are often complex and has hundreds of instructions, dozens of resources (e.g. memory) and complex functional units (e.g. floating point unit).

2. Changeability: Design verification starts when architecture is still evolving.

Many of the changes are due to the bugs found during previous verification and the tests have to change accordingly.

3. Dependability: The design architecture and the scenarios for verification are tightly related.

2.1 Model Based Techniques

In order to tackle the above issues, IBM developed a model based test generation approach. The motivations to generate a model based tool are:

1. To generate better quality tests: If the quality of tests are high, it will result in smaller number of tests, low simulation cost and less time for verification.

2. To allow verification engineers to add knowledge to the test generator: This will help to improve the quality of the test generated by the tool. This will also allow the reuse of the knowledge to similar designs in future.

3. To reuse a test generator irrespective of the design architecture: Usually the test generator is custom built for an architecture and is very expensive. Separating architectural details helps the use of the same test generator for design architecture.

In this approach a knowledge base is made based on the experience of the verification engineers working on different design architectures. The knowledge base includes the formal description of the design architecture and the testing knowledge. In order to deal with complexity, changeability and dependability, the knowledge base is separated from control. The control includes the architecture independent test generation process.

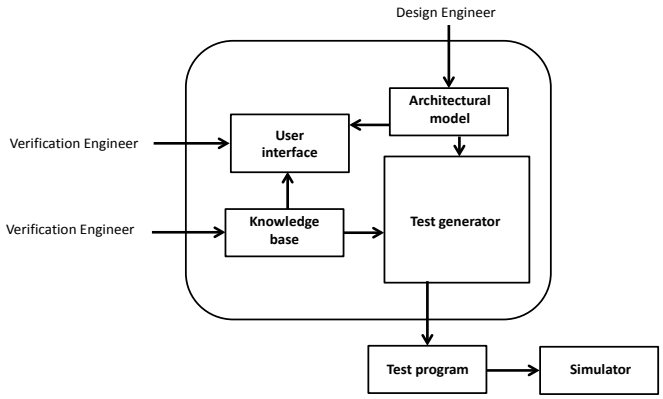

Figure 2.1: Model Based Techniques

A model based test generator (figure 2.1) comprises of the following components:

1. Architectural Model: The architectural model contains the specification of the instructions and the data types used in the designs.

2. Simulator: The simulator simulates the architectural model using the generated test cases.

3. Architecture independent test generator: Great importance is given to the efficiency of the test generator. Constraint solver which is fast, efficient and not based on backtracking techniques is used. Some times modifications are applied to the generator to suit the targeted architecture.

4. Knowledge base: A large number of test programs were analyzed to build the knowledge base. The descriptions of high level verification goals and detailed verification tasks were extracted to generate the knowledge base.

5. User interface: The user interface offers control over the test generation process. The user can define the number of instructions in each test program and initialize the resources used.

The different tools developed by IBM using model based approach are:

Model Based Test Generator : In 1991, IBM developed the first model based constraint random stimulus generator called Model Based Test Generator (MBTG) [13]. In MBTG, the instructions of the architecture were modeled as trees. An instruction tree includes a format and a semantic procedure at the root, operands and sub-operands as internal nodes and length, address and data expressions as leaves. The instructions are generated by traversing this tree. The test generator traverses through the instruction tree and generate the required instructions. The generator accepts the number of instruction from the user through the user interface.

X-Gen : X-Gen [14] provides a framework and a set of building blocks for system level input stimuli generation. It uses a system level model which consists of, component types, their configuration and the interactions between them for stimuli generation. X-Gen accepts a set of user-defined requests known as $'requestfile'$ as input. For a given request file, it generates a set of different input stimuli, each of which realizes the request file. Through request files, users can provide a full or partial description of a required scenario. X-Gen, using its random generator, verification knowledge and the specified scenario will generate the input stimuli for the specified scenario.

Genesys : IBM developed Genesys [9] as a follow-on tool of model based test generator, which can generate infinite number of tests. It is able to minimize the effort required to use the tool with any architecture, allow changes to architecture and can have upgrades which can be easily implemented within the tool. Most importantly, it has the ability to externally and internally improve the knowledge base. The tool consists of mainly three components: a generic architecture independent test generator, the architecture model and the simulator. The model allows incorporation of testing

knowledge along with it. Hence it allows the user to control the test generation by biasing towards specific verification scenarios.

Genesys-Pro : Genesys-Pro [15], IBM's third-generation test generator relies on the same underlying model based approach as Genesys but has three significant advancements. First the test template has the expressiveness of a programming language which allows unlimited control over the test generation. Genesys-Pro has high-level building blocks that can be used to describe the processors. The third major improvement was on the test generator. The test generator translates the test generation process into a CSP problem and uses a generic CSP solver for test generation. This CSP generator became the backbone of later IBM test generators.

FPGen : Mainstream input stimuli generation tools can only provide some scenarios for verifying FP implementations. Because of lack of internal knowledge related to the FP domain, they are inadequate for providing a solution for FP verification. FPGen [16] primarily targets architectures that comply with the IEEE Standard 754, but it can also be used for architectures that are made from the standard FP design. FFGen offers a convenient platform that consists of a language for the definition of the verification requirements, and powerful solving engines that generate random, different input stimuli for different verification scenarios.

Piparazzi : The Piparazzi [17] helps to find bugs that cannot be found by using the architectural level stimuli generators. Piparazzi has mainly two inputs: model of the micro-architecture and the users specification of a required event. The model of the micro-architecture is made by using of a set of in-built building blocks. The building blocks used in the model describe the structure of the micro-architecture and the flow of instructions through it. Each building block is associated with several fixed parameters which determine its nature and basic behavior. For example, a cache

contains parameters for its size, associativity, replacement policy etc. Piparazzi uses the model built (using the building blocks) and the user specification to construct the verification scenario as a CSP problem, and then solves the problem in order to produce the input stimuli.

SoCVer : During the verification of SoCs, it is required to verify the integration of several previously designed cores in a relatively short time. Also, the system's embedded software is not fully written until the late stages in the design development cycle. Moreover it is impossible for a verification engineer to completely comprehend all the dependencies between different cores within the DUV, and therefore, many aspects of the system are ignored or set to static values during the verification process. In SoCVer [18], an abstract model of the SoC which contains a description of the different cores in the system, the operations supported by the system, and the tasks performed by the cores as part of the system's operation are given as input. By incorporating expert knowledge, the tool helps to improve coverage by increasing the probability of hitting corner cases. The SoCVer can generate the software for the DUV's main controller. Hence it does not rely on the existence of embedded software for the verification of the SoC.

In general, the size of design and complexity makes CRV a difficult problem. How ever by having test cases which helps to cover the specified verification scenarios, helps to attain verification closure. In addition, separating the architectural model from the control helped to have a general frame, which can be used for the verification of different architectural design. Table 2.1 draws a brief comparison among the above mentioned tools developed by IBM. Although model based techniques requires a high level of expertise to model verification knowledge and architectures, MBSGs have the following advantages:

Table 2.1: Model Based Techniques

Tool	**Design Level**	**DUV**	**Distinguishing feature from previous version**	**Weakness**
MBSG	Architectural level	Processor	First tool to integrate stimuli generator with design architecture	Performance is about 1/5 of earlier cases.
Xgen	System level	SoC / Server	Has a set of building blocks to describe system level architecture	The building blocks are designed and built for systems and SoCs.
Genesys	Architectural level	Processor	Introduced Testing Knowledge Base	Modeling language has limited expressiveness. It uses a local heuristic search method.
Genesys-Pro	Architectural level	Processor	The language that describes the constraints has the expressiveness of a programming language. It has building blocks specifically suited for describing processors. It has a powerful input stimulus generation engine. It uses customized MAC algorithm for test generation.	It is specifically tuned for verifying processors.
FPgen	Architectural level	FPU	Dedicated tool for FPU	It is designed and built for FPU.
Pipparazzi	Micro-Architectural level	Processor	Has a set of in-built building blocks to describe the structure of micro-architecture	The complexity of the CSP solution puts a limit to the number of instructions per test. It also requires longer test generation time compared to other architectural test generators.
SoCVer	Architectural level	SoC	Can generate the embedded software for the DUV controller	Scheduling problem is solved using CSP solver. There are other more efficient techniques to solve scheduling problem like greedy algorithm, linear programming. ..etc.

1. There is a structured, well defined way to integrate new knowledge about the verified design into the tool.

2. One of the important components is a generic input stimuli generator which can include generic knowledge that applies to many designs. This makes the input stimuli generation of designs which are upgrades of an existing design much easier.

3. Generated input stimuli are higher in quality. As a result full coverage of complex verification plans is possible, and there are very few or no escape bugs.

2.2 Automated Coverage Driven Test Generation

In Coverage Driven Test Generation (CDTG)(figure 2.2)[19], the verification scenarios modeled as constraints, are given to the constraint solver. The constraint solver solves the constraints and generates the input stimuli. The input stimuli and the Design Under Verification (DUV) are then given to the simulator to generate the simulation traces and coverage report. The coverage report shows whether all the required scenarios are covered or not. If the required scenarios are not covered, then the constraints are modified based on the coverage report and are given to the constraint solver. The process is repeated until the required coverage is attained. Present day, the coverage reports generated by the tools are manually analyzed. Test directives required to cover the coverage holes are also generated manually. Incorporating automated coverage analysis along with the generation of test directives will go a long way in tackling the verification problem.

In an effort to increase performance and to decrease the manual effort, automated CDTG has been developed. The research has turned to Artificial Intelligence

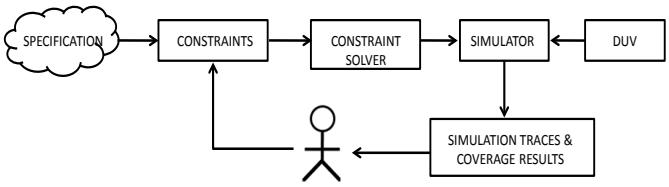

Figure 2.2: Coverage Driven Test Generation Technique

(AI) techniques to automate CDTG. Machine learning algorithms are used to fully automate the process of generating constraints based on the coverage report. Methods used for Automatic Test Pattern Generation (ATPG) (a method used for generating input stimuli for testing) were altered and used to automate CDTG. The recent advances demonstrate that, embedding machine learning techniques into a CDTG framework can effectively automate the stimulus generation process, making it more effective and less error prone.

For automated CDTG only supervised learning techniques are used. The techniques include Bayesian Networks and Data Mining techniques, Markov Models and Inductive Logic Programming (ILP). Although Genetic Algorithms and Genetic Programming fall under the Evolutionary Computation (EC) techniques, their underlying method to achieve their goals are similar to supervised learning techniques. These EC techniques uses Darwinian natural selection theory to find the optimal solution to a problem inside its predefined solution space. Hence they are also used for the automation of CDTG. The different AI based automated CDTG techniques are:

XCS : Learning Classifier Systems (LCS) [20] is an AI technique that provides a set of rule (if "*condition*" then "*action*") that forms the solution to presented problems. The eXtended Classifier System (XCS) is a very popular LCS variant. The first effort in applying XCS for automated CDTG is presented in the work in [21], where the authors first detect the problem and feed it into the system. The next step is to find any classifiers whose condition part is matching the given problem, thus forming the

Match set [M]. An action set [A] is created based on conditions in [M]. According to the effectiveness of the proposed action, a reward value is assigned by the verification environment. These reward values determine the usefulness of each condition toward solving the problem. The system then either chooses a random action or an expected action which will fetch the highest reward. Using the rules/conditions, efficient test programs are generated.

C4.5 : C4.5 is an algorithm used to generate a decision tree. The decision trees generated by C4.5 can be used for classification. It uses hill climbing search to generate constraints from the search space of decision trees. In [22], the authors proposed the use of C4.5 for the extraction of micro architectural data from the simulation traces. The data is then used for the construction of a decision tree. The required input stimuli is generated from the decision tree. In [23] the authors used C4.5 to generate the required constraints for architectural designs. The method is composed of two steps: construction and cutting. In construction, the training set is analyzed in order to obtain required constraints, that can maximize the gain criteria defined by the user. In cutting, the constructed tree is trimmed to increase the generalization capability of the created constraints.

Bayesian Networks : Bayesian Network (BN) is a graphical representation of a model based on probability. BN can be used to find the relationship between the constraints used and coverage obtained. A pre-constructed Directed Acyclic Graph (DAG) is trained to get result. The DAG's edges are trained according to the sample data given. The data set of previously gathered test programs and coverage is used to train a given BN. Once the training is finished, the remaining coverage areas to be targeted and the constraints required to target the uncovered areas is obtained.

The use of Bayesian Networks (BN) for automated CDTG has been initiated and

continued by a research team in IBM [2][24]. They used BN to find the relationship between the test generator constraints and the coverage achieved during simulation. In [25] the same authors propose two adaptation methods to enhance the accuracy of the constraints (test directives) proposed by the BN. Among the two, on-line adaptation method was more effective. The adaptation algorithms resulted in an improvement in the quality of the constraints generated by the BN. In some cases it resulted in a speedup of the coverage process by a factor of 2.

In the above methods, the DAG has to be constructed and it requires initial engineering effort and expertise. In [26], the authors enhanced the methodology by automatically constructing DAG. But the results obtained were not good compared to earlier methods.

Genetic Algorithm : Genetic Algorithm (GA) tries to find solution for a given problem by searching the solution space. Each individual in the search space represents a solution. It uses selection and fitness parameters to guide the research for individuals. The most relevant individuals are selected based on the fitness function. The selected individuals then undergo mutation and cross over to create new individuals. The process is repeated until the required criteria is satisfied.

In [27] authors proposed a self adapting GA which was able to give input stimuli in less time and full coverage for small codes. Since the algorithm is self adapting, no expert knowledge is required to set the fitness parameters. In [28] the authors investigated the effect of the number of evolutionary epochs and population size in test generation. The proposed techniques used GA to obtain the constraints required for a user defined coverage metric called Buffer Utilization.

In [29] the authors use GA along with statement and path coverage in order to generate the required constraints. In addition to the above coverage metrics, the

authors utilize another type of coverage termed error coverage (coverage of gate-level types of faults in RTL or behavioral level models in higher levels). Finally in [30], a GA approach which represents solutions as bit-strings was proposed.

Genetic Programming : Genetic Programming (GP) is an approach that automatically generates input stimulus according to an instruction library by using an evolutionary computation algorithm. Each solution is evolved into a DAG. The nodes represent input instructions with associated parameters. The evolution of new test is based on the changes made on the node branches of the graph. The genetic operators utilized are crossover (2-point crossover) and mutation. In crossover, two nodes in each of the parent's graphs are selected and swapped between them. Then the parent's graph is mutated by addition, deletion or alteration.

The GP approach in [31] proposed a method using only one genetic operator (mutation) but used three special types of mutation. The genetic program were evaluated using a statement coverage model. In [32] and [33] the team presented the same system but the fitness function was altered to include a term that favoured shorter programs. Another change is that they included a crossover operator apart from the standard three mutation operators previously used.

Similarly, a method for evolving test programs by optimizing a multi objective fitness function was proposed in [34]. The main advantage of this approach was the reduced time needed to simulate/evaluate the stimuli evolved.

Markov Model Approach : The use of Markov Models (MM) for automated CDTG was proposed in [35]. Similar to BN, the DUV is represented as a Markov Model where each node represents a transaction sequence of a specific scenario. The methodology adjusts the weights (probabilities) of links between nodes of the MM graph according to the coverage report. At the end, we will get a new MM which

helps to get input stimuli that can achieve higher coverage.

Inductive Logic Programming : Inductive Logic Programming (ILP) is a declarative inductive learning method which requires a training set and some relational background knowledge. ILP aims to discover a single or multiple hypothesis which can cover every element in the training set. In [36] authors proposed the use of an ILP learning algorithm to close the loop between coverage analysis and stimulus generation. Assuming there are enough instances to learn from, at the end of the learning process, the system returns a set of rules, containing at least one rule for each coverage task presented to the system.

Data Mining : Data mining is the technique used to identify patterns or predict the future, based on a set of records. In [25] the authors showed that the data mining algorithms, clustering and instance based learning, can be used for automation of CDTG. In this, the constraints are clustered on their similarity and used to attain maximum coverage. The advantage of DM is that it does not require any domain knowledge.

All the above mentioned techniques use the data that is available from the previous simulations. They automatically analyze the data and try to find the constraints that are require to direct the next round of simulations towards the required coverage. Future works are required to make automated CDTGs more user friendly, to reduce the required engineering effort and to become more consistent in completing coverage closure. Table 2.3 gives a brief comparison among the above mentioned techniques. The table provides the requirements, input and coverage model used by different techniques. An industrial attractive CDTG tool should have the following features:

1. An automated CDTG technique should not be technically demanding for a

Table 2.2: Coverage Driven Test Generation Techniques-1

Ref	DUV	Coverage Type	Requirement	Input	Weakness
[21]	Firepath DSP processor	Cross	Test generation process require distinct XCS rules per each seed of test generator.	Directives/constraints for test generation.	Length of test should be of fixed size. In XCS, fitness is based on the accuracy with which a rule predicts rewards.
[22]	POWER7 processor	Functional		Test template	Small variation in data can lead to different decision trees. Does not work very well on a small training set.
[23]	STREAM-PROC	Functional		Training set	
[37]	NorthStar (PowerPC)	Functional	Expertise for DAG construction	Simulation Data	It is very complex and difficult to make BN and to examine the solutions of BN.
[25]	Storage Control Element of z processor	Functional		Test template	
[26]	Instruction Fetch Unit (IFU) of z10 processor	Functional	Mutual information as a criterion that could be estimated from a sample set	Directives and coverage model	
[27]	Cache Access Arbitration Mechanism	Functional		User defined commands/constraints	GAs are very slow and do not give the optimum solution but the 'fittest' solution. Choosing and implementing fitness function is very difficult.
[28]	PowerPC processor	Functional	Structure of inputs to the architecture and fitness function	Constraints imposed by the Architecture	
[29]	ITC99 benchmark designs	Path (Code)	Fitness function	Designs in System C	
[30]	Godson1 processor	Functional		Constraint specification	
[35]	5-stage DLX pipeline	Bug	Probabilistic information on generating sequences of input	Template files	Data available are sometimes insufficient to estimate reliable probability or transfer rates, especially for rare transitions.

Table 2.3: Coverage Driven Test Generation Techniques-2

[32]	DLX/pII pipelined	Statement	An Instruction Library	RTL level design	Does not give the optimum solution but the 'fittest' solution only or some times give non reliable solution.
[31]	i8051	Statement		An instruction library	
[33]	i8051	Fault	Expertise in construction of DAG	An instruction library	
[34]	i8051	Path-delay fault	Functionally coherent fault group	ISA and design constraints	
[36]	5-stage DLX pipeline	Functional		Background rules	ILP cannot have noise in the training data and can cause over fitting.
[38]	NorthStar	Functional	Simulation parameters and covered tasks per simulation	No domain knowledge is required	Input stimuli is dependent on the training set given as input

machine learning literate.

2. The results provided by the tool must be easily to interpret and should give an insight towards the DUV structure and exposed bugs.

3. It can be used on any DUV regardless of its abstraction level, size and underlying functionality, while supporting the most prominent coverage models.

4. It should be able to analyse the different coverage metric used and should be able to find the coverage holes.

2.3 Sampling Based Techniques

In CRV the constraints are given to the constraint solver to generate the input stimuli. In order to ensure that the majority of the verification effort is spend on the simulation of DUV, it is required that the stimuli generation process should be computationally

inexpensive and consume only a small fraction of the resources.

The distribution of the generated input stimuli in the search space has a direct effect on the time spend to meet the required coverage. A highly skewed distribution can dramatically increase the number of feedback loops that are required to attain the required coverage.

Hence the key requirements of constraint solvers associated with CRV are input stimuli generation speed and even distribution of generated stimuli. But solutions generated by CSPs are not uniformly distributed in the search space. By combining different sampling based techniques with the constraint solver, the speed of solution generation and evenness of the solution distribution can be increased. The sampling based techniques has the following advantages:

1. The input generation process is just constraint solving and is very fast.

2. The sampling techniques treat variable independently, hence it can ensure evenness of the generated input stimuli.

3. It is immune to the complexity of the constraints involved.

Figure 2.3 give the basic framework of a sampling based technique. In sampling techniques, there is a preprocessing stage in which the input variable domains are converted to a tree structure. Various sampling techniques are then applied to find cluster (tree branches) within the search space. Then solutions are generated from these clusters or tree branches using stochastic search techniques or SAT solving techniques.

Range Splitting heuristic and Solution Density Estimation Technique: Range Splitting heuristic and Solution Density Estimation technique (RSSDE) [39] can be used to partition the search space in order to have even distribution of input

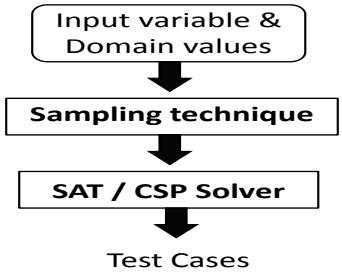

Figure 2.3: Sampling Based Techniques

stimuli. This technique uses statistical analysis for the estimation of solution density. The range-splitting heuristic prunes subspaces which has very low probability to contain a solution. By removing subspace with low solution density, the solution densities in other subspaces are substantially enhanced.

Acceptance and Rejection Technique : The Acceptance and Rejection (A&R) technique [40] ensures uniform or user-specified distribution but the input stimuli generation speed would be slow when constraints cannot be easily solved. Formal solution generators [41] like SAT solvers can solve general constraints very fast but will sacrifice the evenness of distribution. Another approach to increase success ratio for A&R technique, such as RACE [42], is to apply interval propagation to reduce ranges of variables before sampling. Interval propagation based sampling is a technique in which an interval of possible values is maintained for each variable. Each variable is successively assigned a randomly chosen value from its interval, and the intervals of the remaining variables are subsequently refined.

Davis Putnam Logemann Loveland Based Sampling Technique : Davis Putnam Logemann Loveland (DPLL) [43] based sampling utilizes CNF (Conjunctive Normal Form) based DPLL style SAT solvers [44] to generate stimuli from constraints. The advantage is that it has good scalability for a large set of constraints. A random pre-assignment of variables is used to attempt a good distribution for the generated

samples. Starting from this random variable assignment, a DPLL style search is then applied. If there is a conflict, then the solver goes into backtracking mode. In the end it will give a small set of values which are solutions for the SAT problem.

Monte Carlo Markov Chain Technique : In [45], the authors presented a constraint solver that utilizes concepts of Markov chain and Monte Carlo for solving the constraints. In this technique, during each iteration, it proposes a random change to the current assignment and accepts it with a probability that depends on the relative weights of the current and proposed assignments. This technique assumes that, as the number of samples becomes very large, the distribution converges to the desired distribution. The sampler may move across the entire range of a variable in a single step, so it can travel through the sample space faster than other algorithms which uses only local moves.

The dual challenge of generating evenly distributed solution at a faster rate is very demanding. At the same time the search for solutions is NP-hard. The above two problems put very serious limitation to state-of-the-art CRV tools. But only a little research is focused on this problem despite its high relevance in constraint random verification. Some tools generate random values efficiently, but have non even distribution. This results in increased time to achieve coverage. Sampling based techniques tries to address the above issue.

Table 2.4 gives a brief comparison among the above mentioned techniques. The table presents the underlying preprocessing technique used, the solver used and the improvement in performance. The advantage of sampling based techniques is its simplicity and relatively high performance. The techniques requires a large number of iterations for domain clustering or partitioning. Such runtime overheads cannot be neglected when generating solutions from a very large search space. The RSSDE

Table 2.4: Sampling Techniques

Method	**Preprocessing**	**Solver**	**Performance**	**Compared against**	**Disadvantages**
A&R	Range reduction by interval propagation	SAT solver	Speedup of 1.5	BDD based constraint solver	Interval propagation procedure requires a large number of iterations for complete range reduction on complicated constraints.
DPLL	Random preassignment of variables	SAT solver	Speedup of 10	Stimulus generator with self-tuning	For SAT solver the solution distribution can be highly non-uniform.
RSSDE	Range-Splitting	SAT solver	Speedup of 10 and upto 70% uniqueness in the solution generated	Constraint Random Verification	The eliminated sub search space with low solution density may contain solutions which can trigger corner cases.
MCMC	Construction of Markov chain	Monte Carlo Search	Speedup of 1.5 and upto 81% uniqueness in the solution generated	Boolean DPLL	It is hard and inefficient to determine the probabilities required to move from current state to the next state, for non-continuous solution space.

technique reduces the time required for domain partitioning significantly, but sacrifices accuracy since it eliminates domain partitions with low solution density. It would be beneficial to do more research in this area, to improve the accuracy of the technique.

2.4 Graph Based Techniques

As a result of technological advancement, embedded systems continue to face higher performance requirements. Pipelined processors are used to meet these performance requirements. Verification of such programmable processors is a very complex and expensive task. Simulation based verification is the most widely used technique for microprocessor verification. Large amount of time is spend on simulation in traditional

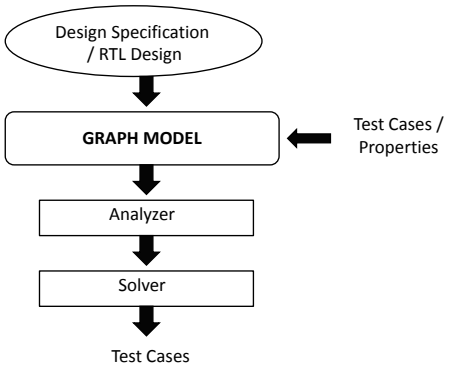

Figure 2.4: Sampling Based Techniques

design flow. Some heuristics and design abstractions are used to generate the require input stimuli. A microprocessor embedded inside a SOC is harder to verify since it is harder to control and observe the behavior of the processor. Also the generated input stimuli may not yield a good coverage.

Graph based techniques was introduced as a promising verification technique for pipelined processors. Figure 2.4 presents the block diagram of the graph based technique. In this technique, first a graph is generated from the specification of the design. Then the graph is analyzed by the test generation algorithm based on the input test cases. The constraints or conditions extracted from the graph by the algorithm is then given to a constraint solver. The solver will generate the required test cases. Hence behavior which are hard to control and observe can be targeted by using graph based techniques.

Control Flow Graph Based Technique : In [46], the authors use Control Flow Graph Based Technique for input stimuli generation. In this technique, the design is simulated for a fixed number of cycles. The branches covered during the simulation is then recorded. The corresponding symbolic trace is extracted from the Control Flow Graph (CFG). They contain executed guard (conditional expression of branch) that evaluates to true or false. This is done with the help of an RTL symbolic

execution engine. The extracted symbolic expressions are placed onto a constraint stack. Then a guard in the symbolic expression is mutated. If the mutated guard has any dependent branches that is not yet covered, then it is passed to an SMT solver. The solver will generate a satisfiable assignment which is a valid input vector. The analysis of RTL is done using the CFG structure of the Hardware Description Language (HDL) source code. This means that the RTL description is analyzed by considering it as a software program, using program analysis techniques.

Pipeline Graph Based Technique : In a Pipeline Graph Based Technique, as mentioned in [47], a pipeline graph model of the processor is generated from the Architecture Description Language (ADL) specification. By using this technique, size of the graph generated is reduced when compared to other techniques. Each node in the graph corresponds to a functional unit or storage component in the processor. An edge in the graph represents instruction or data transfer between the nodes. Then the test program generator will traverse through the graph and generate test cases based on the coverage metric. The test program generation algorithm breaks one processor level property into multiple module level properties and applies them for stimuli generation.

Binary Decision Diagram Technique : In order to increase the evenness of distribution and the speed of solution generation, the weighted Binary Decision Diagram (BDD) technique [48][49] converts the constraints into a single BDD structure. The probability information is assigned on the BDD edges. The idea is to build a BDD from the input constraints and to weight the branches of the vertices in such a way that a simple linear walk procedure from the root to the terminal vertex generates stimuli with a desired distribution. This biased top-down traversal on the diagram, guarantees the evenness of bit level signal distributions and fast production

Table 2.5: Graph Based Techniques

Method	**Graph Type**	**DUV**	**Performance**	**Input format**
CFG	CFG	ITC99 benchmark	90% Functional coverage	Design in RTL format
Pipeline graph	Pipeline graph	DLX processor	Required number of test cases were reduced (about 60%)	Specification in Architecture Description Language
BDD	BDD	12 commercial designs (a crossbar switch, bus interfaces, ALU..)	Speedup of about 2.5 because of reduced size of BDDs	Design in RTL format

of random input stimuli. Hence this technique use smaller size of BDD representation of the constraints. Simulation generation time is reduced due to the smaller BDD size.

In graph based techniques, a graph is generated according to the coverage model used. This graph is then used to generate the constraints required for input stimuli generation. The main advantages of this techniques are:

1. Reduced test generation time.
2. The techniques concentrate only on required path instead of all paths pertaining to each process in the design.
3. The required graph size is significantly reduced by an order of magnitude.
4. It is easier to cover interesting corner cases.

A brief comparison between the different graph based techniques is presented in Table 2.5.

2.5 Domain Partitioning in Software Testing

Quite often the techniques used for verification by hardware industry crosses path with methods used for testing by software industry. So let us have a quick peek at what is happening in software testing. With the expansion of software system size and complexity, there is an ever-increasing demand for innovative testing schemes for software quality and reliability. To test a program, it is necessary to select test data from the program input domain. As it is usually too large to be exhaustively exercised, the usual way for testing is to select a relatively small subset to represent. Therefore, a key issue in software testing is how to select test data from program input domain to detect as many faults as possible with a minimum cost.

There are a large number of test data selection strategies based on partitioning input domain, referred to as partition testing [50][51][52]. In partition testing the input domain is divided into some sub-domains, and one or more representatives from each sub-domain are selected to test the program. Path testing and domain testing are two typical strategies of partition testing.

In the path analysis approach [53][52], partitioning is done based on paths. To understand what is meant by path in this context, consider an example of a very simple program: ***If*** $x < 10$ ***then*** *Event A occurs* ***Else*** *Event B occurs.* Depending on what the value of the variable x is, the program would either go to the path that leads to the execution of $EventA$ or would go to the path that leads to the execution of $EventB$. In the path analysis approach to doing partition testing, the input domain corresponding to a program would be the set of all paths that the program can take.

In domain testing [54][53], the first main task is to partition the input domain into partitions or equivalence classes based on some criterion. All members of one partition or subset of the input domain are expected to result in the execution of the

same path of the program.

A key issue in any partition testing approach is how partitions should be identified and how values should be selected from them. They provide no guidelines for selecting test data. Informal guidelines for creating a partition are discussed in [55][56]. In practice, it is common for the division of the input domain to be into non-disjoint subsets. In order to select values for test cases there are several techniques such as classification trees [50], Simulated Annealing (SA) [57], Automatic Efficient Test Generator (AETG) [58], Genetic Algorithm (GA) [59][60], Ant Crawl Algorithm (ACA) [60], Tabu Search (TS) algorithm [61], In-Parameter-Order (IPO) [62] and Constrained Array Test System (CATS) algorithm [63]. We can see that some of the methods used for selecting data are also used to automate CDTG.

Chapter 3

Preliminaries

3.1 Constraint Satisfaction Problem

Although CSPs were being studied in the seventies, it is only in last two decades that this technique gained huge momentum. Since then it has been successfully applied in various application domains like planning, scheduling, DNA sequencing, resource allocation, query optimization in database ...etc[64]. A constraint represents a relationship that must hold among the participating variables in any solution of the given problem. A CSP consists of a finite set of variables each of which must be assigned a value (or values) from its given finite domain of possible values, and a finite set of constraints that restrict the set of values that these variables may assume simultaneously. A solution to the CSP consists of an assignment of a value (or values) to each of its variables such that constraints of the problem are satisfied. In some problems, the goal is to find all such assignments.

More formally a constraint satisfaction problem is defined as a triple N = $\langle$X,D,C$\rangle$ where

X is a set of n variables X = $\{x_1, \ldots, x_n\}$

D is a finite set of domains for the n variables = {D(x_1),..., D(x_n)}

C is a set of constraints between variables={C_1,...,C_k}

where n and k are non zero positive integers.

A constraint can either be unary, binary, tertiary, or n-ary (affecting n variables) depending on the number of variables it restricts. Constraints affecting more than two variables can be easily converted to an equivalent set of binary constraints using several new auxiliary variables (called binarization of constraints). A CSP containing only unary and binary constraints is called a binary CSP.

CSPs are problems we face in our everyday life. One of the most common example for CSP is N Queens problem. The N Queens problem is a CSP of placing N chess queens on an NxN chessboard so that no two queens threaten each other. Two queens threaten each other if and only if they are on the same row, column or diagonal. We can encode the N Queens problem with N=4, as a CSP as follows:

- Make each of the N rows a variable: X = {var[1], var[2], var[3], var[4]}. The value of each variable will represent the column in which the queen in row_i $(1 \leq i \leq 4)$ is placed.

- Domains: D = {D_1, D_2, D_3, D_4}. Each of these 4 variables can take one of the 4 columns as its value. The domains of the 4 variables are: $D_1 = D_2 = D_3 = D_4$ = {1,2,3,4}.

- Set of constraints: C ={C1, C2}. C is the set of constraints that must be satisfied by the CSP.

  C1 : var[i]$\neq$ var[j] where i,j = 1,2,3,4 and i$\neq$ j.

  C2 : |i-j| $\neq$ |var[i]-var[j]| where || is absolute value.

3.2 Consistency-based Search

To find a solution of a CSP, most constraint satisfaction algorithms use systematic searches by trying combinations of values for variables and checking if they are consistent with the constraints of the CSP. When a variable is assigned a value from its domain, the domains of the connecting variables (via constraints) may get reduced to a currently consistent set. For example, if there is a constraint X $\neq$ Y, and a value for X is chosen during search; that value can be immediately removed from Y's domain, which otherwise may lead to the failure of the inequality constraint between X and Y. This is called domain reduction. Moreover, domain reduction achieved via one constraint can further affect domains of other variables in other constraints owing to their relationships with the variables whose domains are being currently reduced. This is called constraint propagation. Arc Consistency (AC) [64] algorithms utilize constraint propagation and domain reduction to ensure that all binary constraints (binary constraints are called arcs of search tree) are satisfiable with each of the values in the current domains of variables at both ends of the arc (variables in the binary constraint).

The AC can also be taken to higher levels of consistency. Such algorithms are called K consistency algorithms. They ensure that for a consistent assignment of any $K-1$ variables, any K^{th} variable can be assigned at least one value from its domain that is consistent with the constraints of the CSP [64]. Although higher levels of consistency provide stronger constraint propagation, executing them at every node of the search tree requires significant runtime. In general, arc-consistency algorithms represent the best trade-off between the run-time efficiency and constraint propagation achieved.

Several AC algorithms have been developed to increase the efficiency of constraint satisfaction. In AC, for each constraint (C_i) in the CSP, for each variable (v_i) in the constraint C_i, for each domain value of the variable v_i, the algorithm will try to find a list of variable values, which satisfies the constraint.

The arc consistency algorithms are divided into two categories: coarse-grained algorithms and fine-grained algorithms. Coarse grained algorithms are algorithms in which removal of a value from the domain of a variable X will be propagated to only other variables which are related to the variable X. The first consistency algorithms AC-1 [65] and AC-3 [65] belong to this category. These two consistency algorithms are succeeded by AC2000 [66], AC2001-OP [67], AC3-OP [68] and AC3d [69]. Fine grained consistency algorithms are algorithms in which the removal of a value from the domain of a variable will be propagated to other variables in the problem. Fine grained algorithm is faster than coarse grained algorithms. Algorithms AC-4 [70], AC4-OP [71], AC-5 [72] and AC-6 [73] belong to this category. So in both cases, the removal of a value is propagated to the other variables. The difference between both is that the former is based on arc revision while the latter is based on maintaining supports. AC-7 [74] is an algorithm developed based on AC-6. It uses the knowledge about the constraint properties to reduce the cost of consistency check. Overall AC-3 is better than all the other algorithms and the most used one.

Consistency techniques [75][74] 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_i) in the constraint C_i, for each domain value of the variable v_i, the algorithm will try to find a tuple (A tuple is an ordered list which contains values for all the variables in the constraint) which satisfies the constraint. If there is a tuple which satisfies the constraint, then the tuple, constraint variable and the domain value

are stored in a list. The algorithm repeats the process for all the domain value of the variable. At the end of the consistency-based search the domain values which are inconsistent will be removed from the domain of the variable.

To illustrate the idea discussed above, let us consider the following CSP network N: set of variables $X = \{a, b, c, d, e, f, g, h\}$, domain of the variables $D\{a, b, c, d, e, f, g, h\} = \{1, 2, 3, 4, 5\}$ and the constraints $C1 : a + b + c = 6$, $C2 : b + d + e = 8$ and $C3 : e + f + g + h = 10$. After consistency-based search, the tuples stored in the list are as shown in Table 7.1. Constraint 1 shows the tuples which satisfies the constraint $C1$. Similarly constraint 2 and constraint 3 presents the tuples which satisfies the constraint $C2$ and $C3$ respectively.

For variables a, b and c, the domain value 5 is inconsistent. Hence after consistency-based search it is removed from the variable domain. The domain of variables after consistency-based search is as follows: $D\{a, b, c\} = \{1, 2, 3, 4\}$ and $D\{d, e, f, g, h\} = \{1, 2, 3, 4, 5\}$.

3.3 SystemVerilog Constraints

Coding for functional verification becomes more and more crucial as the complexity of the hardware to be verified grows. While verification complexity grows exponentially, it is believed that SystemVerilog serves the coding needs reasonably well. The language's features and expressive capabilities make it usable for functional verification. SystemVerilog hold out the promise of a single unified language to span almost the entire SoC design flow, from module level design and gate level simulation, all the way up to system level verification.

SystemVerilog provides a complete verification environment, employing Directed and Constraint Random Generation, Assertion Based Verification and Coverage Driven

Table 3.1: List of Tuples after Consistency-based Search

Constraint	a	b	c	d	e	f	g	h
1	1	1	4					
	2	1	3					
	3	1	2					
	4	1	1					
	1	2	3					
	1	3	2					
	1	4	1					
2		1		2	5			
		2		1	5			
		3		1	4			
		4		1	3			
		1		3	4			
		1		4	3			
		1		5	2			
		2		5	1			
3					1	1	3	5
					2	1	2	5
					3	1	1	5
					4	1	1	4
					5	1	1	3
					1	2	2	5
					1	3	1	5
					1	4	1	4
					1	5	1	3
					1	1	4	4
					1	1	5	3
					1	1	5	1
					1	2	5	2

Verification. SystemVerilog has been adopted by hundreds of semiconductor design companies and is supported by more than 75 EDA, IP, and training solutions providers worldwide [76].

SystemVerilog allows object-oriented programming for random stimulus generation, subjected to specified constraints. The randomization can be with uniform distribution, weighted distribution, weighted range, weighted case.

SystemVerilog allows two kinds of constraints: domain(membership) constraints and model constraints. The domain constraints are used to specify the domain values of the random variables. Model constraints are used to model the required verification scenarios. Modeling constraint are composed of *foreach* constraints (for constraining elements of array), inline constraints, conditional constraints and implication constraints.

The following shows the SystemVerilog constraint model from the N Queen's problem with $N = 4$:

```
rand int var[4];
constraint svc1{foreach (var[i]) var[i] inside {1,2,3,4};}
constraint svc2{foreach (var[i])foreach (var[j]) j>i->var[i]!=var[j];}
constraint svc3{foreach (var[i])foreach (var[j]) j>i->var[i]-var[j]!=(i-j);}
constraint svc4{foreach (var[i])foreach (var[j]) j>i->var[i]-var[j]!=-(i-j);}
```

The first constraint *svc*1 is the domain constraint. It sets the domain values of the variable to be 1,2,3 and 4. The next three constraints model the two conditions (C1 and C2) that must be satisfied for the N Queen's problem.

Chapter 4

Proposed Methodology

A main drawback in simulation based verification is that an assurance of correctness of the design requires exhaustive simulation, which makes it possible only for small designs. CDTG gives effective method to achieve coverage goals faster and most importantly it helps in finding corner case problem. Because CDTG can automatically generate a large number of test cases with constraints specified by the verification engineers, it can hit corner cases that neither the design nor verification engineers would have ever anticipated. A traditional flow of CDTG is shown in Figure 4.1.

4.1 Existing CDTG Methodology

High quality requirements are one of the most important prerequisite for a successful system development. The requirements specified by a customer or a user of the system need to be fulfilled, in order to ensure the acceptance of the developed product. Several techniques can be used to specify the requirements in the system requirement document. The system requirements document describes the requirements from a users point of view. The user requirements are usually specified by using natural

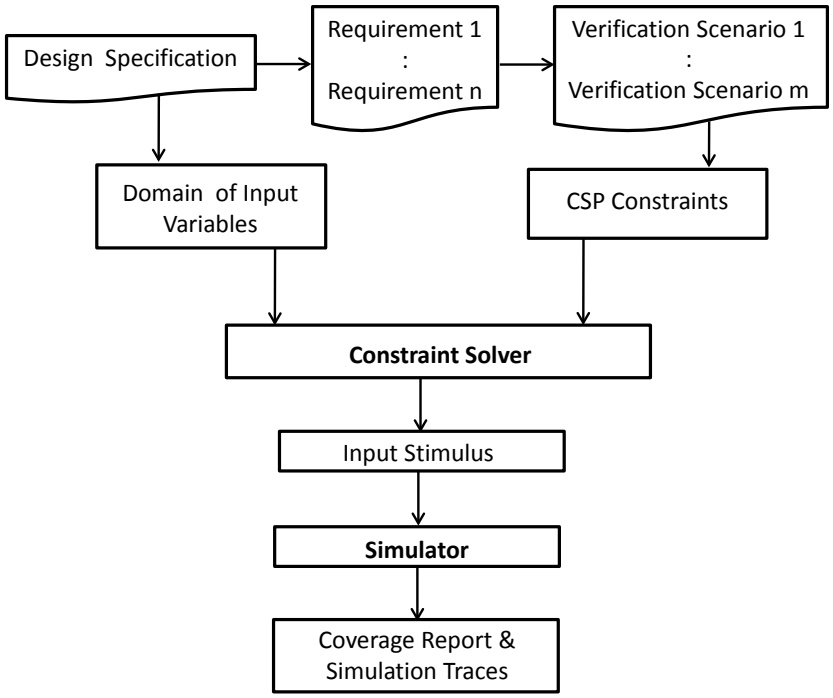

Figure 4.1: Existing CDTG Methodology

language, formal specification languages, data flow diagrams and use cases.

During the past ten years, the use case has been establishing itself for the specification of requirements and high-level designs for various types designs. Since use cases are often used at a very abstract level, close to user requirements, tests derived from use case models have much potential for verifying implementations at the system level, or for verifying more detailed design models. Use cases focus on the description of functional requirements.

Then the use cases are checked for any ambiguities, inconsistencies and omissions. This is done in order to avoid changes at later stages (if the use case is changed then verification scenarios have to be changed). Then from the use case diagrams all possible paths (requirements) are generated. The paths represent all possible user actions and system reactions (the different system requirements). Priorities are set to

paths and verification scenarios are generated based on priorities. The priorities are set based on

- Frequency of use
- Errors found in past in similar situation
- Complexity of path

Each of the verification scenario is then modeled into a CSP problem. The constraints for the CSP and the domain of input variables are given to the constraint solver. The constraint solver generates the input stimuli for the DUV. Cover points are defined on the generated input stimuli. The simulation trace and the coverage results obtained from the simulator are analyzed. Targets that have been missed are found out. Using the simulation traces obtained so far, the constraint generator generates constraints that will help to cover the missed targets. This process is iterated until desired coverage is achieved.

4.2 Proposed Methodology

The goal of this research is to provide an efficient functional input stimuli generation methodology for generating evenly distributed input stimuli, thereby reducing overall verification efforts. Since generating and simulating all possible input sequences is not possible, we need a method to generate effective input stimuli to achieve high confidence of the design correctness. In addition, stimuli generation techniques should be able to generate input stimuli in short time and with less usage of resources (memory). Therefore a preprocessing stage is added to the existing CDTG technique. Figure 4.2 shows the overall flow of the proposed coverage driven input stimuli generation methodology.

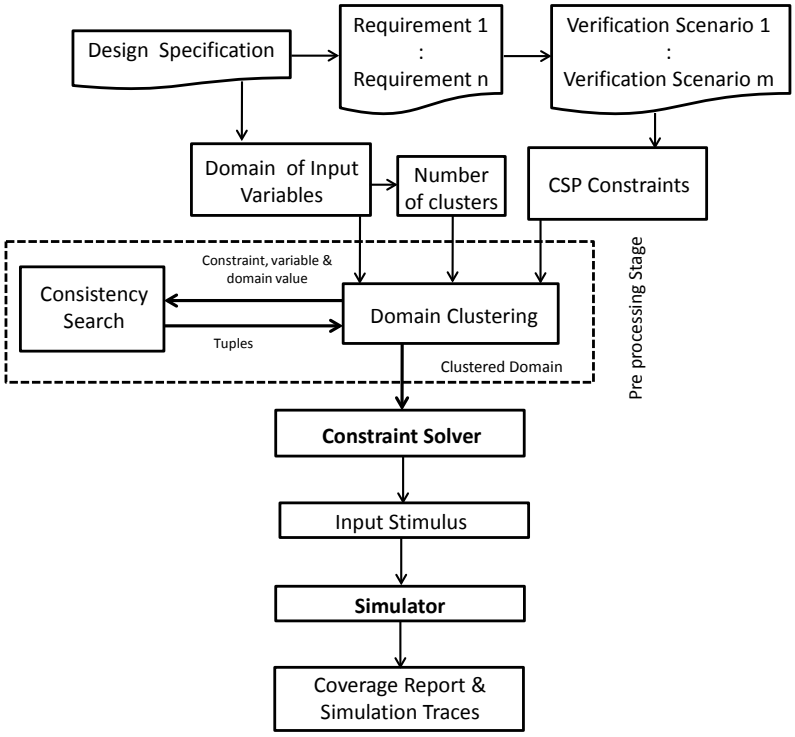

Figure 4.2: Proposed Framework

The design specification gives the different requirements that must be satisfied by the design. The different requirements are converted to verification scenarios. Each of the verification scenario is modeled into a CSP problem. In a CSP, the solutions are clustered together in the search space [77]. Hence partitioning the search space into clusters and generating solutions from the partitions can improve the evenness of the solutions generated by the solver. So the CSP and the domain of the input variables are given to the domain clustering block. The domain of the input variables is used to determine the required number of partitions. Consistency-based search is done on the domain of input variables. It then generate partition tuples (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 to generate tuples. Then the partition tuples are used

to partition/cluster the variable domain. The partitioned 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.

Chapter 5

Consistency Algorithm

5.1 Introduction

As mentioned in preliminaries, consistency search techniques are used to generate solutions for CSPs. GAC-scheme [75] is a consistency algorithm developed for n-arity constraints (n variables are there in the constraint). It is the extension of AC-7 for n-arity constraints. Conjunctive Consistency [75] enforces GAC-scheme on conjunctions of constraints. We chose GAC-scheme on Conjunction of Constraints (GACCC) for our purpose because:

1. We need to eliminate as much invalid domain values as possible. This can be done by performing consistency-based search on conjunction of constraints.

2. GAC-scheme does not require any specific data structure. Other consistency based algorithm use specific data structure to keep track of consistency during their search.

3. The constraints used in CDTG can have more than two variables and GAC-scheme can handle constraint of n-arity.

4. The constraints used in CDTG are not of a fixed type and GAC-scheme can be used with any type of constraints. Some other search algorithms can only deal with a specific class of constraints. For example AC algorithms can deal with binary and unary constraints only.

5.2 Notations

Tuple: A tuple τ on an ordered set of variables is an ordered list which contains values for all the variables. X(τ) represents the set of variables in the tuple τ.

Valid Tuple: The value of variable x in a tuple τ is denoted by τ[x]. A tuple τ on C_i is valid iff $\forall$x $\in$X(C_i), τ[x] $\in$D(x) and τ satisfies the constraint C_i.

Support: If a $\in$D(x_i) and τ is a valid tuple on C_j, then τ is called a support for (x_i, a) on C_j.

Arc Consistency: A value a $\in$D(x_j) is consistent with C_i iff x_j $\in$X(C_i) and $\exists\tau$ such that τ is a support for (x_j, a) on C_i. C_i is arc consistent iff $\forall x_j$ $\in$X(C_i),D(x_j)$\neq$ $\emptyset$ and $\forall$a $\in$D(x_j), a is consistent with C_i.

Generalized Arc Consistency of a network: A CSP is generalized arc consistent iff $\forall C_i \in C$ is arc consistent.

Conjunctive Consistency: If X(S_j)=X(C_1)$\cup\ldots\cup$X(C_k) where X(C_i)= set of variables in C_i, then S_j is conjunctively consistent iff $\forall$a $\in$D(x_k), x_k $\in$X(S_j) and there exists a tuple τ such that a=$\tau[x_k]$ and τ is a support $\forall x_k$.

Conjunctive Consistency of a network: Let P=$\langle$ X, D, S$\rangle$ be a constraint network. P is conjunctive consistent network iff $\forall S_j \in S$ is conjunctive consistent.

5.3 GACCC

In GACCC [75], a variable in, a conjunction of constraint is selected and the selected variable will be assigned a value from its domain. The algorithm will generate tuples in lexicographical order (the selected variable value will not change) and check whether the tuple satisfies the constraint. The algorithm continues to generate tuple until all possible tuples are generated or a tuple which satisfies the constraint is generated. If there is no tuple which satisfies the constraint for the selected variable value, then that variable value is inconsistent and removed from the variable domain. The process will be repeated for all the domain values of the selected variable, then for all the variables in the constraint and for all the constraints in the constraint network.

To illustrate the idea discussed above, let us consider the following CSP: set of variables $X = \{m, n, o, p, q\}$, domain of the variables D(m)={1, 2}, D(n)={2, 3}, D(o)={1, 2}, D(p)={1, 3}, D(q)={2, 3} and the constraints $C1 : m+n+o+p = 7$ and $C2 : m + o + q = 9$. The consistency-based search (for conjunction of constraints) for $m = 1$ has to go through 16 tuples (because each of the remaining variables (n, o, p, q) has two variables in the domain) to find out that value is not consistent (which is the worst case).

5.4 Intuitive Idea of GACCC-op

In consistency check, if any one constraint is not satisfied, then the tuple generated is inconsistent with the conjunction set. We can reduce the number of tuples generated during consistency-based search by using this property. Initially for a given variable, we consider the constraint with lowest number of variables and contain the specified variable. We generate tuples for the above constraint and search for consistency. If

the tuple generated for the smallest constraint is not consistent then all the tuples generated for the conjunction of constraints are also not consistent. If the tuple generated for the smallest constraint is consistent, then only we need to generate the tuples for the conjunction of constraints (tuple generated for conjunction of constraints should contain the tuple which is consistent with the smallest constraint). Since the number of variables in the smallest constraint is less when compared to tuple for conjunction of constraints, consistency can be checked in less number of iterations.

In the above CSP, $C2$ is the smallest constraint in the set, which has 3 variables and the variable m. Consistency check is first performed on this constraint. In 4 (because each of the remaining variables (o, q) has two variables in the domain) iterations we can find that $m = 1$ is inconsistent with the constraint $C2$. Hence $m = 1$ is inconsistent for the conjunction of constraints. The tuples for a variable in conjunction of constraints is generated only if the smallest constraint containing the variable is satisfied by the tuple. Consider another set of constraints $C3 : m + n + o + p = 8$ and $C4 : m + o + q = 6$. By GACCC we have to generate 8 tuples to find a consistent tuple (which is the worst case). By using the new algorithm we need only 5 (4 iterations for $C3$ and 1 for conjunction of $C3$ and $C4$) iterations to find the tuple which satisfies the constraints. So by using the proposed algorithm, consistency check can be completed in less number of iteration when compared to GACCC.

So the difference between GACCC and GACCC-op are as follows:

1. In GACCC the support list is made by using some existing variable order scheme. The known heuristic is to use the most constrained variable in GAC. In GACCC-op we propose a new variable ordering scheme in which the consistency-based search starts with the variable, which is present in the constraint with the lowest arity and has the largest number of domain values.

2. In GACCC during consistency-based search of a domain value of a variable, the tuples generated will contain all the variable in the conjunction set. In GACCC-op the consistency-based search for a variable x will begin with tuples which contain only variables from the smallest constraint(C_s)(C_s should contain the variable x). If there is a tuple which satisfies the constraint C_s, only then GACCC-op generate tuples with the entire variable in the conjunction set.

5.5 GACCC-op

Let us start the discussion of the proposed GACCC-op algorithm with the main program (Algorithm 5.1). First the data structures (lastSc, supportlist, deletionlist and Sclast) must be created and initialized. Sclast, supportlist, deletionlist and lastSc are initialized in such a way that:

1. Sclast contains the last tuple returned by the function **SeekValidSupportSet** as a support for variable value.
2. supportlist contains all tuples that are support for variable value.
3. deletionlist contains all variable values that are inconsistent.
4. lastSc is the last tuple returned by the function **SeekValidSupport** as a support for variable value.

Conjunct the constraints based on the heuristics explained in section 5.6. Then for each set of conjuncted constraints, for each variable present in the constraints, all the domain values of the variable are put in supportlist. The domain values of the variables in a conjunction set are added to supportlist using the following heuristics:

1. Find the lowest arity constraint(C_l) in the conjunction set.

Algorithm 5.1: GACCC-op Algorithm

```
1: Conjunct the constraints based on the conjunction heuristics
2: for each constraint set (S) do
3:    for each variable in set (y) do
4:       for each domain value of variable (b) do
5:          Add to support stream(S,y,b)
6:       end for
7:    end for
8: end for
9: while support stream ≠ nil do
10:   σ = SeekInferableSupport(S,y,b)
11:   if σ = nil then
12:      c = smallest constraint containing variable y
13:      while found soln || checked all tuples do
14:         σ* = lastSc(C,y,b)
15:         if σ* = nil then
16:            LOOP2: σ* = SeekValidSupport (C,y,b,σ*)
17:            if σ* = nil then
18:               DeletionStream (y,b)
19:            else
20:               if variables in all the constraints are same then
21:                  Add to Sclast(S,y,b)
22:               else
23:                  Add to lastSc(C,y,b)
24:                  go to LOOP1
25:               end if
26:            end if
27:         else
28:            if Sclast(S,y,b)≠ nil then
29:               σ** = Sclast(S,y,b)
30:               go to LOOP1
31:            else
32:               σ** = nil
33:            end if
34:         end if
35:         LOOP1: λ* = SeekValidSupportSet(S,y,b,σ**)
36:         if λ*≠ nil then
37:            Add to Sclast(S,y,b)
38:         else
39:            go to LOOP2
40:         end if
41:      end while
42:   end if
43: end while
```

2. Find a variable (x_l) where the variable and C_l is not added to the list, the variable is in C_l and has the highest number of domain values.

3. Add all the domain values of the selected variable (x_l), variable and the constraint to the list.

4. Repeat step 2 until all the variables in the constraint C_l are considered.

5. If there is any variable or constraint set to be added to the list from the conjunction set, then find the next highest arity constraint and repeat steps 2-4.

This supportlist is used to find the support (support is a tuple which satisfies the constraint) for each variable value in the constraint set. For each value in supportlist the algorithm will try to find a valid support by using the function **SeekInferableSupport**. Function **SeekInferableSupport** checks whether an already checked tuple is a support for (y,b). If there is no valid support to be inferred then we will search for a valid support.

Algorithm 5.2: SeekInferableSupport

```
1: SeekInferableSupport (in S:constraint; in y:variable; in b:value):tuple
2: while support stream ≠ nil do
3:    if Sclast(var(S,y),τ[y]) = b then
4:       zigma = Sclast(S,y,b)
5:    else
6:       zigma = nil
7:    end if
8:    return zigma
9: end while
```

For every value 'b', for a variable 'y' in X(C), lastSc(C,y,b) is the last tuple returned by **SeekValidSupport** as a support for (y,b) if **SeekValidSupport**(C,y,b) has already been called or empty otherwise. The above two functions help to avoid

checking several times whether the same tuple is a support for the constraint or not. If the search is new we look for support from the first valid tuple.

If no valid tuple is found then the variable value is not consistent with the constraint. Hence it is not consistent with constraint set. This variable value will be deleted from the domain of the variable by the function **DeletionStream**(y,b).

Algorithm 5.3: SeekValidSupport

```
1: SeekValidSupport (in C:constraint; in y:variable; in b:value; in τ:tuple):tuple
2: if τ≠ nil then
3:    zigma = NextTuple(C,y,b,τ)
4: else
5:    zigma = FirstTuple(C,y,b)
6: end if
7: zigma1 = SeekCandidateTuple(C,y,b,τ)
8: solution found = false
9: while (zigma1 ≠ nil) and (not solution found) do
10:   if zigma1 satisfies constraint C then
11:      solution found = true
12:   else
13:      zigma1= NextTuple(C,y,b,zigma1)
14:      zigma1 = SeekCandidateTuple(C,y,b,zigma1)
15:   end if
16:   return zigma1
17: end while
```

If a tuple is returned by lastSc(C,y,b), we will check for Sclast(S,y,b). Sclast(S,y,b) is the last tuple returned by **SeekValidSupportSet** as a support for (S,y,b) if **SeekValidSupportSet** has already been called or empty otherwise. If a tuple is returned we start the search for support for conjunction constraint set from that tuple, else we will start search from the first valid tuple for the conjunction set, with variables in constraint C has the values of the tuple from lastSc(C,y,b). If the **SeekValidSupportSet** returns empty then we will call function **SeekValidSupport** and repeat the process until a valid tuple for the for conjunction constraint set is found or the lastSc(C,y,b) returns empty. If the lastSc(C,y,b) returns empty then the variable value

is deleted the function **DeletionStream**(y,b). The above processes will be repeated until both the deletionlist and supportlist are empty.

The function **SeekInferableSupport**(Algorithm 6.1) ensures that the algorithm will never look for a support for a value when a tuple supporting this value has already been checked. The idea is to exploit the property: "If (y,b) belongs to a tuple supporting another value, then this tuple also supports (y,b)".

Algorithm 5.4: SeekCandidateTuple

```
1: SeekCandidateTuple (in C:constraint; in y:variable; in b:value; in τ:tuple):tuple
2: k = 1
3: while (τ ≠ nil) and (k≤X(C)) do
4:   if lastc(var(C,k),τ[k])≠ nil then
5:     λ = lastSc(var(C,k),τ[k])
6:     split = 1
7:     while τ[split] = λ[split] do
8:       split = split+1
9:     end while
10:    if τ[split] < λ[split] then
11:      if split < k then
12:        (τ,k')= NextTuple( C,y,b,λ)
13:        k = k'+1
14:      else
15:        (τ,k')= NextTuple( C,y,b,λ)
16:        k = min(k'-1, k)
17:      end if
18:    end if
19:  end if
20:  k = k+1
21: end while
22: return τ
```

After the function **SeekInferableSupport** fails to find any previously checked tuple as a support for (y,b) on the constraint C, the function **SeekValidSupport** (Algorithm 5.3) is called to find a new support for (y,b). But the function has to avoid checking tuples which are already checked. This is taken care by using the function **SeekCandidateTuple**. The function **NextTuple** will generate new tuples

in a lexicographical order which can be a valid support for the constraint variable value.

Algorithm 5.5: SeekValidSupportSet

```
1: SeekCandidateTuple (in S:constraint set; in y:variable; in b:value; in
   τ:tuple):tuple
2: if τ≠ nil then
3:    zigma = NextTuple(S,y,b,τ,θ)
4: else
5:    zigma = FirstTuple(S,y,b)
6: end if
7: zigma1 = SeekCandidateSet(S,y,b,τ,θ)
8: solution found = false
9: while (zigma1 ≠ nil) and (not solution found) do
10:   if zigma1 satisfies constraint set S then
11:      solution found = true
12:   else
13:      zigma1= NextTuple(S,y,b,zigma1,θ)
14:      zigma1 = SeekCandidateSet(D,y,b,zigma1,θ)
15:   end if
16:   return zigma1
17: end while
```

Function **SeekCandidateTuple**(C,y,b,τ) (Algorithm 5.4) returns the smallest candidate greater than or equal to τ. For each index from 1 to $|X(C)|$ **SeekCandidateTuple** verifies whether τ is greater than lastSc (λ). If τ is smaller than λ, the search moves forward to the smallest valid tuple following τ, else to the valid tuple following λ. When the search moves to the next valid tuple greater than τ or λ, some values before the index may have changed. In those cases we repeat the previous process to make sure that we are not generating a previously checked tuple.

The function **SeekValidSupportSet** (Algorithm 5.5) is called to find a new support for (y,b) on the conjunction of constraints. But the function has to avoid checking tuples which are already checked. This is taken care by using the function **SeekCandidateSet**. This function is similar to the function **SeekCandidateTuple**.

The function **SeekCandidateSet** returns the smallest tuple which is a support of the conjunction of constraints.

Algorithm 5.6: DeletionStream

```
1: SeekCandidateTuple (in y:variable; in b:value)
2: if Sclast(var(C,y),τ[y])= b then
3:    Add to supportlist (S,(var(C,x)),a) where x≠ y and τ[x]=a
4:    delete λ from Sclast
5: end if
```

If there is no support for a variable value, then that variable value is deleted from the variable domain by the function **DeletionStream** (Algorithm 5.6). The function also checks whether any tuple in Sclast contains the variable value. If there is such a tuple, then all the variable values in the tuple are added to supportlist to find new support.

5.6 Heuristic for Generating Conjunction Set

The CSPs associated with the verification scenarios have large number of constraints, large domain for each input variable and many of the constraints have the same variables. The pruning capability by consistency-based search can be increased, by combining/conjuncting a large number of constraints together. If a large number of constraints are conjuncted, the variables in the tuple increases and the number of tuples that has to be generated also increases. So there should be a limit to the number of constraints conjuncted together. Similarly the number of variables in the tuple has to be regulated to prevent the tuple from becoming very large. For conjunction of constraints to be effective in reducing the domain values, the constraints in the conjunction set should have a certain number of variables in common. The number of constraints (k), number of variables in the conjunction set (j) and the number of

variables common to all the constraints in the conjunction set (i) depends on the CSP and the machine capacity. So there should be a heuristic based on the parameters i, j and k to determine which constraints can be combined together to make the conjunction set.

The heuristic for grouping constraints into conjunctive sets is as follows:

1. Initially there will be 'n' conjunctive sets($\boldsymbol{S}$), each containing a single constraint (where n is the total number of constraints in the CSP).

2. If there exists two conjunctive sets S1, S2 such that variables in S1 is equal to variables in S2, then remove S1 and S2 and add a new set which is conjunction of all the constraints in S1 and S2.

3. If there exist two conjunctive sets S1, S2 such that (a) S1, S2 share at least i variables (b) the number of variables in S1 $\cup$ S2 is less than j (c) the total number of constraints in S1 and S2 is less than k then remove S1 and S2 and add a new set which is conjunction of all the constraints in S1 and S2.

4. Repeat 2 and 3 until no more such pairs exist.

Table 5.1: Conjunction of Constraints

Constraints in CSP	After step1	After step2	After step3 (i=1,j=5,k=4)
$C1 : a * b > 20$	$S1 : a * b > 20$	$S5 : S1 \bigwedge S2 :$	$S6 : S5 \bigwedge S3 :$
$C2 : a > b$	$S2 : a > b$	$a * b > 20 \bigwedge a > b$	$a * b > 20 \bigwedge a > b \bigwedge a + c = 25$
$C3 : a + c = 25$	$S3 : a + c = 25$	$S3 : a + c = 25$	
$C4 : c + d = 19$	$S4 : c + d = 19$	$S4 : c + d = 19$	$S4 : c + d = 19$

The Table 5.1 shows how constraints can be conjuncted using the above heuristic. During step 3 the constraints $S5$ and $S3$ are conjuncted to form constraint $S6$. The constraint $S4$ cannot be conjuncted with $S6$ because the total number of constraints in the conjunction set should be less than 4 (since k=4).

5.7 Correctness of the GACCC-op Algorithm

To show the correctness of the algorithm it is necessary to prove that every arc inconsistent value is removed (completeness) and that no consistent value is removed by the algorithm (soundness) when the algorithm terminates. Moreover, we need to prove that the algorithm terminates.

Lemma 5.1. *Algorithm will terminate.*

Proof. The algorithm consists of a for loop and two while loops. The generation of elements for the list called *support stream(S,y,b)* uses a for loop. The number of domain values, variable and constraints are finite. Hence the elements generated for the list is finite and the for loop will terminate. The pruning process for the domain values uses a while loop. During each cycle, one element is removed from the list. The elements are added to this list only when a value is removed from some domain. Thus, it is possible to add only a finite number of elements to the list (some elements can be added repeatedly). Hence the while loop will terminate. The algorithm uses a while loop to find support for a variable value in a constraint. The algorithm generates tuples in lexicographic order starting for the smallest one. Since the number of possible tuples for a constraint is finite, the while loop will terminate when it finds a valid support tuple or when all the tuples are generated. □

Lemma 5.2. *SeekCandidateTuple will not miss any valid tuple during the generation of next tuple.*

Proof. Consider that there is a candidate tuple σ' between σ and the tuple returned by the function NextTuple. This implies that $\sigma'[1...k] = \sigma[1...k]$ else σ' will the tuple returned by NextTuple. Hence σ' should be smaller than λ (lines 10-11). If σ' is

smaller than λ then that tuple is already generated and checked for consistency. So σ' cannot be a tuple between σ and the tuple returned by the function NextTuple.

Another possibility is that there can be a candidate tuple σ' between σ and λ. Then $\sigma'[1...k]$ should be equal to $\lambda[1...k]$ (lines 7-11). This is not possible candidate since λ is not a valid support tuple. □

Lemma 5.3. *The algorithm does not remove any consistent value from the domain of variables.*

Proof. A value is removed from the domain of a variable only if the value is not arc consistent i.e. there is no valid support tuple for the variable value. Thus, the algorithm does not remove any consistent value from the variables' domains so the algorithm is sound. □

Lemma 5.4. *When the algorithm terminates, then the domain of variables contain only arc consistent values (or some domain is empty).*

Proof. Every value in the domain has to pass the consistency test and inconsistent values will be deleted. When an inconsistent value is deleted and if the deleted value is part of a valid support tuple, then all variable values in that tuple are checked for consistency again. Hence when the algorithm terminates only consistent values remain in the domain. □

5.8 Complexity of the GACCC-op Algorithm

Lemma 5.5. *The worst case time complexity of the algorithm is* $\boldsymbol{O}(en^2d^n)$.

Proof. The worst-case time complexity of GACCC-op depends on the arity of the constraints involved in the constraint network. The greater the number of variables

involved in a constraint, the higher the cost to propagate it. Let us first limit our analysis to the cost of enforcing GAC on a single conjunction constraint, S_i of arity n (n = $|X(S_i)|$) and d = size of the domain of the variable. For each variable $x_i \in X(S_i)$, for each value a$\in D(x_i)$, we look for supports in the search space where $x_i = a$, which can contain up to d^{n-1} tuples. If the cost to check whether a tuple satisfies the constraint is in $\mathbf{O}$(n), then the cost for checking consistency of a value is in $\mathbf{O}(nd^{n-1})$. Since we have to find support for nd values, the cost of enforcing GAC on S_i is in $O(n^2d^n)$. If we enforce GAC on the whole constraint network, values can be pruned by other constraints, and each time a value is pruned from the domain of a variable involved in S_i, we have to call **SeekValidSupportSet** on S_i. So, S_i can be revised up to nd times. Fortunately, additional calls to **SeekValidSupportSet** do not increase its complexity since, $last(S_i, y, b)$ ensures that the search for support for (x_i, a) on S_i will never check twice the same tuple. Therefore, in a network involving e number of constraints with arity bounded by n, the total time complexity of GACCC-op is in $\mathbf{O}(en^2d^n)$. □

Lemma 5.6. *The worst case space complexity of the algorithm is* $\boldsymbol{O}(en^2d)$.

Proof. Consistency-based search generates at most one valid support tuple for each variable value. Then there are at most nd tuples in memory for a constraint. One tuple will contain n elements. Then the set of all tuples which are a valid support for a constraint can be represented in $\mathbf{O}(n^2d)$. Therefore, in a network involving e constraints with arity bounded by n, the total space complexity of GACCC-op is in $\mathbf{O}(en^2d)$. □

5.9 Experimental Results

5.9.1 Case Study: CSPs

Table 5.2: Time for consistency-based search for 3-SAT Problem Instances

No: of Variables	No: of Constraints	No of tuples with GACCC	No of tuples with GACCC-op	%improvement in time
10	14	98	76	12.34
12	14	96	70	10.66
14	14	103	82	11.46
18	30	168	120	19.86
20	30	170	131	17.96
20	40	256	216	17.43

We performed our experiments on different CSP models. The first is a model for the 3-SAT problems [78] with different number of variables. The SAT problems with a set of clauses are converted into CSPs containing the same set of variables. In our case, we set i=2, k=2 and j=5 (i, j and k are the values from the heuristic for generating conjunction set) and generated the conjunction set. Hence the model contained some conjunction set which has 2 variables shared between member constraints. The results are shown in Table 5.2. For each problem the experiment is repeated for 20 instances. We implemented the GAC-scheme on conjunction of constraints and the proposed algorithm using the C++ language. The result shows that the proposed algorithm attains consistency faster than the existing algorithm.

In order to show the effect of consistency check on constraint solvers associated with CDTG, we took three different CSP benchmark problems, Langford Series, Magic Sequence and Golomb Ruler. The three CSPs are modeled using SystemVerilog. The SystemVerilog constraints are then used for consistency-based search. The reduced input variable domain are generated by the consistency-based search. This reduced domain is then used by the VCS (CDTG tool) to generate the CSP solutions. From

Table 5.3: Results for Benchmark CSP Problems using VCS

			Improvement After Domain Reduction	
Benchmark Problem	No: of Variables	No: of Domain Values	Time (%)	Memory (%)
Langford Series	6	3	10.0	23.5
	8	4	21.4	27.7
	14	7	25.0	40.8
Golomb Ruler	3	4	8.3	23.2
	4	7	7.1	28.2
	5	12	9.5	39.1
	6	18	13.8	73.1
Magic Sequence	4	4	30.0	50.0
	5	5	40.0	71.6
	7	7	55.0	73.3
	8	8	62.5	81.5

Table 5.3, we can see that the time to solve the three CSPs is reduced after giving the reduced domain. In the cases of Magic Sequence the time is significantly reduced, because, after the domain reduction the number of domain values in most of the variables is reduced to one. Since the domain of input variables are reduced, the search space which has to be covered by the solver is reduced. This helps the solver to generate the solutions for CSP in less time and with reduced memory consumption.

5.9.2 Case Study: Xbar Switch

For a real life case study, the example of an Xbar crossbar switch [79] was chosen. According to the design specification, the Xbar consists of four receive and transmit ports. The data request can either be in protocol A or B. As shown in Figure. 5.1 *port 0* supports protocol A, *port 1* supports protocol B and *port 2* and *port 3* support both protocol A and B. Therefore, for example, *port 0* can only respond to an incoming data request following protocol A and transfer the incoming data to a port supporting protocol A (*port 2* and *port 3*).

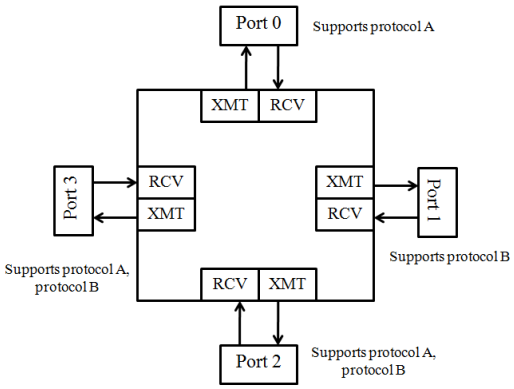

Figure 5.1: Xbar Switch Design

In the proposed methodology, the requirements that must be satisfied by the design are extracted from the design specification. In this example, one of the requirements to be verified is, whether each of the Xbar transmit port responds correctly to the data requests arriving at that port. The next step in the methodology is to convert the requirement into verification scenarios. The above requirement is converted into two verification scenarios, where, in one verification scenario the protocol for data transfer is A and in the other the protocol is B. When the protocol for data transfer is protocol A, the incoming (source) port can be *port 0* or *port 2* or *port 3*. For each possible source port, Table 5.4 shows the corresponding possible output (destination) ports. The stimulus generator has to randomly generate the different possible combinations of source port and destination port to verify the above requirement.

Table 5.4: Verification Scenarios

Protocol	Source Port	Destination Port
Data request Protocol A	0	2,3
	2	0,3
	3	0,2
Data request Protocol B	1	2,3
	2	1,3
	3	1,2

The verification scenarios are then modeled into CSPs. The first verification scenario can be modeled into a CSP by the following constraints:

```
constraint c1{foreach (dest[i])dest[i] inside{[0-3];}
constraint c2{foreach (source[i])source[i] inside{[0-3];}
constraint c3{foreach (source[i])protocol==A-> (source[i]!=1);}
constraint c4{foreach (dest[i])protocol==A-> (dest[i]!=source[i] & 1);}
```

In the verification scenarios mentioned above there are two variables: source port and destination port. The domain of source port is 0,1,2 and 3, and the domain of destination port is 0,1,2 and 3. In our model the source port is called source and the destination port is called dest. The constraint *c1* specifies the domain of the variable source and the constraint *c2* specifies the domain of the variable dest. The number of input stimuli generated by the model will be *i*. When the protocol used for data request is protocol A, *port 1* cannot be a source port. This constraint is implemented by the constraint *c3*. Similarly when the protocol used for data request is protocol A, *port 1* cannot be a destination port and the destination port cannot be same as the source port. This constraint is implemented by *c4*.

These constraints are given to the constraint solver which generates the different possible combinations of source port and destination port. To generate the different values of the source port and destination port, the solver has to traverse through the entire search space which contains 16 possible values(4 source port * 4 destination port). Here we can see that because of the constraints *c3* and *c4*, *port 1* cannot be part of source port or destination port. In the proposed methodology consistency-based search is performed on the above constraints. The consistency-based search will also show that *port 1* cannot be part of the domain of source port and destination port. Hence *port 1* will be removed from the domain of source port and destination

port. The modified domain is then given to the constraint solver to generate the different combinations of source port and destination port. The new search space contains only 9 possible combination (3 source ports * 3 destination ports). Since the search space is reduced, the solver can generate solutions faster.

Table 5.5: Results for Xbar Switch using VCS with Domain Reduction

	Coverage with simulation time=0.6sec		Coverage with simulation time=1.2sec		Coverage with simulation time=3sec	
No:of ports	M1	M2	M1	M2	M1	M2
16	84.71	87.32	86.32	89.41	87.42	90.54
32	74.81	82.22	80.58	87.27	83.78	89.59
48	47.52	71.18	49.75	75.10	52.89	83.09
64	34.05	64.67	34.90	71.55	36.57	77.31
80	27.48	67.98	28.09	70.04	29.21	74.27

In order to show the scalability, the number of ports is increased from 16 to 80. Input stimulus for both the verification scenarios were generated. The cover points were defined on the possible source and destination port values. In the case of 16 ports, the ports 1-4 support protocol A, the ports 5-8 support protocol B and the ports 9-16 support both protocol A and B. The source port can have 12 values and the destination port can also have 12 values in both the verification scenarios. This potentially yields 144 $(12 * 12)$ cases that must be covered in each of the verification scenario. Obviously, not all cases are possible (e.g. the source port and destination port cannot be the same), so the actual number of cases is, in fact, lower. For both the verification scenarios together there are 264(132+132) plausible cover points.

The time to manufacture and market is the main bottleneck in verification. So the verification engineer is given a fixed time to verify the design. In this experiment we also kept the verification time constant. In Table 5.5 M1 represents the stimulus

generation without reduced domain and M2 represents stimulus generation with reduced domain. The consistency-based search took about 20ms. For the same time period the CDTG tool was able to attain more coverage when compared to existing methodology. In some of the cases the coverage is increased by about 40%. So with a small overhead for consistency-based search, we were able to obtain higher coverage when compared to existing CDTG methodology.

5.10 Conclusion

We presented a consistency-based search algorithm which helps to generate partial solution which are required for domain partitioning. The proposed algorithm can be used with n-arity constraints. The proposed algorithm is much more efficient than the GACCC algorithm, since it requires less number of tuples to determine consistency. The results showed that the proposed algorithm helps in getting solution faster and with reduced memory consumption.

Chapter 6

Estimation of number of clusters

6.1 Introduction

A basic feature of the search space (data set) is that it contains k number of sub populations or clusters [77]. One way to attain evenness in input stimuli generation is to partition or cluster the input domain and generate input stimuli from the clusters. In k-partition/clustering technique, given a set of n points in Euclidean space and an integer k, the clustering algorithm will partition the n points into k subsets, each with a representative known as a centroid. Estimating k is a preliminary step in any cluster analysis. However many cluster algorithms consider k as an input chosen by the user. Hence these techniques arises the question, "What is the best number of clusters in a dataset?".

Clustering problems have been studied for the past many years by data management and data mining researchers. A thorough review of the clustering literature, can be found in a plethora of surveys [80][81][82][83]. There are several approaches to find the optimal number of clusters.

In one approach adopted, the data set is plotted as an evaluation graph, where

the values in the y-axis represents any evaluation metric, such as: distance, similarity, error, or quality and the x-axis values are number of cluster from 2,...,n (number of elements in a data set). The knee of this obtained graph provides the best/optimum number of clusters. There are many methods to find the *knee* of the graph. Some of the methods evaluate each point in the evaluation graph, and use the point that either minimizes or maximizes some function, as the number of clusters. Such methods include the Gap statistic [84] and prediction strength [85]. These methods generally require the entire clustering algorithm to be run for each potential value of k (2,...,n). Hence, this is computationally expensive and requires a large amount of time.

Another way to determine the knee of a curve is the L method [86]. The L method makes use of an evaluation function to construct an evaluation graph where the x-axis is the number of clusters and the y-axis is the value of the evaluation function.

L Method

In Figure. 6.1, starting from the right, the graph continues to the left in a rather straight line for some time (points marked by dots). In this region, many clusters are similar to each other and should be merged. Another distinctive area of the graph is on the far left side where the graph is a straight line for some time (points marked by triangle). The increase in distance indicates that very dissimilar clusters are being merged together. The knee region is the area where the above two lines meet each other. Clusterings in this knee region contain a balance of clusters that are both highly homogeneous, and also dissimilar to each other. Determining the number of clusters in this knee region will therefore give the best number of clusters.

In order to determine the location of the transition area or knee of the evaluation graph, a property that exists in these evaluation graphs is used. The regions to both

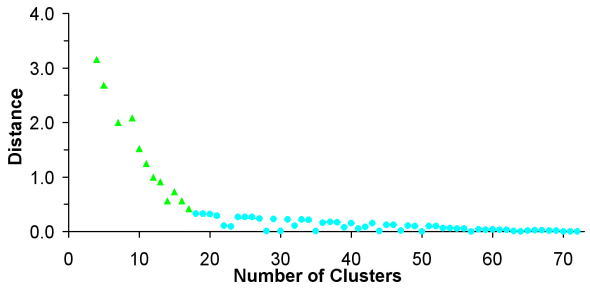

Figure 6.1: Number of Cluster vs Distance

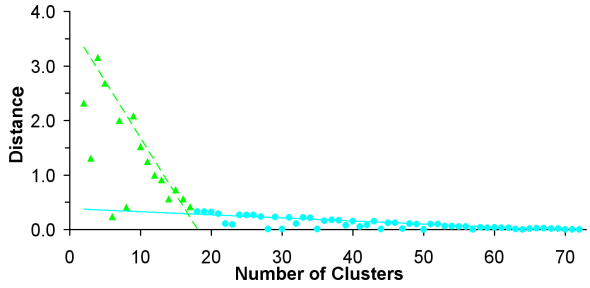

Figure 6.2: Finding the Number of Clusters using the L Method

the right and the left of the knee (see Figure. 6.2) are often approximately linear. If a line is fitted to the right side and another line is fitted to the left side, then the area between the two lines will be in the same region as the knee. The value of the x-axis at the knee can then be used as the number of clusters to return.

To create the two lines that intersect at the knee, the pair of straight lines that most closely fit the curve is to be determined. Both lines together must cover all of the data points (or max possible number of data points). Hence if one line is small, the other must be large to cover the rest of the remaining data points.

The L method algorithms treat every data point as a cluster. This will result in an evaluation graph as large as the original data set. In such an evaluation graph, very large values of x (number of clusters) are irrelevant. In the following section we propose a new methodology for the determination of best number of cluster which requires less number of statistical evaluation (determination of cost for a cluster).

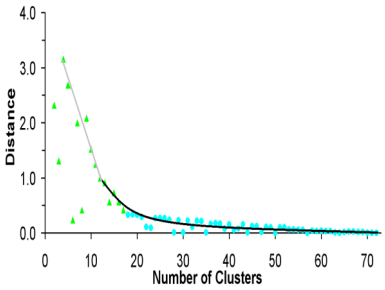

Figure 6.3: Knee Value Under-estimated

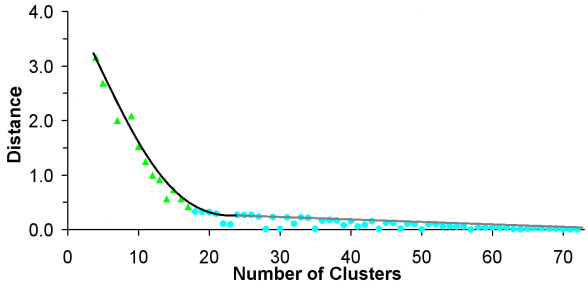

Figure 6.4: Knee Value Over-estimated

6.2 Proposed Method

We can find the number of clusters in a data set by searching for the number of cluster at which there is a knee, peak or dip in the evaluation measure, when it is plotted against the number of clusters. The existing approaches to estimate the optimal number of clusters generally depend on some clustering operation for each number of clusters. These approaches are time consuming. We take a different approach to estimate the number of clusters. We devise an algorithm which can efficiently determine the best number of clusters.

In order to find the knee of the graph, two lines, one nearly parallel to x-axis and another parallel to y-axis are required. To determine the equation of the straight lines at least two point are required. One set of points should be before the knee of the graph and another set must be after the knee of the graph in order to get the above mentioned parallel lines. According to [81] if no information is available, for a

data set containing n elements, the best number of cluster is nearly equal to $\sqrt{n/2}$. Hence, this could be the value where the knee of the graph exist.

Algorithm 6.1: Number of clusters

1: Number of clusters (**in** n-number of elements in search space): k-best number of clusters
2: $n_{min} = 2$, $n_{mid} = \sqrt{n/2}$, $n_{max} = 2 * \sqrt{n/2}$
3: find 3 random natural numbers x_{l1}, x_{l2}, x_{l3} between n_{min} and n_{mid}
4: let y_{l1}, y_{l2}, y_{l3} be the cost of clustering using k-means method for x_{l1}, x_{l2}, x_{l3} respectively
5: *linel*= line passing through the points (x_{l1},y_{l1}), (x_{l2},y_{l2}), (x_{l3},y_{l3})
6: find 3 random natural numbers x_{r1}, x_{r2}, x_{r3} between n_{mid} and n_{max}
7: let y_{r1}, y_{r2}, y_{r3} be the cost of clustering using k-means method for x_{r1}, x_{r2}, x_{r3} respectively
8: *liner*= line passing through the points (x_{r1},y_{r1}), (x_{r2},y_{r2}), (x_{r3},y_{r3})
9: n_{mid} = value on x-axis (no: of clusters) where *linel* and *liner* intersect each other
10: let C_{n1}, C_{n2}, C_{n3} be the cost of clustering using k-means method for $n_{mid} - 1$, n_{mid}, $n_{mid} + 1$ respectively
11: S_{mean}= mean of *linel* and *liner* slopes
12: S_1=slope of line passing through the points ($n_{mid} - 1$,C_{n1}) and (n_{mid},C_{n2})
13: S_2=slope of line passing through the points (n_{mid},C_{n2}) and ($n_{mid} + 1$,C_{n3})
14: **while** best number of cluster = nil **do**
15: **if** ($S_1 > S_{mean}$) & ($S_2 < S_{mean}$) **then**
16: best number of clusters = n_{mid}
17: **else**
18: **if** ($S_1 < S_{mean}$) **then**
19: $n_{mid} = n_{mid}$ -1
20: **end if**
21: **else**
22: **if** ($S_2 > S_{mean}$) **then**
23: $n_{mid} = n_{mid}$ +1
24: **end if**
25: **end if**
26: **end while**

If $\sqrt{n/2}$ is not in the knee region, from Figure. 6.3 (knee value under-estimated) and Figure. 6.4 (knee value over-estimated), it can be seen that the shape of the graph is a combination of straight line and a semi parabola (dark line). Minimum three point are required to find the equation of a parabola and two points are required to find the

equation of the line. Hence three point are used to determine the equation. Three values are chosen randomly between 2 and $\sqrt{n/2}$ and the equation of the line/semi parabola passing through the three points is generated.

Another three values are chosen randomly between $\sqrt{n/2}$ and $2 * \sqrt{n/2}$ and the equation of the line (not necessarily a straight line) passing through the three points is generated. These two equations represent the required two lines. Then the point of intersection of these two lines in the first quadrant is determined. Once the intersection point is found the algorithm generates the cost for the 3 consecutive number of clusters $n_{mid} - 1$, n_{mid}, $n_{mid} + 1$ (C_{n1}, C_{n2}, C_{n3} respectively). The slope of the straight line (S_1) passing through the points (C_{n1}, $n_{mid} - 1$) and (C_{n2}, n_{mid})is determined. Also the slope of the straight line (S_2) passing through the points (C_{n2}, n_{mid}) and (C_{n3}, $n_{mid} + 1$) is determined. Then the mean value of the slopes (S_{mean}) of the two line is determined.

Next, the algorithm tries to find the location of N (best number of cluster) in the graph. For any point to be the knee of the graph, the difference between S_1 and S_2 should be large (i.e. rate of change in slope should be large). If $S_1 > S_{mean}$ and $S_2 < S_{mean}$, this means that the point n_{mid} is in between the two lines which are parallel to x-axis and y-axis respectively. Hence n_{mid} is the best number of clusters in the data set. If $S_1 < S_{mean}$, it means that n_{mid} is located towards the right side of the graph. But the knee of the graph (N) is towards the left. So the n_{mid} value is decremented to move towards the left of the graph. If $S_2 > S_{mean}$, it means that n_{mid} is located towards the left side of the graph. But N is towards the right of graph. So the n_{mid} value is incremented to move towards the right of the graph.

Theorem 6.1. *The worst case complexity of the algorithm is* $\boldsymbol{O}(\sqrt{n/2})$.

Proof. The algorithm starts with finding equation of a line between 2 and $\sqrt{n/2}$ and equation of another line between $\sqrt{n/2}$ and $2^*\sqrt{n/2}$. Then the point of intersection of the two lines is found out. In worst case the point of intersection can be a point near 2 or near $2^*\sqrt{n/2}$. Then the algorithm will iterate until it it reaches the knee of the graph which is when the slope $S_1 > S_{mean}$ and $S_2 < S_{mean}$. If the point of intersection is near 2, then $S_2 > S_{mean}$. The algorithm can iterate only until the point $\sqrt{n/2}$ after which $S_1 < S_{mean}$. If the point of intersection is near $2^*\sqrt{n/2}$, then $S_1 < S_{mean}$. The algorithm can iterate only until the point $\sqrt{n/2}$ after which $S_2 > S_{mean}$. Hence the worst cases complexity is $\mathbf{O}(\sqrt{n/2})$. □

6.3 Experimental Results

The goal of this evaluation is to demonstrate the ability of the proposed method to identify the best number of clusters in a given data set. The algorithm is first used with a set of data where the evaluation graph has different distinct shapes. Figure. 6.5 shows the shape of the evaluation graphs used for the determination of the best number of clusters. In some of the graphs, from the shape of the graph clearly shows joining of the two distinct lines and the best number of clusters. In some others, the graph is a smooth curve where the number of clusters is not so visible. Table 6.1 shows the best number of clusters determined by the proposed method and the number of time the cost function was called. M1 represents the result obtained by L method and M2 the result obtained by proposed method.

For L method, the cost function is called for all possible values of n (number of clusters). Evaluation of cost is a time consuming process. Also finding the two lines that is parallel to the axis and passing through majority of the points is also a time consuming process. But with the proposed method, the number of time the cost

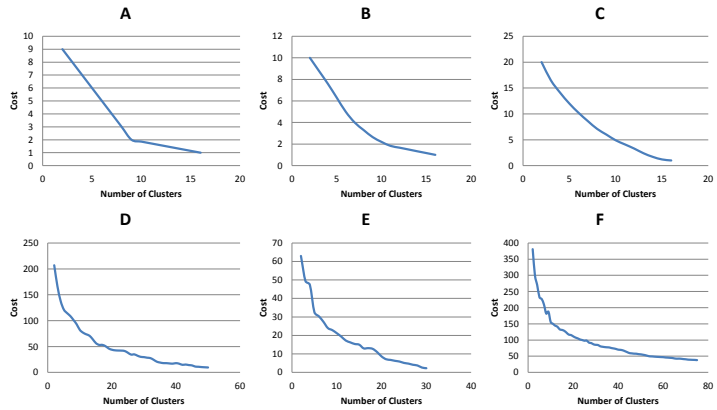

Figure 6.5: Test Cases

function was called is less when compared to L method. Hence the proposed method is faster than the L method.

We also used five sets of diverse data to compare the answer provided by L method to the proposed method. The diverse data sets varied in size, number of clusters, separation of clusters and density. The five data sets that were used are:

1. A data set with four well separated clusters with 1289 instances and 50 variables (dimensions).

2. A data set with five well separated clusters with 2000 instances and 200 variables.

3. A data set with four well separated clusters with 1286 instances and 500 variables.

4. A data set with ten well separated clusters with 3814 instances and 800 variables.

5. A data set with ten well separated clusters with 2729 instances and 1000 variables.

Using the tool Weka (which is a data mining tool) [87], we clustered the five diverse data sets mentioned above by using K-means algorithm. The cost for clustering

Table 6.1: Test Cases with Different Evaluation Graph Shapes

Case	# Clusters			#times cost function is called	Time in sec	
	Actual	M1	M2		M1	M2
A	9	9	9	10	0.16	0.10
B	9	8	9	10	0.16	0.10
C	7	7	7	11	0.16	0.11
D	17	15	17	29	0.50	0.30
E	16	13	17	18	0.35	0.19
F	19	18	19	36	0.75	0.38

Table 6.2: Test Cases with Diverse Data Sets

Case	# Clusters		#times cost function is called	Time in sec	
	M1	M2		M1	M2
1	5	5	17	3.90	1.70
2	5	5	16	4.50	1.90
3	4	4	24	5.70	3.20
4	6	9	12	5.10	3.10
5	5	10	15	6.65	3.90

is then determined and this is plotted against the number of clusters. In L method, the maximum number of cluster is equal to the number of datapoints/instances. Computing the cost of all the possible number of clusters is computationally expensive and is not required. Hence in this experiment we limited the maximum number of clusters to 50. Then the best number of cluster is determined using L method and the proposed method. The results are shown in Table 6.2. M1 represents the result obtained by L method and M2 the result obtained by proposed method. 8 out of 11 times, the proposed method determined the correct number of clusters. In other cases the obtained results were very close. In all the above cases the actual number of clusters weare known since the data sets were synthetic.

The proposed method fine grains the search for the best number of clusters only when an approximate value for best number of clusters is determined. Hence, the frequency of calling the cost function is very small when compared to L method. This can be seen from the number of time the algorithm calls the cost function. Hence the

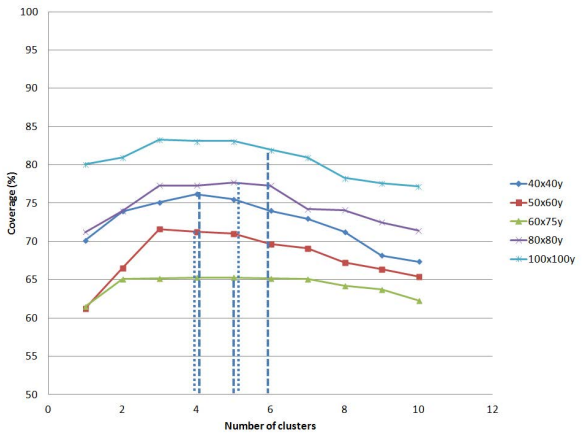

Figure 6.6: Coverage vs Number of Clusters

proposed method runs more quickly than then the L method.

In the above experiments, we used data sets where we knew the number of clusters in the data set. But in real life problems this is not the case. So in order to see the effect of the proposed method on domain partitioning and coverage obtained, we considered a search space with two variables but with different number of domain values. Then we clustered the search space with the number of clusters varying from 1...10. By using the proposed method we generated the optimum number of cluster for each search space.

Figure. 6.6 shows the coverage obtained for different values for the number of clusters (the label on the right side shows the number of domain values for the variable x and y). The straight line shows the optimal number of clusters generated for each case of varying number of domain values. It is very clear that the proposed method was able to give a good estimate of the best number of clusters and as a result the coverage obtained is higher than the coverage obtained by using some random number of clusters. In some cases the coverage obtained by partitioning is less than the coverage obtained with no partitioning or when the number of cluster is equal to one. This is because, in those cases, some of the partitions has no solutions. Hence it

affected the overall coverage obtained.

6.4 Conclusion

We have detailed our number of clusters determination method. It has been shown to work reasonably well in determining the number of clusters or segments for a given data set. In our evaluation, the proposed method was able to determine the number of clusters for the given data set. The proposed method is much more efficient than the existing L method, since it requires only a fraction of a second to determine the number of clusters rather than minutes or even many hours in the case of other methods.

Chapter 7

Domain Partitioning Algorithm

7.1 Introduction

In a CSP, the solutions are clustered together in the search space [77]. Hence partitioning the search space into clusters and generating solutions from the partitions can improve the evenness of the solutions generated by the solver. 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 result, it is computationally expensive. We propose a method to cluster the search space using the tuples generated by consistency search. The clustering of variable domain into n groups can be divided into the following three steps:

7.2 Step 1: Selection of n Tuples

First, the 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 which satisfies the highest arity constraint have to be selected from the generated tuples. The selected n tuples should be far away (different) from each other. Selection of n values which are far away from each other is a hard problem to solve [88]. 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 7.1.

Algorithm 7.1: Selection of n Tuples

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

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. Constraint $C3$ is the highest arity constraint and the tuple generated for the constraint $C3$ is shown in the Table 7.1. If n is set to 4 we need to select 4 tuples which satisfies $C3$ from the list and are far away from each other. The selected tuples are shown in the Table 7.2.

Table 7.1: Tuple after Consistency check on constraint C3

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

7.3 Step 2: Generation of n Partition Tuples

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

Table 7.2: 4 Tuples Selected from the Tuples Generated 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

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 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.

Algorithm 7.2: 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: τ_{Cl2} = tuple returned by consistency search that satisfies the constraint C_{l2} and domain values of $comvar$ is equal to $comval$
9: **if** τ_{Cl2} = nil **then**
10: $comval$ = next lexicographic higher value
11: Go to step 8
12: **else**
13: Update the domain value of variables in $\tau_{CHN}[i]$ with the domain values in τ_{Cl2}
14: **end if**
15: **end for**
16: **end while**

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 7.2. After

Table 7.3: 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

this process, the n partition tuples generated for the above CSP are as shown in the Table 7.3.

7.4 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. In Table 7.3 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 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. Figure 7.1 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 8.1 gives the domain values of all the variables of the 4 groups used for solution generation. This partitioned domain

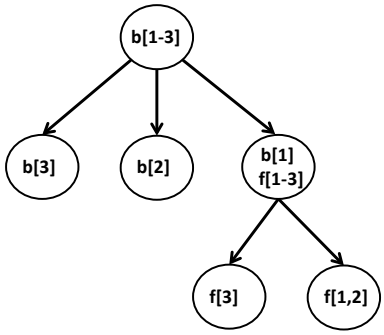

Figure 7.1: Partitioning of Variable Domain

Table 7.4: 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

values along with the CSP constraints are then given to the constraint solver.

7.5 Proofs

Theorem 7.1. *In algorithm 7.2, if a variable value is different in* $\tau_{CHN}[i]$ *and* τ_{Cl2} *then the highest domain value is assigned in* $\tau_{CHN}[i]$.

Proof. 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. □

Theorem 7.2. *The partitioning of the tuples is equivalent to partitioning of the solutions of the CSP.*

Proof. 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 and this will help to generate partitions which contains solutions (partitions with no solutions is not useful). 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. □

Theorem 7.3. *For a set of m euclidian points (S), if T is the solution returned by the proposed algorithm (T contains n points selected from the m euclidian points) and T_{op} be the optimal solution, then $Cost(T) \leq 2 * Cost(T_{op})$ where Cost is the average distance between points.*

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.

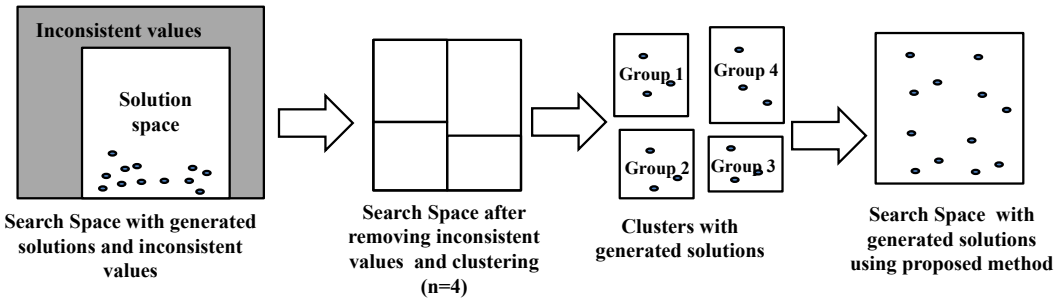

Figure 7.2: Search Space with Generated Solutions

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 $\leq a/2$. As a result the $Cost(T_{op})$ is increased by a factor of $a/2$. Similarly, considering all other points in T_{op}, the cost of T_{op} is $\geq a/2$. □

7.6 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.

7.6.1 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. Some solution may be repeatedly generated. 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.

7.6.2 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 σ_{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 σ_{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 δ_{DNP} where $\delta_{DNP} = \frac{\sigma_{DNP}}{\overline{d}_{min}}$

7.6.3 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 δ_{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

Table 7.5: Differentsoln Evaluation on Random Cases

#vars	#cons	Differentsoln	
		M1	M3
31	9	5351	17232
34	24	4987	19497
36	16	6323	12943
38	13	7208	22268
40	20	6766	21919

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 [89].

7.7 Experimental Results

We used Weka[87], 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 7.6.

In Table 7.5, we list five cases shown in [39]. Columns 1 and 2 indicate the number of variables and constraints respectively. The domain of each variable contains 1024 values (0 to 1023). The number of different solutions generated is shown in columns 3 and 4. 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. 10^6 solutions were generated. We can see that the number of different solutions generated by the proposed methodology is nearly 6 times than the random generation method.

Table 7.6: Evenness Evaluation on Random Cases

#vars	#cons	σ_{DNP}			δ_{DNP}			δ_{KM}(k=100)			δ_{KM}(k=1000)		
		M1	M2	M3	M1	M2	M3	M1	M2	M3	M1	M2	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

To ensure the evenness of generated solutions, we used K-Means Clustering and Distance of Nearest Neighbor analysis. Table 7.6presents the results obtained. We used the same CSPs, which were used for different solution evaluation. M2 represents the result obtained using the technique RSSDE [39]. 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 the number of centroids is set to 1000.

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 σ_{DNP}, better the distribution. δ_{DNP} is the ratio between σ_{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 δ_{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 7.6 we can see that the values of σ_{DNP} and δ_{DNP} are smaller and the value of δ_{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.

7.8 Conclusion

The distribution of generated input stimuli by CRV tools can be improved by domain partitioning. We presented a domain partitioning algorithm based on consistency search for input stimulus generation. Experiments showed that the proposed domain partitioning algorithm helped to improve the distribution of input stimuli generated by a CRV tool called VCS.

Chapter 8

Implementation and Evaluation

We implemented a tool called DPCGEN that incorporate the algorithms for consistency search, determination of number of clusters and domain partitioning. In this chapter, we present an overview of the tool along with the results obtained by applying the tool to a variety of challenging CSPs.

8.1 Implementation

We implemented our algorithms to explore the search space of a given design in C++. The tool will take SystemVerilog constraints and the domain of the input variables as input and generates the reduced domain as output. For our purpose we considered a subset of SystemVerilog constraints which can be given as input to the tool. Our tool can handle unary constraint, binary constraints and some high order constraints. The high order constraints considered includes arithmetic, logical and implication constraints.

The DPCGEN consists of 3 main modules - consistency search, determination of best number of clusters and a domain partitioning module. Figure 8.1 presents an

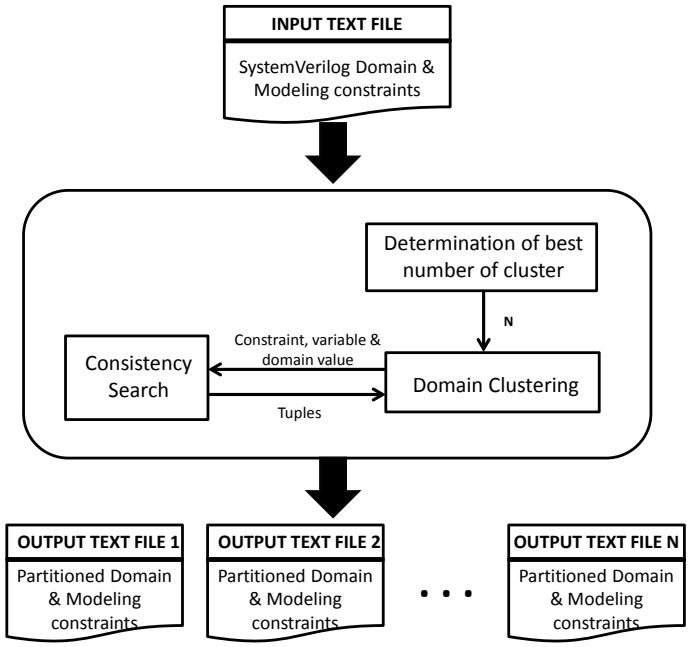

Figure 8.1: Implementation of Proposed Methodology

overview of the DPCGEN.

The DPCGEN takes as input a *text file* which contains the SystemVerilog codes to specify the CSP. The text file contains declaration of the random variables, constraints which specifies the domain of random variables and constraints for modeling the CSP. Figure 8.2 shows a sample input file for the tool. The input file contains declaration of variables (lines 1-8), domain constraints (lines 9-13) and the modeling constraints (lines 14-17).

The output generated by the DPCGEN is a set of text files which contains the declaration of the random variables, constraints which specifies the partitioned domain of random variables and constraints for modeling the CSP. Figure 8.2 shows a sample set of output files generated by the DPCGEN. Each of the output file contains declaration of variables (lines 1-8), domain constraints (lines 9-13), the modeling constraints (lines 14-17) and the domain of the variables involved in partitioning (line

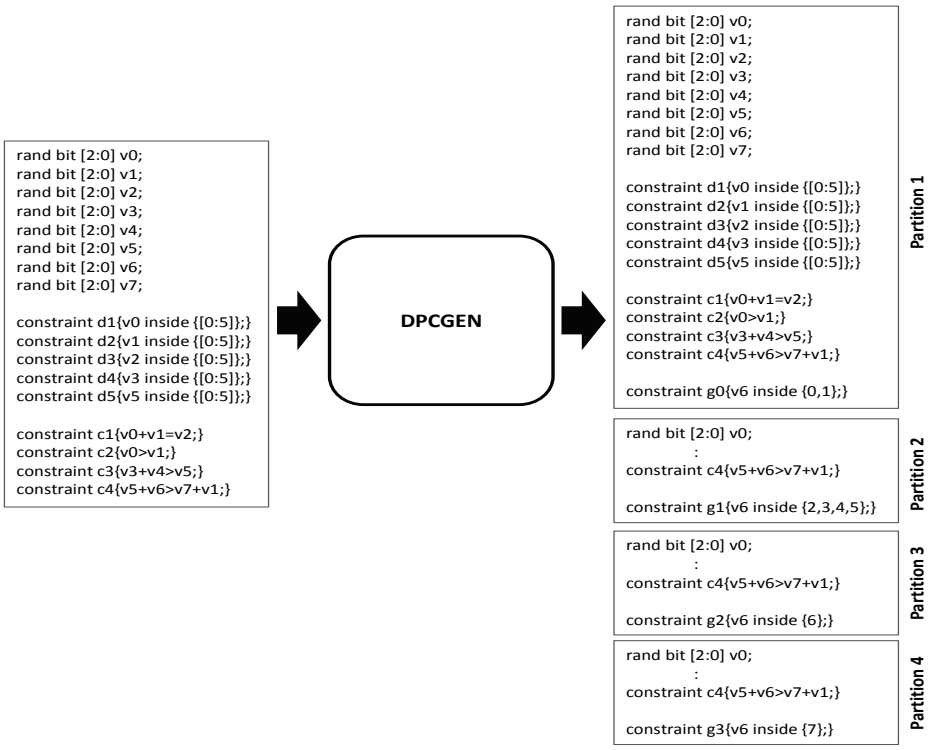

Figure 8.2: Sample Input/Output

18). In the output files, the declaration of variables and modeling constraints remain the same as in input file. Only the domain constraints are modified in the output files.

8.2 Experiments and Results

Using the above implementation, we experimented on several small examples and a design. Following are the CSPs we used:

8.2.1 Random CSPs

In order to show the effect of the consistency search and domain clustering on coverage we used some random CSPs. The random CSPs contain 20 to 40 variables with 14

to 29 general constraints. The domain of each variable was from 0 to 11. For each case, we generated 10^4 patterns using VCS, with and without domain clustering. The optimum number of partitions in the search space of the highest arity constraint is determined (as mentioned earlier partitioning of the highest arity constraint is equivalent to partitioning of the CSP) and used for domain partitioning. In Table 8.1 the columns 4 and 5 gives the number of arithmetic operators in the CSP. The columns 6 to 9 gives the number of comparators in the CSP. Similarly column 10 gives the number of logical operator (not) in the CSP. M1 represents the results obtained by constraint random test generation and M2 represents the results obtained by using the proposed domain clustering technique. We generated 10^4 solutions for the same problem. The results show that, the proposed methodology was able to attain more coverage with the generated CSP solutions.

Table 8.1: Random CSPs

Case	No: of		Arithmetic		Comparator				Logic	Coverage(%)	
	Variables	Constraints	+	*	<	>	=	!=	not	M1	M2
1	21	14	2	8	3	8	1	3	3	72	77
2	24	23	5	6	8	8	5	3	1	63	69
3	25	24	5	7	11	10	3	1	1	67	75
4	26	17	3	7	8	6	1	4	1	68	73
5	31	27	6	14	9	15	1	5	3	70	75
6	33	24	4	7	5	7	5	7	4	71	78
7	34	22	1	12	10	7	2	5	5	66	75
8	35	23	6	11	6	10	0	4	2	65	71
9	38	29	3	8	10	11	2	6	6	62	70

8.2.2 Case Study: CORTEX M0

In the ARM processor line, the Cortex family, consist of cores ranging from low cost micro controller solutions to high end processors capable of supporting large, complex

operating systems. The Cortex-M0 processor is the lowest member of the Cortex-M family. ARM Cortex-M0 processor is the smallest ARM processor available. It has exceptionally small silicon area and low power consumption. The ultra low gate count also enables it to be deployed in analog and mixed signal devices. **Scenarios for verification**

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 programmer's model operation and memory if the N, Z, C and V flags in the APSR satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP.

4. Most of these 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.

The requirements are converted into various verification scenarios. The verification scenarios are then modeled using SystemVerilog constraints. These constraints are the used to generate the input stimuli required for verification.

Experimental Setup

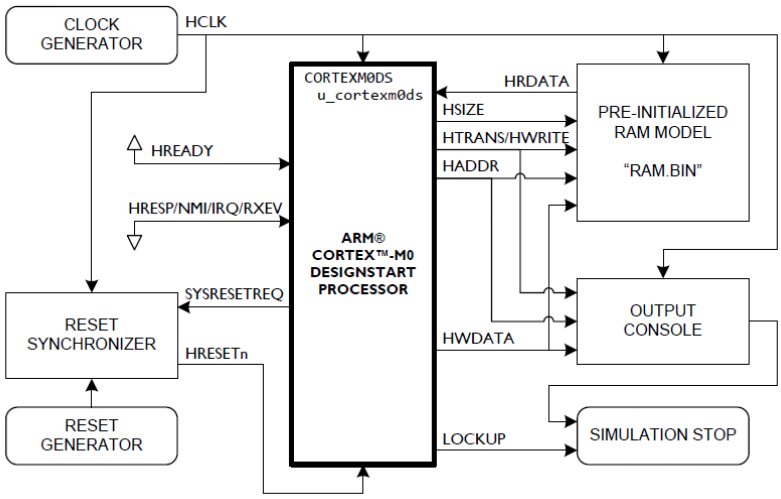

Figure 8.3: Experimental Setup for Cortex-M0

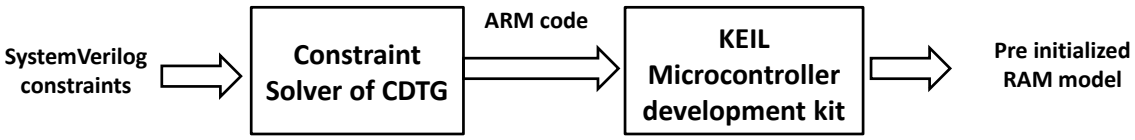

Figure 8.4: Generation of Memory Image

The experimental setup, shown in Figure 8.3, has a Cortex-M0 DesignStart processor connected to a memory image loaded with the basic program. The processor is also connected to a clock and reset generator. It is also connected to a console output which provides a means to output information from the processor. The Cortex-M0 DesignStart processor uses a system bus interface compatible with the AMBA3 AHB-Lite specification. All signals are sampled and driven at the positive clock edges of the AHB-Lite HCLK signal. The memory image for the processor is provided by the ram.bin file. The Figure 8.4 shows how the memory image file ram.bin is generated.

The verification scenarios which are modeled using systemverilog constraints are given to the constraint solver of the CDTG tool. The constraint solver solves the constraints and generates a sequence of ARM codes. The ARM codes are then given to a Keil Microcontroller Development Kit, to generate memory image file required for

the Cortex-M0 processor. The Keil Microcontroller Development Kit fully supports the Cortex-M0 processor. The generated memory image file is then used during simulation.

The Table 8.2 gives one of the ARM instruction sequence used to verify the first requirement. Initially some random values are stored in some of the low registers.

Table 8.2: Sample Instruction Sequence

#	Instruction	Remark
1	MOVS R0 ,135	Assigning some random value to some randomly selected low registers
2	MOVS R1 ,5	
3	MOVS R7 ,24	
4	MOVS R3 ,50	
5	MOVS R2 ,85	
6	CPY R4, R7	Some data processing instructions using the above low registers
7	ORR R7, R3	
8	SUB R7, R2, R4	
9	ADD R1, R6, R1	
10	ADD R4, R5, R3	

Then those low registers were used by the instructions which can access the low registers. Then we verified whether the instructions could correctly access the low registers. In order to determine the coverage we used cross coverage between the instructions and the different registers. For example if we have an instruction of the format *Ins Rd, Rm, Rn*, where *Ins* is the instruction with domain values 1-5, *Rd, Rm* and *Rn* are low registers, then the cross coverage checks whether all possible combinations(5x8x8x8) are generated or not.

We run the experiment with and without domain clustering. The Table 8.3 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.

Table 8.3: Coverage

Scenarios	No of instruction sequence generated	Time (msec)		Coverage (%)	
		M1	M2	M1	M2
1	60	210	190	24.2	32.6
2	61	240	200	34.6	40.2
3	65	230	200	23.5	35.6
4	60	240	210	78.3	83.7
5	62	220	200	67.5	78.9

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Efficient functional verification is a critical issue in modern SoC methodology because verification complexity increases at an exponential rate. Simulation based verification is widely used in modern SoC design flow because formal verification methods have difficulty in verifying complex processors due to the state explosion problem and expertise required. CRV has become the dominant approach in state-of-the-art verification because of its scalability, predictability, and ability to handle complex input constraints. For high productivity, the constraint solver that generates random stimuli for simulation must solve the constraints quickly and produce values that are well distributed over the input space.

To address these issues, this dissertation presented an input stimuli generation approach using domain partitioning of the input domain. The framework we provided consists of a consistency search algorithm, an algorithm to determine the best number of clusters in a data set and a domain partitioning algorithm.

We presented a consistency search algorithm which can conjunct together constraints and can handle n-arity constraints. The proposed algorithm is faster than the existing GACCC algorithm and requires less number of tuples to determine consistency.

We developed a method to determine the best number of cluster for a given data set. The proposed method was able to generate the number of clusters with less number of cost determination function calls than the existing L method.

We then introduced a domain partitioning algorithm which can be used along with the proposed consistency search algorithm. The proposed partitioning algorithm partitioned the domain of the input variables into non overlapping clusters and forces the solvers to generate the required input stimuli from the clusters. The generated input stimuli were evenly distributed in the search space when compared to stimuli generated by existing CRV tools.

Finally, we illustrated the usefulness of our framework by using some CSPs and a microprocessor design. Experimental results demonstrated significant reduction in stimuli generation time as well as memory requirement. It also shows that the evenness of the solutions generated is higher than the existing CRV techniques. Another benefit of this approach is that it can be incorporated with existing CRV tools.

9.2 Future Work

Some future research directions are outlined below.

- The constraints can be divided into hard constraints and soft constraints. Hard constraints are constraints that must be satisfied by the constraint solver and soft constraints help to give directionality to search. If we can incorporate

quality constraints or soft constraints in domain partitioning it will help to attain required coverage faster.

- In our implementation we considered only arithmetic and logic constraints. But constraints in CRV are not only expressed as simple arithmetic or logic relation but may contain complex relations such as CRC (cyclic redundancy check) of a packet. We should improve the implementation by including complex relations constraints.

- In functional verification, many scenarios involve temporal relations.In our implementation we didn't consider temporal relations. Developing a method to represent time (or temporal relation) over variables in the constraint space will help to include temporal relations in verification scenarios.

- The rapid advancement of the Graphics Processing Unit (GPU), over the last few years has opened up a new world of possibilities for high speed computation. We need to look into the development of an easily accessible parallel algorithm model which can accommodate nearly any GPU architecture.

- Finally, we believe this thesis is an important milestone towards building a complete environment for automatic coverage enhancing methodology. Therefore, it is important to develop a method to automatically generate the required constraint for stimuli generation from the specification to fully automate the verification cycle.

Bibliography

[1] M. Law, "http://betanews.com/2013/10/15/breaking-moores-law." Accessed: 15/9/2014.

[2] S. Fine, A. Freund, I. Jaeger, Y. Naveh, A. Ziv, and Y. Mansour, "Harnessing machine learning to improve the success rate of stimuli generation," in *10th IEEE International High-Level Design Validation and Test Workshop*, 2005.

[3] "http://www.mentor.com/products/fv/multimedia/verification-academy-webseminar." Accessed: 15/9/2014.

[4] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall PTR, 1st ed., 2008.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information Computer*, vol. 98, no. 2, pp. 142–170, 1992.

[6] T. Kropf, *Introduction to Formal Hardware Verification*. Springer, 1999.

[7] W. Bruce, G. John, and R. G. Wolfgan, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2005.

[8] "http://blogs.mentor.com/verificationhorizons/blog/tag/functional-verification." Accessed: 15/9/2014.

[9] F. Laurent, A. Yaron, and L. Moshe, "Functional verification methodology for microprocessors using the genesys test-program generator," in *Proceedings of the conference on Design, automation and test in Europe*, ACM, 1999.

[10] E. Guralnik, M. Aharoni, A. J. Birnbaum, and A. Koyfman, "Simulation-based verification of floating-point division," *IEEE Transactions on Computers*, vol. 60, pp. 176 –188, feb. 2011.

[11] R. Maharik, I. Nehama, I. Nikulshin, and A. Ziv, "Solving constraints on the invisible bits of the intermediate result for floating-point verification," in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pp. 76–83, IEEE Computer Society, 2005.

[12] L. Fournier, Y. Arbetman, and L. Levinger, "Functional verification methodology for microprocessors using the genesys test-program generator. application to the x86 microprocessors family," in *Design, Automation and Test in Europe Conference and Exhibition*, 1999.

[13] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, and C. Metzger, "Test program generation for functional verification," in *Design Automation Conference (DAC)*, 1995.

[14] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, and M. Farkash, "X-gen: A random test-case generator for systems and socs," in *IEEE International High Level Design Validation and Test Workshop*, 2002.

[15] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: innovations in test program generation for functional processor verification," *IEEE Design Test of Computers*, vol. 21, no. 2, pp. 84 – 93, 2004.

[16] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "Fpgen - a test generation framework for datapath floating-point verification," in *IEEE International High Level Design Validation and Test Workshop*, 2003.

[17] A. Adir, E. Bin, O. Peled, and A. Ziv, "Piparazzi: a test program generator for micro-architecture flow verification," in *High-Level Design Validation and Test Workshop*, 2003.

[18] A. Nahir, A. Ziv, R. Emek, T. Keidar, and N. Ronen, "Scheduling-based test-case generation for verification of multimedia socs," in *43rd ACM/IEEE Design Automation Conference*, 2006.

[19] C. Liu, C. Chang, J. Jou, M. Lai, and H. Juan, "A novel approach for functional coverage measurement in hdl," in *The IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 217–220, 2000.

[20] L. Bull, "Learning classifier systems: A brief introduction," in *Applications of Learning Classifier Systems*, vol. 150 of *Studies in Fuzziness and Soft Computing*, pp. 1–12, Springer, 2004.

[21] C. Ioannides, G. Barrett, and K. Eder, "Introducing xcs to coverage directed test generation," in *2011 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2011.

[22] Y. Katz, M. Rimon, A. Ziv, and G. Shaked, "Learning microarchitectural behaviors to improve stimuli generation quality," in *Design Automation Conference (DAC)*, 2011.

[23] E. Romero, M. Strum, and W. Chau, "Manipulation of training sets for improving data mining coverage-driven verification," *Journal of Electronic Testing*, vol. 29, no. 2, pp. 223–236, 2013.

[24] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Design Automation Conference*, 2003.

[25] M. Braun, S. Fine, and A. Ziv, "Enhancing the efficiency of bayesian network based coverage directed test generation," in *9th IEEE International High-Level Design Validation and Test Workshop*, 2004.

[26] D. Baras, L. Fournier, and A. Ziv, "Automatic boosting of cross-product coverage using bayesian networks," in *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, pp. 53–67, Springer, 2009.

[27] J. Smith, M. Bartley, and T. Fogarty, "Microprocessor design verification by two-phase evolution of variable length tests," in *IEEE International Conference on Evolutionary Computation*, 1997.

[28] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir, "A genetic approach to automatic bias generation for biased random instruction generation," in *Proceedings of the Congress on Evolutionary Computation*, vol. 1, pp. 442 –448, 2001.

[29] X. Yu, A. Fin, F. Fummi, and E. Rudnick, "A genetic testing framework for digital integrated circuits," in *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE Computer Society, 2002.

[30] H. Shen, W. Wei, Y. Chen, B. Chen, and Q. Guo, "Coverage directed test generation: Godson experience," in *17th Asian Test Symposium (ATS)*, pp. 321 –326, nov. 2008.

[31] F. Corno, F. Cumani, and G. Squillero, "Exploiting auto-adaptive gp for highly effective test programs generation," in *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, pp. 262–273, Springer, 2003.

[32] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Automatic test program generation for pipelined processors," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 736–740, 2003.

[33] F. Corno, G. Cumani, M. Reorda, and G. Squillero, "Fully automatic test program generation for microprocessor cores," in *Design, Automation and Test in Europe Conference and Exhibition*, 2003.

[34] P. Bernardi, K. Christou, M. Grosso, M. K. Michael, E. Sánchez, and M. S. Reorda, "Exploiting moea to automatically geneate test programs for path-delay faults in microprocessors," in *Proceedings of the conference on Applications of evolutionary computing*, pp. 224–234, Springer, 2008.

[35] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *Design Automation Conference*, pp. 783–788, 2005.

[36] K. Eder, P. Flach, and H. Hsueh, "Inductive logic programming," ch. Towards Automating Simulation-Based Design Verification Using ILP, pp. 154–168, Springer.

[37] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Design Automation Conference, 2003. Proceedings*, pp. 286–291, June 2003.

[38] M. Braun, W. Rosenstiel, and K. Schubert, "Comparison of bayesian networks and data mining for coverage directed verification category simulation-based verification," in *Eighth IEEE International High-Level Design Validation and Test Workshop*, 2003.

[39] 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*, pp. 109–114, 2012.

[40] L. Devroye, "Random variate generation for unimodal and monotone densities," *Computing*, vol. 32, no. 1, pp. 43–68, 1984.

[41] 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.

[42] M. Iyer, "Race a word-level atpg-based constraints solver system for smart random simulation," in *Proceedings of International Test Conference*, vol. 1, pp. 299–308, 2003.

[43] M. M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver,"

[44] 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 (CSCWD)*, 2009.

[45] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *Proceedings of the International conference on Computer-aided design*, pp. 258–265, 2007.

[46] L. Lingyi and S. Vasudevan, "Efficient validation input generation in rtl by hybridized source code analysis," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2011.

[47] P. Mishra and N. Dutt, "Graph-based functional test program generation for pipelined processors," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2004.

[48] 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, Digest of Technical Papers. 1999.*

[49] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Simplifying boolean constraint solving for random simulation-vector generation," in *IEEE/ACM International Conference on Computer Aided Design*, pp. 123–127, 2002.

[50] G. Matthias and G. Klaus, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[51] W. Gutjahr, "Partition testing vs. random testing: the influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, pp. 661–674, Sep 1999.

[52] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Transactions on Software Engineering*, vol. 14, pp. 868–874, June 1988.

[53] B. Boris, *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., 1990.

[54] J. Bingchiang and J. W. Elaine, "A simplified domain-testing strategy," *ACM Transactions Software Engineering Methodology*, vol. 3, pp. 254–270, July 1994.

[55] D. J. Richardson and L. A. Clarke, "A partition analysis method to increase program reliability," in *International Conference on Software Engineering (ICSE)*, 1981.

[56] E. Weyuker and T. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 236–246, May 1980.

[57] R. H. J. M. Otten and L. P. P. P. van Ginneken, *The Annealing Algorithm*. Kluwer, B.V., 1989.

[58] D. Cohen, S. Dalal, M. L. Fredman, and G. Patton, "The aetg system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, pp. 437–444, Jul 1997.

[59] H. Dick and T. Ross, "Partition testing does not inspire confidence (program testing)," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1402–1411, Dec. 1990.

[60] S. Panda and N. P. Padhy, "Comparison of particle swarm optimization and genetic algorithm for facts-based controller design," *Applied Soft Computing*, vol. 8, no. 4, pp. 1418 – 1427, 2008. Soft Computing for Dynamic Data Mining.

[61] Y. Lei and K. C. Tai, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, vol. 28, pp. 109–111, Jan 2002.

[62] Y. Lei and K. C. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium*, pp. 254–261, Nov 1998.

[63] G. Sherwood, "Effective testing of factor combinations," in *Proceedings of the 3 rd International Conference on Software Testing, Analysis & Review*, 1994.

[64] K. Marriott and P. Stuckey, *Programming with Constraints: An Introduction*. Adaptive Computation and Machine, MIT Press, 1998.

[65] A. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, pp. 99–118, 1977.

[66] C. Bessire, "Refining the basic constraint propagation algorithm," in *In Proceedings IJCAI*, pp. 309–315, 2001.

[67] M. Arangú, M. A. Salido, and F. Barber, "Ac2001-op: An arc-consistency algorithm for constraint satisfaction problems," in *IEA/AIE (3)*, pp. 219–228, 2010.

[68] M. Arangú, M. A. Salido, and F. Barber, "Ac3-op: An arc-consistency algorithm for arithmetic constraints," in *Proceedings of Conference on Artificial Intelligence Research and Development*, pp. 293–300, 2009.

[69] M. R. C. V. Dongen, "Ac-3d an efficient arc-consistency algorithm with a low space-complexity," in *Proceedings of International Conference on Principles and Practice of Constraint Programming*, pp. 755–760, 2002.

[70] R. Mohr and T. Henderson, "Arc and path consistency revisited," *Artificial Intelligence*, pp. 225–233, 1986.

[71] M. Arangú and M. Salido, "A fine-grained arc-consistency algorithm for non-normalized constraint satisfaction problems," *International Journal Applied Mathematics Computer Science*, pp. 733–744, 2011.

[72] P. Van-Hentenryck, Y. Deville, and C. Teng, "A generic arc consistency algorithm and its specializations," tech. rep., 1991.

[73] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions onSystems, Man, and Cybernetics, Part B: Cybernetics*, vol. 26, pp. 29–41, Feb 1996.

[74] C. Bessière, E. C. Freuder, and J. C. Regin, "Using inference to reduce arc consistency computation," in *Proceedings of international joint conference on Artificial intelligence*, pp. 592–598, 1995.

[75] C. Bessière and J. C. Régin, "Local consistency on conjunctions of constraints," in *Proceedings of the ECAI Workshop on Non-binary constraints*, pp. 53–59, 1998.

[76] "http://www.synopsys.com/community/interoperability/pages/systemverilog.aspx." Accessed: 15/9/2014.

[77] A. J. Parkes, "Clustering at the phase transition," in *Ninth Conference on Innovative Applications of Artificial Intelligence*, 1997.

[78] "http://www.satlib.org." Accessed: 15/9/2014.

[79] S. Iman, *Step-by-Step Functional Verification with SystemVerilog and OVM*. Hansen Brown Publishing, 2008.

[80] J. Kogan, C. K. Nicholas, and M. Teboulle, eds., *Grouping Multidimensional Data - Recent Advances in Clustering*. Springer, 2006.

[81] E. Chandra and V. P. Anuradha, "A survey on clustering algorithms for data in spatial database management systems," *International Journal of Computer Applications*, vol. 24, pp. 19–26, June 2011.

[82] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computer Survey*, vol. 31, pp. 264–323, Sept. 1999.

[83] E. Kolatch, "Clustering algorithms for spatial databases: A survey," tech. rep., Techreport, Department of Computer Science, University of Maryland, 2001.

[84] R. Tibshirani, G. Walther, and T. Hastie, "Estimating the number of clusters in a dataset via the gap statistic," *Journal of the Royal Statistical Society*, vol. 63, pp. 411–423, 2003.

[85] R. Tibshirani, G. Walther, D. Botstein, and P. Brown, "Cluster validation by prediction strength," *Technical Report, 2001, Dept. of Biostatistics, Stanford University*.

[86] S. Salvador and P. Chan, "Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms," in *IEEE International Conference on Tools with Artificial Intelligence*, 2004.

[87] "http://www.cs.waikato.ac.nz/ml/weka." Accessed: 15/9/2014.

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

[89] 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.