Analysis of Constrained Random Variable Solvers in the Field of ASIC Verification

Non-Volatile Memory Solutions Group
Intel of Canada, Ltd.
Vancouver, British Columbia

James Mackenzie
V00737373
Work Term 4
Computer Engineering
jimack@uvic.ca

January 1st, 2014

In partial fulfillment of the requirements of the B.Eng. Degree

Supervisor's Approval: To be completed by Co-op Employer

I approve the release of this report to the University of Victoria for evaluation purposes only.
The report is to be considered (select one): □ NOT CONFIDENTIAL □ CONFIDENTIAL

Signature: __________________________ Position: ________ Date: ______________

Name (print): ______________________ E-Mail: ____________________ Fax #: __________

If a report is deemed CONFIDENTIAL, a non-disclosure form signed by an evaluator will be faxed to the employer. The report will be destroyed following evaluation. If the report is NOT CONFIDENTIAL, it will be returned to the student following evaluation.
# Table of Contents

List of Figures and Tables ................................................................. i

Figures .............................................................................................. i

Tables ............................................................................................... i

Summary ............................................................................................. ii

Glossary of Terms and Symbols .......................................................... iii

1.0 - Introduction ................................................................................. 1

1.1 - Intel and Non-Volatile Memory Solutions Group .................... 1

1.2 - Constrained Random Verification and Stimulus in ASIC Verification .... 2

1.2.1 - Constrained Random Verification and Coverage-Driven Verification .... 2

1.2.2 - Constrained Random Stimulus ................................................. 3

2.0 - Analysis of Solving Constrained Random Variables ................. 4

2.1 - Constraint Solving ..................................................................... 4

2.1.1 - P, NP, NP-Hard, and NP-Complete ........................................ 5

2.2 - Types of Constraint Satisfaction Problems ............................... 6

2.2.1 - Integer Linear Programming and Binary Integer Programming .... 6

2.2.2 - Boolean Satisfiability and Automatic Test Pattern Generation .... 6

2.2.3 - Binary Decision Diagrams ...................................................... 7

2.2.4 - Boolean Unification ................................................................ 7

2.2.6 - Analysis of Approaches Presented ......................................... 7

2.3 - Informal Comparisons of Constrained Random Variable Solvers ... 7

2.3.1 - Using a Constraint Solver to Solve a Sudoku ....................... 8

2.3.2 - Using a Constraint Solver to Solve a Magic Square ............... 8
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3 - Analysis of Results</td>
<td>9</td>
</tr>
<tr>
<td>3.0 - Techniques to Improve Constrained Random Variable Solving</td>
<td>11</td>
</tr>
<tr>
<td>3.1 - Graph-Based Methods</td>
<td>11</td>
</tr>
<tr>
<td>3.2 - Reducing Complexity of Constraints</td>
<td>11</td>
</tr>
<tr>
<td>Conclusions</td>
<td>12</td>
</tr>
<tr>
<td>Recommendations</td>
<td>13</td>
</tr>
<tr>
<td>References</td>
<td>14</td>
</tr>
<tr>
<td>Cited References</td>
<td>14</td>
</tr>
<tr>
<td>General References</td>
<td>15</td>
</tr>
</tbody>
</table>
List of Figures and Tables

Figures

1.2a  An example of multiple constraints: a hard constraint (top) and a soft constraint using the “soft” keyword (bottom) in SystemVerilog 2012

2.1a  A production machine model of constraint solving

2.1.1a A pictorial representation of a decision problem

2.3.1a The Sudoku puzzle solved by the EDA industry-class solvers (unsolved, left; solved, right)

Tables

2.3.1b Data from solving Sudoku puzzles

2.3.2a Data from solving 3rd order magic square
Summary

The ability to produce well-distributed, constrained random values is crucial to the field of ASIC verification. Verifying ASICs by creating specialized directed tests is one method of testing, but then one is presented with the inevitable question: How many tests would adequately verify the design? Moreover, what defines “adequate” verification? In the place of directed testing, functional verification along with coverage-driven verification are used. Coverage-driven verification aims to verify a device by stimulating said device with the broadest amount of inputs as possible. This stimulation is intended to coerce the device under test into as many different states as possible. These different states should ideally activate the device’s full functionality, which leads to the functional verification of the device. These different states are triggered by constrained random stimulus. This stimulus is constrained by the use of boolean functions, which must evaluate to true when that value is randomized. The end goal of constraint solving in this context is to produce input for a DUT or provide configuration values for a test bench. Generally, constraints are applied to data to a DUT through connecting the corresponding signals in the modelled constraint logic to those of the DUT. Input generation with constraints can be considered a process of removing the values that don’t meet the constraints and re-randomizing. This allows the modelling of the process as a constraint satisfaction problem. In general, constraint satisfaction problems have: a set of variables, a non-empty domain for each of these variables, a set of constraints restricting the value of these variables, and optionally, a cost function that measures the quality of the assignments to these variables. The goal of these CSPs is to solve for values that meet all the specified constraints, or to determine that no solution exists to the specified constraints. Depending on the domains being continuous or discrete, and finite or infinite, and depending on the constraints being linear or non-linear, the general CSP definition can be refined to many familiar special cases. Four broad types of constraint satisfaction problems are often encountered in the EDA field: linear programming, propositional satisfiability and automatic test pattern generation, binary decision diagrams, and boolean unification. One issue with constraint satisfaction problems is that they are members of a mathematical set called NP-complete. This essentially means that while the solutions to these problems can be verifiable (NP) in polynomial time, they are not necessarily solvable (P) in polynomial time. An informal comparison of two leading EDA vendors’ tools is then presented. Due to legal concerns, company names are withheld but it is shown that though both tools claim to be the absolute fastest solvers available, one is able to solve a 9x9 Sudoku faster and the other solves an order 3 magic square faster. The demonstration is intended to show that improving constraint solvers is an inherently difficult task, and often improving in one area (speed, memory/CPU usage, or distribution of variables) causes a regression in another. Techniques to improve constrained random variable solving are then outlined: the usage of graph-based methods and reducing the complexity of constraints. While solvers that make use of graph-based constraints are not available from any of the “big three” vendors, there are smaller venture-backed companies that have developed graph-based solvers and other tools for functional verification that make use of graph-based methods. The final recommendation is that care should be taken to write constraints as straightforward as possible, making use of language features whenever possible and breaking large constraint sets up into smaller sets.
Glossary of Terms and Symbols

Accellera Systems Initiative (Accellera)
A standards organization focused mainly on developing electronic design automation and integrated circuit design standards.

Application Specific Integrated Circuit (ASIC)
An integrated circuit designed specifically to perform a certain task or set of tasks.

binary
The base-2 counting system, consisting of combinations of 1s and 0s.

bus
A communications system that transfers data between computer components.

Cadence
Cadence is an American electronic design automation software and engineering services company, specializing in IC, PCB and SOC design tools, and verification.

Constraint Satisfaction Problems (CSPs)
Constraint Satisfaction Problems (CSPs) are mathematical problems that are required to satisfy local constraint conditions.

coverage
A measure used to describe the degree of which a particular metric has been tested.

Device Under Test (DUT)
A device being tested.

Electronic Design Automation (EDA)
A field in computing that focuses on creating software for the design and validation of ASIC and PCB designs.

First In First Out (FIFO)
A software data structure concept. The first item to be “pushed” onto the queue will be the first item “pulled” off the top when accessed.

functional verification
The task of ensuring a logic design used for an integrated circuit performs as expected.

Hardware Design Language (HDL)
A specialized programming language used to design integrated and electronic circuits. Common HDLs are Verilog and VHDL. See “Verilog”.
**Hardware Verification Language (HVL)**
A programming language used to verify circuits designed in hardware design languages (HDLs). Common HVLs are SystemVerilog. See “SystemVerilog”.

**hexadecimal**
Also known as “hex”, this is a base-16 numbering system.

**Mentor Graphics**
An electronic design automation company, producing among other tools such as Questa Advanced Simulator.

**Negated AND or NOT AND (NAND)**
A logic gate used as the basic building block of digital systems due to its ability to implement any boolean function using a combination of them. Often arranged in series and used to store data (NAND flash memory).

**Open Verification Methodology (OVM)**
The verification methodology succeeded by UVM.

**Printed Circuit Board (PCB)**
A non-conductive sheet (usually made of plastic) that physically holds electronic components in place. These components are linked together using conductive pathways printed directly onto the board.

**Questa Advanced Simulator**
A HDL simulator. Also a competitor to Synopsys’ Verilog Compiler Simulator (VCS).

**register**
In computing, a register stores bits of information. These bits can usually be both written and read out of the register, but this is register-dependant (i.e., some registers may only allow external devices read access to certain register, only allowing those registers to be modified by internal influence).

**regression testing**
A method of software testing that aims to discover new bugs after making any change to the environment or the software itself.

**stepping**
A term primarily used by Intel as a sort of revision number for integrated circuits. For example, a new design would be designated A0, with minor and major revisions of the circuit increasing either the number or the letter, respectively.
**Synopsys**
One of the largest companies in the electronic design automation (EDA) industry, Synopsys designs tools for engineers to design and verify electronic designs. See “VCS”.

**SystemVerilog**
An object-oriented programming language, combining both a hardware design language (HDL) and a hardware verification language (HVL). Based on the Verilog HDL.

**test bench**
A virtual software environment to verify a certain product. In the context of this report, the test bench is the verification platform that contains the DUT along with the framework of tests and related software used to verify the design of the DUT.

**Universal Verification Methodology (UVM)**
A standardized methodology from Accellera for verifying integrated circuit designs. Also includes a base class library that allows for the straightforward adoption of its methodologies.

**Verilog Compiler Simulator (VCS)**
A HDL design simulator and functional verification tool designed by Synopsys.

**word**
In the context of this report, a word is a unit comprised of 8 bits.
1.0 - Introduction

The ability to produce well-distributed, constrained random values is crucial to the field of ASIC verification. Generated by constrained random value generators, these values are used to stimulate ASIC designs during their initial pre-silicon phase. If a given ASIC design is anything more than trivial, ensuring all generated values are as random as possible while adhering to all specified constraints can consume a significant portion of simulation time. Clearly, if one could reduce the time random constrained solvers take to resolve all values requested of them, there would be a significant time savings. Further, if the randomized values were guaranteed to be uniformly distributed, fewer differently-seeded simulations would have to be run, saving potentially even more time during testing.

Solving constrained random values is often referred to as a constraint-satisfaction problem (CSP). This report intends to introduce the reader to five different commonly used methods for solving these CSPs, analyze the constrained random number generators in two commonly-used electronic design automation (EDA) software packages, and provide recommendations on how to potentially improve existing verification methods.

1.1 - Intel and Non-Volatile Memory Solutions Group

Intel is known worldwide as a leader in excellence and innovation in the technology and semiconductor sectors. With divisions that focus on computer processors, non-volatile memory, supercomputers, and client software, Intel’s worldwide workforce of over 82,000 employees foster an unique atmosphere of innovation and collaboration [1]. This dedication to innovation is easily verified by the more than 1,600 US patents issued to Intel in 2010, and has seen the US Patent Office issue more than 1,000 patents to Intel a year for the last 20 years [2]. Its Canadian division, Intel of Canada, Ltd. (hereafter “Intel Canada”), maintains a strong presence in Canada with its Vancouver location serving as an important office for ASIC, firmware, and hardware engineering teams in the Non-Volatile Memory Solutions Group (hereafter “NVM Solutions Group”).

The NVM Solutions Group designs and manufactures solid-state drives for desktops, laptops, supercomputers, and data centres. The NVM Solutions Group often designs every major part of any given Intel solid-state drive, from the controller ASIC through the firmware architecture and the PCB layout to the final verification of the drives themselves.

Intel’s solid-state drives (hereafter “SSDs”) are renowned in the semiconductor industry for their high reliability and extremely fast and consistent performance when compared to other vendors [3]. One 2010 report found that the mean failure rate for Intel SSDs was “more than three times [lower] than [other SSD-producing companies]” [4]. The root of this reliability can be theorized to arise from a combination of different factors: NAND sourcing, PCB design and testing, ASIC design and testing, and overall quality control.
1.2 - Constrained Random Verification and Stimulus in ASIC Verification

The verification of a pre-silicon ASIC design is an inherently challenging task. Any given ASIC may have tens of different inputs, hundreds or thousands of different internal states, and millions of potential outputs. Clearly, ensuring the design under test (DUT) has been verified to operate as expected under every possible stimuli is an impossible task. Therefore, constrained random stimulus is used to drive the inputs of the DUT; this stimulus should ideally provide adequate randomness to the DUT so as to verify the DUT’s operation in the widest number of states possible.

1.2.1 - Constrained Random Verification and Coverage-Driven Verification

When it comes time to verify a block-level ASIC design, there is often little discourse on whether or not to primarily pursue constrained random verification as opposed to directed verification. While writing directed test cases to cover certain important features of your design can still lead to bugs being found, the edge cases of the DUT can be difficult or impossible to discover and verify. Using constrained random values and coverage-driven verification, a verification engineer can build a test bench that ensures the stimulation and configuration of the device is as random as possible while ensuring the stimulation stays within such values that keep the DUT in a predictable state.

In coverage-driven verification, there are two types of coverage: code coverage and functional coverage. Code coverage measures whether or not the code in the design of the DUT has all been run: if it has not, then the code that did not run - the “hole” in the coverage - must be examined. There may be a bug in the DUT, or the constrained random verification has not so far been able to coax that section of the design into operation. Code coverage is very useful in the sense that if a section of code has not been run in a test, it has clearly not been verified. Further to that, in many simulators, code coverage can be enabled by simply enabling a code coverage option before a given test is run, making it very simple to collect. However, high code coverage does not necessarily mean the design of the DUT is verified, as high code coverage does not imply the code in the DUT is performing its task correctly.

In order to ensure the DUT is producing the correct values, there is functional coverage. Functional coverage is used to record the values and states of the DUT for later analysis by a verification engineer or commercial analysis program. The process of defining the functional coverage of a device is not an automatic one unlike code coverage. Defining functional coverage requires an engineer to manually define what values must be input and output of the DUT. For example, functional coverage could measure whether or not every type of instruction was created and written to an external interface by the DUT, or whether every possible value of a pointer was seen when reading from a FIFO stack.
As more and more constrained test runs of constrained random verification take place, both functional coverage and code coverage of the DUT will increase. Bugs are found and fixed as edge cases occur, and the test bench is revised through the removal, addition, and adjustment of constraints and the addition of both constrained random tests and directed tests so as to improve coverage.

1.2.2 - Constrained Random Stimulus

The generation of constrained random stimulus in ASIC verification is of the utmost importance. These values must be generated using a method that is relatively efficient, fast, uniformly distributed, and able to adhere to multiple complex constraints. The constraints are user-created, and used to ensure the values generated fit within certain parameters. For example, in certain cases a verification engineer may not want to generate error conditions in her design under test (DUT). To do that, she could set up boolean constraints on her data using language-specific constructs to indicate that the constraint solver must not violate.

Depending on the language, these constraints can be hard (inviolable) or soft (violable only when the random number generator is not able to satisfy it). As seen in Figure 1.2a, the topmost constraint will ensure cost is between the integer values of 2000 and 1000. The bottommost constraint will ensure that when cost is randomly generated, it takes on a value less than 1600. This topmost constraint will always be satisfied, but the bottommost one may not be if another constraint that depends on cost requires a value greater or equal to 1600.

```systemverilog
constraint flight_cost_c {
    cost inside {[2000:1000]};
    soft cost < 1600;
}
```

*Figure 1.2a: An example of multiple constraints: a hard constraint (top) and a soft constraint using the “soft” keyword (bottom) in SystemVerilog 2012*
2.0 - Analysis of Solving Constrained Random Variables

The most closely-guarded area of any advanced EDA tool is the solver - the algorithm (or algorithms) that solve the constraints generated by the user. The detailed, internal workings of the solvers offered by any of the “big three” EDA software vendors - Cadence, Mentor, and Synopsys - are closely-guarded trade secrets. Therefore, this section will analyze the types of constraint satisfaction problems that are commonly encountered in the EDA field as well as demonstrating solving non-trivial constraints with EDA software from two popular vendors.

2.1 - Constraint Solving

In the constrained random verification field, the end goal of constraint solving is to provide the input for a DUT or provide configuration values for a test bench. Generally, constraints are applied to a DUT through connecting the corresponding signals in the modelled constraint logic to those of the DUT. The result is called the production machine of the constraints and the DUT. The direction of data flow divides the signals into two groups: the signals that leave the DUT are called the state signals, and the signals that flow into the DUT are called inputs. State signals are values set by the DUT during simulation. The goal of constraint solving is to calculate values for the input based on the state values from the DUT.

Input generation with constraints can be considered a process of removing values that don’t fit the constraints and re-randomizing until the generated input meets the given constraints, as in the following steps:

1. **Simulation**: Assuming this is not the beginning of the simulation, apply the current constraint solution to the DUT, simulate the results, and update the state of the production machine. If this is the beginning of the simulation, this step is skipped.
2. **Constraint solving**: In this step, the current state of the production machine is propagated to the constraints. The constraints are then solved so that all constrained values satisfy all their given constraints simultaneously.
3. **Randomization**: There are two possible outcomes after solving the constraints:
   A. No solution can be found. This results in a so-called "dead-end state" and means that simulation must be aborted once this happens.
   B. If there is at least one solution, one set of solutions is chosen randomly. The values are commonly chosen at this stage so as to result in a uniform distribution after a statistically significant number of test runs, but this is usually left to the discretion of the user.

![Figure 2.1a: A production machine model of constraint solving [5]](image)
Following this process, constrained random generation is reduced to a generic constraint satisfaction problem of boolean formulas. The various types of constraint satisfaction problems encountered in EDA applications will be discussed in section 2.2.

2.1.1 - P, NP, NP-Hard, and NP-Complete

As mentioned in the introduction of this report, solving a large set of constraints for suitable values can be time-consuming. Generally, the more constraints one applies to a set of values, the longer the solver must take to try to solve them. The fact that the time to solve for a set of values that satisfy a growing set of constraints increases in such a way can be related to one of the most important and unsolved problems in computer science: the \textbf{P} versus \textbf{NP} problem.

The terms \textbf{P}, \textbf{NP}, \textbf{NP-hard}, and \textbf{NP}-complete all denote different complexity classes. A complexity class can be thought of as a group of problems that all have similar computational requirements to solve. Problems that can be solved in polynomial time ($T(n) = O(n^k)$) are assigned to the complexity class \textbf{P}. Problems that lie in \textbf{P} are considered to be quickly solvable on computational devices [6]. These problems are decision problems, and so are answered with a “yes” or “no” answer. For example, testing whether or not a given natural number is a prime number by testing every possible non-trivial factor of it is considered a decision problem.

If their answer is “yes”, problems that lie in the \textbf{NP} complexity class must be capable of being quickly verified (i.e., verifiable in polynomial time) that their answer is indeed “yes”. Unlike problems in \textbf{P}, \textbf{NP} problems may take a non-polynomial amount of time to solve by a regular (deterministic) computer but can be solved in polynomial time by a nondeterministic machine. In this context, nondeterministic means that a different run of the problem would yield different results. This is the opposite of a conventional computer - a deterministic machine - which, given a particular input, will always produce the same output, with the underlying algorithm always passing through the same sequence.

Commonly, a mathematical problem can be solved by reducing it to a different problem. For example, problem B could be reduced to problem A if given a solution to problem A a solution could be constructed to problem B in polynomial time. Similarly, if a problem is \textbf{NP}-hard, this means any problem in \textbf{NP} can be reduced to that \textbf{NP}-hard problem. An \textbf{NP}-complete problem is both \textbf{NP} and \textbf{NP}-hard.

The \textbf{P} versus \textbf{NP} problem asks whether some algorithm exists that allows problems that are quickly verifiable (\textbf{NP}) can also be quickly solvable (\textbf{P}). The subset sum problem is one of many examples of where an answer to the \textbf{P} versus \textbf{NP} problem would dramatically change the speed of solving the problem. The subset sum problem asks, given a set of integer values, does some nonempty subset of them sum to zero? For example, in the set of numbers \{1, -30, -7, -8, 10, -2, 5\}...
it is polynomially easy to verify that \{-8, 10, -2\} add up to zero, but not necessarily easy to find a subset in polynomial time. If \(P = NP\), then problems like the subset sum problem could also be solved in polynomial time. The inverse - \(P \neq NP\) - implies that NP problems like the subset sum problem can only be verified in polynomial time and there exists no such algorithm to solve it in polynomial time.

### 2.2 - Types of Constraint Satisfaction Problems

As mentioned in section 2.1, sets of constraints can be modelled and solved as a generic constraint satisfaction problem of boolean formulas, which are analogous to decision problems as mentioned in section 2.1.1. Generally, constraint satisfaction problems fit the following criteria: they posses a set of variables, a non-empty domain for each of these variables, and a set of constraints restricting the value of these variables. Optionally, there may also be a cost function applied to some or all of the variables in the CSP, which are used to measure the “quality” or statistical rank of a given value.

Ultimately, the goal of these CSPs is to solve them so that their values meet all the specified constraints. In some cases, there will be no solution to the constraints given, which can occur in cases of over-constrained variables, multiple dependant variables on constraints, and other similar problems. These over-constraints are almost invariably programming errors. For example, a constraint could state that \(x\) must be greater than 9, whereas another in the same CSP could state that \(x\) must be equal to 9.

#### 2.2.1 - Integer Linear Programming and Binary Integer Programming

Linear programming is the most commonly-applied form of constraint optimization. These problems are solved by using a linear set of constraints over the set of real numbers, which are then optimized to a cost function.

Integer linear programming is a special case of linear programming where all unknowns are required to be integers, and similarly binary integer programming is the case where the unknowns must be either 0 or 1. Both of these special cases, unlike linear programming, are NP-hard. Some common algorithms to solve these problems are the cutting-plane method, branch and bound, branch and cut, and branch and price.

#### 2.2.2 - Boolean Satisfiability and Automatic Test Pattern Generation

Both boolean satisfiability (SAT) and automatic test pattern generation (ATPG) are methods for solving propositional (e.x., \((P \text{ AND NOT } Q) \text{ IMPLIES } (P \text{ OR } Q))\) formulas. The major difference between the two is that SAT uses a set of clauses that are functionally equivalent to a gate-level design, whereas ATPG uses the (often higher-level) design itself.
It is important to mention that both of these methods are NP-complete, meaning that there are cases where possible solutions exist but both methods will take an unacceptably long time to find them.

2.2.3 - Binary Decision Diagrams

Binary decision diagrams (BDDs) are graphical representations of boolean functions. When given logically equivalent functions (and fixed variable ordering) in a given graph, the BDDs of those logically equivalent functions are isomorphic. By creating the BDD of a boolean function, every solution to the constraint (and logically similar constraints) is found.

2.2.4 - Boolean Unification

Boolean constraints can be solved using boolean unification (BU) [8]. BUs are similar to BDDs in the sense that once a solution is found, the solution represents all the possible solutions of the constraints. In the case of BUs, the solution can essentially be viewed as a transformation of the boolean constraints into a vector function.

2.2.6 - Analysis of Approaches Presented

In the case of constrained random number generation, ILP, BIP, SAT, and ATPG are “online” approaches, as the solutions to those types of CSPs are found during simulation. Since decision diagrams are often solved before simulation so as to allow for statistical distribution of results, both BDD and BU are classified as “offline” approaches. The practical difference between online and offline solving is that offline approaches can be solved in exponential time in the best case, whereas online approaches can be solved in exponential time in the worst case.

2.3 - Informal Comparisons of Constrained Random Variable Solvers

As mentioned in introduction to section 2.2, the solver - the part of the EDA simulation environment that solves constraints - is a very closely-guarded secret in the EDA developer community. Solvers are considered so sensitive that many EDA companies forbid head-to-head comparisons of solvers and publication of benchmarking results. Therefore, the data presented in the following sections will not be presented with product names or versions; rather, the solvers will be arbitrarily labeled solver A and solver B.

Both solvers were benchmarked with a well-known puzzle that relies on constraint satisfaction in order to solve: Sudoku. A magic square calculation was also benchmarked. The code used to generate these results was found on a public GitHib repository and will not be reproduced in full in this report. See [9].
### 2.3.1 - Using a Constraint Solver to Solve a Sudoku

In Sudoku, one is presented with a partially filled cube of numbers as seen in figure 2.3.1a. In order to solve the puzzle, the 9x9 number grid must be filled with digits so that each column, row, and each of the nine 3x3 sub-grids contain all the numbers from 1 to 9. When the puzzles are solved, it can be seen that the same single integer may not appear twice in the same 9x9 number grid row or column or in any of the nine 3x3 sub-grids.

The problem of solving a Sudoku can be easily transferred to an EDA solver. As one can see from the results in table 2.3.1b, the time taken to solve these problems is on the order of seconds to milliseconds.

![Figure 2.3.1a: The Sudoku puzzle solved by the EDA industry-class solvers (unsolved, left; solved, right)]

<table>
<thead>
<tr>
<th>Solver A (user time, seconds)</th>
<th>Solver A (system time, seconds)</th>
<th>Solver A (CPU share, %)</th>
<th>Solver B (user time, seconds)</th>
<th>Solver B (system time, seconds)</th>
<th>Solver B (CPU share, %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.996</td>
<td>0.072</td>
<td>22.8</td>
<td>0.320</td>
<td>0.032</td>
<td>11.3</td>
</tr>
<tr>
<td>1.104</td>
<td>0.048</td>
<td>23.7</td>
<td>0.336</td>
<td>0.056</td>
<td>15.0</td>
</tr>
<tr>
<td>0.948</td>
<td>0.044</td>
<td>20.3</td>
<td>0.356</td>
<td>0.060</td>
<td>13.8</td>
</tr>
<tr>
<td>0.956</td>
<td>0.060</td>
<td>20.3</td>
<td>0.332</td>
<td>0.064</td>
<td>13.6</td>
</tr>
<tr>
<td>0.876</td>
<td>0.048</td>
<td>19.9</td>
<td>0.312</td>
<td>0.060</td>
<td>12.5</td>
</tr>
<tr>
<td>0.976</td>
<td>0.054</td>
<td>21.4</td>
<td>0.331</td>
<td>0.054</td>
<td>13.2</td>
</tr>
</tbody>
</table>

### 2.3.2 - Using a Constraint Solver to Solve a Magic Square

Magic cubes are a mathematical concept where the numbers in each row, in each column, and in the forward and backward main diagonals all add up to the same number. This constant (called the “magic sum”) can be calculated using the following equation, where \( n \) is the order of the square (the number of rows or columns the square has):
\[ M = \frac{n(n^2+1)}{2}. \]

For example, in the case where \( n = 3 \), \( M = 15 \). This was the case used to generate the data in table 2.3.2a.

### Table 2.3.2a

<table>
<thead>
<tr>
<th>Solver A (user time, seconds)</th>
<th>Solver A (system time, seconds)</th>
<th>Solver A (CPU share, %)</th>
<th>Solver B (user time, seconds)</th>
<th>Solver B (system time, seconds)</th>
<th>Solver B (CPU share, %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.244</td>
<td>0.032</td>
<td>6.3</td>
<td>0.296</td>
<td>0.060</td>
<td>11.8</td>
</tr>
<tr>
<td>0.236</td>
<td>0.040</td>
<td>6.7</td>
<td>0.336</td>
<td>0.076</td>
<td>14.8</td>
</tr>
<tr>
<td>0.296</td>
<td>0.044</td>
<td>8.1</td>
<td>0.316</td>
<td>0.056</td>
<td>13.3</td>
</tr>
<tr>
<td>0.256</td>
<td>0.044</td>
<td>7.4</td>
<td>0.336</td>
<td>0.056</td>
<td>12.7</td>
</tr>
<tr>
<td>0.264</td>
<td>0.052</td>
<td>8.0</td>
<td>0.328</td>
<td>0.052</td>
<td>13.7</td>
</tr>
<tr>
<td>0.259</td>
<td>0.042</td>
<td>7.3</td>
<td>0.322</td>
<td>0.060</td>
<td>13.3</td>
</tr>
</tbody>
</table>

### 2.3.3 - Analysis of Results

The results in tables 2.3.1b and 2.3.2a were generated using the Linux `time` command. This command shows by default, among other things, the user time, system time, and the CPU share of the process. Five concurrent runs of each EDA vendor’s simulator (i.e., simulation phase only) were recorded and averaged in both tables.

Comparing the results of the Sudoku puzzle between solver A and solver B in table 2.3.1b, and the magic square in table 2.3.2a, it can be seen that there are distinct differences in solver technology and optimization. In the case of the Sudoku puzzle, solver A took an average of 0.976 seconds of user time and 0.054 seconds of system time, with 21.4% CPU utilization. Solver B took an average of 0.331 seconds of user time and 0.054 seconds of system time, with 13.2% CPU utilization. Considering how the results of each remained relatively uniform throughout the successive test runs (discounting effects like memory cacheing or prediction algorithms), one might draw the conclusion that solver B was the fastest solver compared. However, the results of the magic square complicate solver B’s victory. When solving a magic square, solver A manages to edge out solver B, with a 0.018 second difference in solver A’s favour for system time, and a 0.063 second difference between user time (again in solver A’s favour), it is clear to see that designing and optimizing a solver to be fast and efficient is not an easy feat.
Comparing the results of the magic square solving to those of the Sudoku puzzle demonstrates the tradeoffs EDA vendors make when designing combinations of approaches to the various constraint satisfaction problems to implement in their solvers. In trivial cases such as these puzzles the difference between these two solvers can be imperceptible, but in cases where a DUT is being tested, hundreds of thousands of seeds of a single test can be run over many months. Assuming even a difference on the order of seconds between solvers for constraint sets an order of magnitude larger than those tested above (a reasonable estimate given the size and scale of constrained random test-benches), when on average 5000 tests a week for a year are run there is clearly a need to save every second possible.
3.0 - Techniques to Improve Constrained Random Variable Solving

As the field of EDA is dominated by three major vendors (Synopsys, Mentor, and Cadence), there is still a fair degree of competition between those vendors as well as smaller entities trying to break into the market. As one could imagine, if there were any “easy” improvements to make to algorithms they would be made as quickly as possible to gain a competitive edge. There are, however, some techniques to improve both the speed of solving and the distribution of results.

3.1 - Graph-Based Methods

There are smaller EDA companies that have focused their solver efforts on graph-based methods of solutions to constrained random variables. These graph-based methods look for ways to transform the given constraints into a geometric problem. Once this has been done, all possible solutions to the constraints are easily found, as they will only lie in a certain area of the graph.

Trek from Breker Verification Systems is such a solver. The graph-based solving method also allows Trek to generate tests for DUTs and close “coverage holes”, areas of functional coverage that are missing values [10]. This often indicates the edge cases of a design are not being hit, sometimes because of bugs in the design or verification environment, but usually because of over-constraining values. Because every solution to the random values in the test-bench has been calculated when using graph-based methods, these coverage holes are easy to fill because the values that can fill such a coverage hole are already known. If these values are not known or the coverage is otherwise unreachable, then it is likely that there is a bug in either the design or the test-bench.

3.2 - Reducing Complexity of Constraints

A good practice when programming is to keep whatever code that one writes as simple and clear as possible; the same extends to writing constraints. The author of this report has anecdotally experienced the fact that rewriting particularly complex constraints (often by using a more efficient algorithm or breaking larger constraint sets into smaller ones) can save 30% or more time during randomization.

This phenomenon has also been documented by Gregory Tang and Rajat Bahl of AMD and Alex Wakefield and Padmaraj Ramachandran of Synopsys. Their article in the EDA magazine Design & Reuse showed that “…by partitioning the constraints hierarchically into smaller groups of opcodes, the memory requirements were drastically reduced, which increased performance…” [11]. Using the default VCS 2009.12 solver (RACE, a word-level solver based on the ATPG method in section 2.2.2) with the new constraints, the speed of constraint solving improved by 4x [12]. Using the alternate VCS 2009.12 solver (BDD, see section 2.2.3), this constraint refactoring improved randomization times by 2x and memory usage by more than 2x.
Conclusions

This report aimed to present a brief overview of solver technology, constraint solving, and techniques to improve constraint solving through experience gained in an eight-month work term at Intel in ASIC verification. As the author of this report hoped to make clear, the field of EDA solving is a competitive and cutthroat world and improving solver technology is not a simple task.

The “big three” EDA vendors all assert that their EDA tools are the fastest (especially when it comes to random number generators and constraint solvers). As one can see from the comparisons in section 2.3.3 though, the title of “fastest solver” depends very much on the problem being solved. There is a tradeoff that solver architects must balance: the amount of memory consumed vs. the speed of the solver vs. the distribution of the results, and every EDA vendor has tackled these issues in different (and secret) ways. Moreover, attaining a balance between memory and processor utilization, speed, and uniformity of results is very difficult.

The big three are continually making improvements to their solvers: as mentioned in section 3.2, Synopsys’ VCS added ILP as a method for the solving of small constraints in 2010 to join the existing RACE (ATPG) and BDD algorithms. It is often mentioned in the EDA community that all the straightforward optimizations have already been done when it comes to improving solver technology, but there are a number of small, venture-backed firms like Breker Verification (section 3.1) that have introduced new methods catered specifically to engineers in the EDA community with tools to ease coverage closure and ensure a very uniform distribution of variables.
Recommendations

As mentioned many times in this report, designing a solver is a non-trivial task. Balancing processor and memory utilization, speed, and uniformity of results is very difficult. Improving the underlying algorithms of the solvers without a significant overhaul is not likely at this point. Of course, a major improvement to solvers would arrive if \( P \) was found to equal \( NP \), as this would allow solutions to constraint satisfaction problems to be found in polynomial time. Unfortunately, the likelihood of that happening is very low; the inverse (\( P \neq NP \)) is suspected by many, but has not been conclusively proved. Instead, it is recommended that alternate technologies (like graph-based solvers) be investigated for use in verification, as there is evidence to support that these technologies can dramatically enhance coverage closure times, which helps meet the end goal of constrained random verification of ASICs.

However, users of EDA solvers from the big three vendors do not necessarily have to wait for improvements to come by way of algorithmic updates or landmark events in mathematics, as there are methods (presented in section 3.2 of this report) that can improve constraint solving speed by a significant margin. Reducing the complexity of constraints by breaking constraints into smaller classes can yield a significant improvement in runtime. Making use of language-specific constraint features, like the “soft” constraint in SystemVerilog 2012, can aid in simplifying constraints further and reduce the need to constrain variables multiple times in through different classes. It is recommended that steps are taken to simplify constraints whenever possible, as significant performance savings could result from simplified constraints. Note that simplifying constraints will likely not improve distributions of random variables; this is only possible with either saving every random value from every test case or using a graph-based solving method.
Cited References


General References


