Performance of a SystemVerilog Sudoku Solver with Synopsys VCS

Jeremy Ridgeway

Avago Technologies, Ltd.
Fort Collins, CO, USA

www.avagotech.com

ABSTRACT

Constrained random verification relies on efficient generation of random values according to constraints provided. As constraint solver metrics are not easily determined, usually solver efficiency can only be measured per-project and late in the verification cycle. In this paper we dissect several SystemVerilog-based Sudoku puzzle solvers and compare their efficiency with the VCS constraint solver. Further, we compare efficiency between constraints applied over object instance hierarchies (game board is object oriented) versus flat constraints (game board is fully contained within a single class). Finally, we compare both approaches with several optimizations in the Sudoku solver. The common Sudoku game board is a 9x9 grid yielding approximately 2,349 constraint clauses to solve. We show that VCS can solve grid sizes up to 49x49 with 357,749 clauses. While each clause is a simple inequality, the size of the constraint formula to solve and its structure provides valuable feedback on the solver's efficiency.

Table of Contents

1. Introduction 4

2. SystemVerilog Constraints 4
   - Constructing a Boolean Formula 5
   - Solving the Boolean Formula 6

3. Sudoku 7
   - Game Board Rules 7
   - Dissecting Sudoku Constraints 7
   - Sudoku vs. the Real World 11
   - Methodology 11
   - Uniqueness Constraint 12

4. Object Oriented Game Board (OOP) 13
   - OOP Optimization 1: if-else constraint in place of implication 16
   - OOP Optimization 2: constant or parameter in cell constraint 18
   - OOP Optimization 3: combine optimization 1 & 2 21
   - OOP Optimization 4: reduction of array duplicates 23

5. Flat Game board (FLAT) 27
   - FLAT Optimization 1: reorder array inequalities 28
   - FLAT Optimization 2: reduction of array duplicates 29
   - FLAT Optimization 3: no inequality redundancies 29
   - FLAT Test Bench Results 29

6. Conclusions 32

References 33

Table of Figures

Figure 1: Sudoku game board model: (A) a single cell, (B) a Latin-square, and (C) the full grid with Latin-squares (blue), rows (red), and columns (green). 7

Figure 2: A 2x2 Latin-square Sudoku game board (4x4 cells total). 7

Figure 3: VCS-2014 outperforms VCS-2011 in nearly all cases. 15

Figure 4: Solving Times for OOP Test Bench using if-else construct. 18

Figure 5: Solving Times for OOP Test Bench with constant `m_max` class member. 20

Figure 6: Solving Times for OOP Test Bench with parameter `m_max`. 21

Figure 7: Solving Times for OOP Test Bench using optimizations 1 and 2, constant class member. 22

Figure 8: Solving Times for OOP Test Bench using optimizations 1 and 2, parameter. 22

Figure 9: Solving Times for OOP Test Bench comparing storage types, parameter vs constant. 23

Figure 10: Solving Times for OOP Test Bench with reduced inequalities, variable class member. 25

Figure 11: Solving Times for OOP Test Bench with reduced inequalities, constant class member. 26
Figure 12: Solving Times for OOP Test Bench with reduced inequalities, constant parameter. . 26
Figure 13: Solving times comparing Basic OOP TB vs Flat TB with optimization 3..... 31

Table of Tables

Table 1: Derivation contiguous range term to literals. 5
Table 2: Unique predicates and literals in cell class formula. 5
Table 3: Derivation of cell row constraint. 9
Table 4: Sudoku Board Sizes Employed in Testing 11
Table 5: Number of Clauses in Basic OOP Test Bench. 14
Table 6: Solving Time Summary for Basic OOP Test Bench. 15
Table 7: Expected Number of Classes in OOP Test Bench using if-else Construct..... 16
Table 8: Solving Time Summary for OOP Test Bench using if-else Construct. 17
Table 9: Solving Time Summary for OOP Test Bench against different class storage types (optimization 2)..... 19
Table 10: Solving Time Summary for OOP Test Bench with Optimizations 1 and 2..... 21
Table 11: Expected Number of Clauses for Array Reduction OOP Test Bench. 24
Table 12: Solving Time Summary for Array Reduction OOP Test Bench. 24
Table 13: Clause Count Generated by FLAT Test Bench Generation Script. 29
Table 14: Solving Time Summary for all FLAT Test Bench Techniques. 30
Table 15: Solving Time Summary VCS versus Mathsat5 and Yices. 31

1. Introduction

Constrained random verification is dependent on efficient generation of random values according to constraints provided. Poorly designed constraints or an inefficient constraint solver can prove to be a bottleneck for simulation at the block level and, especially, at the sub-system or system-on-a-chip (SoC) level. Yet, solving provided constraints effectively is as much an art as it is a science. Thus, constraint solvers can vary widely in their efficiency and performance even over successive iterations from the same supplier.

In this paper we dissect the Sudoku game board as a non-proprietary approach for some measure of efficiency. Sudoku solving lends itself well to a constraint-based approach [1], [2]. The game board follows strict rules that are easy to parameterize and scale. Large puzzles provide significant challenges for satisfiability (SAT) solvers [3], [4]. As the game board size increases the number of conjunctive normal form (CNF) clauses needing resolution explodes. As such, while the game board is simple to visualize, reducing the formula to only necessary clauses is essential.

We present Sudoku constraint solving in the context of a SystemVerilog test bench. We utilize the Sudoku SAT formula CNF clausal explosion to isolate small changes in the constraint composition and their effect on the solver. We show that the Synopsys VCS constraint solver is adept at handling all our constraint structures. This efficiency is against the norm, as we show with time metrics from two public domain SAT solvers, mathsat5 [5] and yices [6].

In order to discuss the mechanics of solving Sudoku constraints, we must first introduce some terminology for Boolean satisifiability, in section 2. Then, we dissect the Sudoku game board itself in the simplest forms in section 3. From there, we explore several approaches for designing a Sudoku solver in SystemVerilog, sections 4 and 5. For each tweak of the constraint we discuss time metrics and, when feasible, comparisons to a "base" approach. We conclude in section 6.

2. SystemVerilog Constraints

SystemVerilog specifies that all constraints on a given random variable must be considered simultaneously in a conjunctive fashion. For example, consider the valid constraint in the following class:

```
class cell;
    rand int val;
    constraint valid { val inside {[1:9]}; }
endclass
```

Here, the random class member, `val`, is constrained to the range between one and nine. When an instance of the `cell` class is randomized, the constraint solver performs a composition and pre-processing step to construct a Boolean formula to solve, simplifies that to CNF form, then employs SAT and satisfiability modulo theory (SMT) techniques to solve.

```
module tb;
    cell c0;
    initial begin
        c0 = new;
        c0.randomize(); // solve applicable constraints on val
    end
endmodule
```

Constructing a Boolean Formula

Constraint blocks that are applicable at randomization time must be determined prior to propositional (Boolean) formula construction. Then, the formula is transformed into conjunctive normal form (CNF). The general CNF structure is given in (1).

$$\phi = \bigwedge_{i=1}^{L} \bigvee_{j_i=1}^{K_j} l_{j_i} \quad \textbf{(1)}$$

A *predicate*, p, is some Boolean term, a value. A *literal*, l, represents a predicate in either its positive, p, or negative $!p$, form. A literal is said to be negative when representing $!p$, otherwise the literal is positive. In the equation above, l_{j_i}, is a single Boolean literal. A disjunction of literals (logical OR), $c_i = \bigvee_{j_i} l_{j_i}$, is a single Boolean *clause*. A propositional formula, ϕ, is the conjunction (logical AND) of all its clauses, $\bigwedge_i c_i$. A CNF formula is essentially a formula in product of sums Boolean logic form.

For example, the `valid` constraint block in the `cell` class, above, contains two clauses, as shown in Table 1.

Table 1: Derivation contiguous range term to literals.

`val` inside [1:9]	$\Longrightarrow$	(1 <= `val` <= 9)	
	$\Longrightarrow$	(1 <= `val`) && (`val` <= 9)	Distribution
	$\Longrightarrow$	!(!(1 <= `val`) || !(`val` <= 9))	De Morgan's law
	$\Longrightarrow$	!(1 > `val`) && !(`val` > 9)).	De Morgan's law

In the general CNF formula, clauses are the logical OR of multiple literals. It is possible for a clause to resolve to **true** when only a subset of literals are assigned a Boolean value. Conversely, when exactly one literal exists then that clause is considered a *unit clause*. The single literal must resolve to **true** in order for the clause to resolve to **true**.

Table 2: Unique predicates and literals in cell class formula.

$!\,l_0 \leftrightarrow !\,p_0$	$p_0 \leftrightarrow$ (1 > `val`)
$!\,l_1 \leftrightarrow !\,p_1$	$p_1 \leftrightarrow$ (`val` > 9)

The Boolean formula for `val` in the class cell consists of two clauses, $\phi = c_0$ && c_1. Each clause contains a single negative literal, $c_0 \leftrightarrow !\, l_0$ and $c_1 \leftrightarrow !\, l_1$, as shown in Table 2.

Solving the Boolean Formula

The SystemVerilog simulator's constraint solver can solve a propositional formula by assigning a Boolean value to all literals such that the formula itself is *satisfied* (evaluates to **true**) [7]. A *model* is the complete set of literal assignments in the formula such that the formula is satisfied. The solver must successively try literal assignments to determine if a model is possible. If no model can be found, then the formula is *unsatisfiable*. For example, if the cell class were extended with another constraint such that:

```
class invalid_cell extends cell;
    constraint invalid_c { val inside [10:100]; }
endclass
```

When randomizing an instance of `invalid_cell`, the constraint solver would compose and attempt to solve the following propositional formula:

((1 <= `val`) && (`val` <= 9)) && ((10 <= `val`) && (`val` <= 100)).

The clauses in this formula are inconsistent because they direct `val` to hold a value that occurs in both mutually exclusive ranges simultaneously. Therefore, the formula is unsatisfiable. At run-time the simulator provides detailed information in a constraint inconsistency report [8].

Generally, the constraint solver is constructed of two distinct layers: the Boolean layer (satisfiability layer or SAT) and the theory-specific layer (satisfiability modulo theory layer or SMT) [7]. The SAT layer is responsible for finding a feasible assignment on all *literals* in the formula while the SMT layer is responsible for finding a feasible assignment on all *variables* represented by the literals and based on the literal Boolean value. Returning the cell class, the simplified Boolean formula to solve is:

$\phi =$ c_0 && $c_1 = (!\, l_0)$ && $(!\, l_1)$ = !(1 > `val`) && !(`val` > 9).

The SAT layer assigns each predicate to **false**, meaning the literals l_0 and l_1 are **true**, while the SMT layer determines a value to assign `val` such that the predicate Boolean value holds, resulting in a satisfied formula. The theory of linear arithmetic solver must choose val in the range of one to nine in order for the predicate assignments to hold. For example, if val = 5:

$$\begin{aligned} \texttt{val} \leftrightarrow 5 \implies & !(1 > 5) \;\&\&\; !(5 > 9) \\ \implies & !(\textbf{false}) \;\&\&\; !(\textbf{false}) \\ \implies & \textbf{true}. \end{aligned}$$

Iteration between SAT and SMT may occur if predicate/literal assignments are ambiguous without additional theory-specific information.

3. Sudoku

In this section the Sudoku game board is described and some questions posed regarding SystemVerilog constraint composition and solver efficiency.

Game Board Rules

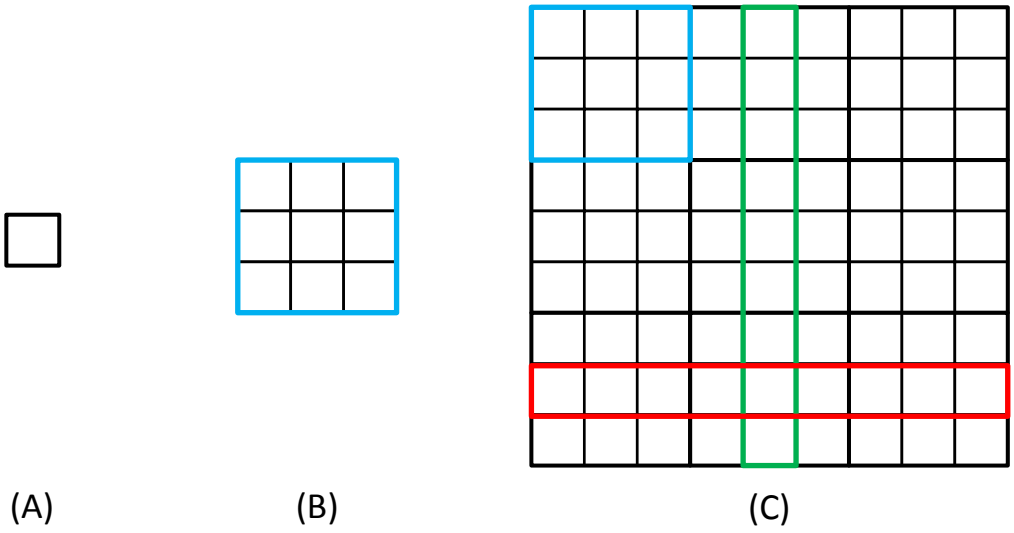

Figure 1: Sudoku game board model: (A) a single cell, (B) a Latin-square, and (C) the full grid with Latin-squares (blue), rows (red), and columns (green).

A Latin-square is an $n \times n$ array of cells populated with n^2 different symbols each occurring exactly once in every row and column [9]. The blue square in Figure 1-(B) and again in Figure 1-(C) is one Latin-square. In practice, the set of integer numbers is commonly used as the symbols for cells. A Sudoku game board is a $n \times n$ Latin-square of $n \times n$ Latin-squares, as in Figure 1-(C). Each of n different symbols occur exactly once within each row and column and Latin-square of the game board [10].

Dissecting Sudoku Constraints

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

Figure 2: A 2x2 Latin-square Sudoku game board (4x4 cells total).

Consider a 2×2 Latin-square Sudoku game. Each Latin-square contains 4 cells total, while the entire game board contains 16 cells. Each cell in Figure 2 is labelled according to row and column position: $C_{\text{row cell}}$. The set of cells { C_{00}, C_{01}, C_{10}, C_{11} } is one Latin-square.

There are four sets of constraints for the game board (cell, row, column, square). First, each cell must constrain its value to the valid range. In general, for an $n \times n$ Latin-square Sudoku game, the valid range for any one cell is constrained to:

$$\text{val } \textbf{inside } \{ [\, 1 : n^2 \,] \}. \qquad \textbf{(2)}$$

The total number of range constraints on all cells is equivalent to the number of cells on the Sudoku game board: $n^2 * n^2 = n^4$ constraints. From the derivation in Table 1, equation (2) simplifies to two clauses in conjunctive-normal form. Therefore, $2 * n^4$ unit clauses are required to properly constrain the range for all cells.

Question 1: What is performance impact when n is a number versus n is a variable?

SystemVerilog is an object-oriented language [11]. We can describe n as a class data member, a local function variable, or simply as a constant value (`` `define ``). In section 4, page 13, we test the effect of the storage class of the range constraint maximum on the solver for an object-oriented game board.

For the remaining set of constraints, the value in a cell must be unique in each of three arrays:

1. Row
2. Column
3. Latin-square

In general, for an $n \times n$ Latin-square Sudoku game, the constraints for each cell within the row set of constraints is:

```
for(int row = 0; row < n^2; row += 1)
    for(int c = 0; c < n^2; c += 1)
        for(int i = 0; i < n^2; i += 1)
            c != i -> c_row_c.val != c_row_i.val;
```
(3)

In a Sudoku game board, there are n^2 rows. For each row, there are n^2 cells (i.e., the number of columns is equivalent to the number of rows). Suppose each cell is modelled as an instance of the cell class with randomized data member, `val`. Latin-square and Sudoku rules state that `val` must be unique within the row. In other words, a single cell's `val` cannot be equivalent to any other cell's `val` in that row. Thus, a set of inequalities must be built to ensure the cell has a unique value in the row.

The outermost loop (`int row`) in (3) considers one row at a time. The next loop (`int c`) considers one cell at a time, the c^{th} location within the row. The innermost loop (`int i`)

compares the cell currently under examination, `c`, with the `i`th cell in the row. Either the current cell is the `i`th cell (`c == i`) or we constrain such that the `i`th cell's value is not the same as the current cell's value.

For a 2×2 Latin-square Sudoku game, the following *complete set* of inequalities should be composed at randomization-time for row 0:

$$\begin{aligned} &C_{00}.val != C_{01}.val \;\&\&\; C_{00}.val != C_{02}.val \;\&\&\; C_{00}.val != C_{03}.val \;\&\& \\ &C_{01}.val != C_{00}.val \;\&\&\; C_{01}.val != C_{02}.val \;\&\&\; C_{01}.val != C_{03}.val \;\&\& \\ &C_{02}.val != C_{00}.val \;\&\&\; C_{02}.val != C_{01}.val \;\&\&\; C_{02}.val != C_{03}.val \;\&\& \\ &C_{03}.val != C_{00}.val \;\&\&\; C_{03}.val != C_{01}.val \;\&\&\; C_{02}.val != C_{03}.val. \end{aligned} \quad \textbf{(4)}$$

In general, the number of inequalities in a complete set per row is $(n^2 - 1) * n^2$. No inequality should be composed for the current cell versus the current cell in each row, thus the minus 1. The total number of inequalities in the complete set for all rows is: $((n^2 - 1) * n^2) * n^2$. For a Latin-square size of 2×2, $n = 2$ and there are $((2^2 - 1) * 2^2) * 2^2 = 48$ row inequalities in the complete set for all rows of the Sudoku game board.

Ideally, each inequality would be a unit clause in the formula for the constraint solver to solve. However, according to the constraint composition, the last line in (3), some parasitic literals are included due to the implication operator.

Table 3: Derivation of cell row constraint.

i != j → C_{row_i}.val != C_{row_j}.val	⟹	!(i != j) || (C_{row_i}.val != C_{row_j}.val)
	⟹	(i == j) || (C_{row_i}.val != C_{row_j}.val)

At randomization time, each loop is unrolled to compose a propositional formula that is then presented to the constraint solver. Making the assumption that `i` and `j` are replaced with numeric constants during unrolling, their performance impact on the solver *should* be negligible. Thus, there are two literals per clause and $2 * ((n^2 - 1) * n^2) * n^2$ (non-unique) literals in the formula.

Question 2: Is it better to use an implication or an if-else constraint?

Instead of the implication operator, however, we might opt to employ an if-else construct, with an empty else expression. For example, the constraint in the last line of (3) could be changed from an implication to an if-statement.

```
if(c != i)
    c_row_c.val != c_row_i.val;
```

The SystemVerilog standard states that the "if-else style constraint declarations are equivalent to implications" [11]. Therefore we suppose that an implication or an if-else are similarly efficient during solving. We test this theory in the object oriented Sudoku game board in section "OOP Optimization 1: if-else constraint in place of implication", on page 16.

Question 3: Are constants more efficient than variables?

Often verification environments use variables or parameters for configuration. We assume that the constraint solver would be more efficient with constant values. We test this theory in section 4, pages 18 and 21.

Question 4: Would removing duplicates from the row/column/square improve efficiency?

As the Sudoku game board increases in size, the number of range constraints and inequalities increases exponentially. Consider the complete set of row 0 inequalities in (4).

C_{00}.val != C_{01}.val && …
C_{01}.val != C_{00}.val && …

While it is easy to see that the quantities above are simply reversed but logically the same, it may not be so obvious to the solver. The question here is if the solver employs a re-ordering scheme to identify quantities like these. If it does, then the solver will automatically identify these clauses as duplicate and *not* create a new clause instance for the second. Therefore, removing the duplicate from the constraint composition should have negligible effect on solver efficiency. We test this theory in section 4, page 23, for an object-oriented Sudoku game board, and again in section 5, page 29 for a flat game board.

The set of column constraints are similarly composed, except that the invariant in the inner loops is the column.

```
for(int col = 0; col < n^2; col += 1)
    for(int i = 0; i < n^2; i += 1)
        for(int j = 0; j < n^2; j += 1)
            i != j -> c_i_col.val != c_j_col.val
```
(5)

Finally, the composition of the set of Latin-square constraints follows the same pattern such that cell values are unique in the square. However, the loops must track the square and both row and column (note: implementation in the test bench is easier than (6) appears).

```
for(int sq = 0; sq < n^2; sq += 1)
  for(int r = ((sq % n) * n);   // each row of square
          r < (((sq % n) * n) + n); r += 1)
    for(int c = ((sq / n) * n); // each col of square
            c < (((sq / n) * n) + n); c += 1)
      for(int i = 0; i < n; i += 1)   // build the
        for(int j = 0; j < n; j += 1) // inequalities
        {
          int ri = (i % n) + ((sq % n) * n);
          int cj = (j % n) + ((sq / n) * n);
          (r != ri) || (c != cj) ->
                       c_r_c.val != c_ri_ci.val
        }
```
(6)

Question 5: Would removing all duplicates from the game board improve efficiency?

Consider the 2×2 Latin-square Sudoku game board in Figure 2. Cell C_{00} will be compared to cell C_{01} in two constraint set compositions: row and Latin-square. In both, the composed inequality will be identical. Also, there will be reversal inequalities amongst several constraint sets. Similar to **Question 3**, this theory tests to see how much more efficient the solver can be for an exponential search space. This is tested in section 5, page 29.

Sudoku vs. the Real World

We can make some assumptions about the effectiveness of the simulator constraint solver based on the Sudoku game board. Sudoku is easily parameterized and scalable. A solver may be implemented in an object-oriented fashion or within a single class or module. These are trade-offs taken when architecting a constrained random verification environment. Since we put much faith in our ability to generate random stimulus, it might be nice to know:

a) How much a solver can handle (capacity);
b) How can we compose our constraints to best fit the solver's solving algorithms.

With Sudoku we can easily saturate the constraint solver with unit clauses whose contents must be fully satisfied. Composing a constraint in SystemVerilog may look very different from the actual formula presented to the solver. With Sudoku we have control over our constraint and its presentation. Finally, Sudoku has a specific construct that is expanded and replicated. We can test how the constraint solver handles these scenarios ... or must the verification engineer tinker with the constraint for the sake of efficiency?

Methodology

We have composed two basic test benches: an object-oriented test bench (OOP) and a flat test bench (FLAT). The OOP test bench separates constraints on cells from the inequality sets (arrays) and the game board. The constraint formula in the OOP test bench must cross instances of cells and arrays at randomization-time of the game board. The FLAT test bench simplifies the structure to a single class: all range constraints and inequality sets are represented in the single class.

For both OOP and FLAT types, a single Sudoku game board was instantiated (with no hints—no cell has an initial value) at the test bench top. The size of the game board was set at compile-time via a command-line argument (`+define+N=`n). For each test bench type, the same set of one hundred random seeds was executed in VCS [8] over all sizes of n shown in Table 4.

Table 4: Sudoku Board Sizes Employed in Testing

n	Latin-Square Size ($n \times n$)	Sudoku Board Size
2	2×2	4×4
3	3×3	9×9
4	4×4	16×16
5	5×5	25×25
6	6×6	36×36
7	7×7	49×49

Additionally, for FLAT test benches, a comparison was made between the SystemVerilog simulator's constraint solver and two publicily available SAT solvers: mathsat5 [5] and yices [6]. For these, the set of SystemVerilog constraints were generated in SMT2, these solvers' input language [12].

All tests were executed on a compute farm of identical machines, each containing Intel Xenon processors with 64 cores, 4096 GB of RAM. Jobs were limited to a CPU execution time of 40 minutes and maximum 60 GB of memory (RAM and/or swap space as designated by the scheduler). Test benches are compiled in one step and simulated in another so that the CPU time results are only for simulation. Furthermore, each simulation created the Sudoku game board and randomized a single time only. We assume the overhead for normal SystemVerilog simulation is considered minimal and therefore not taken into account.

Finally, post-processing scripts written in Perl [13], verified correctness on completed Sudoku game boards to ensure cell values were unique in row, column, and Latin-square according to rules. Solutions between optimizations in the same test bench were compared, when feasible, for sameness per game board size and seed. Thus, unless indicated, the solutions found between optimizations using the same seed were identical. Therefore, the metrics can accurately show differences in constraint solving time. If a test experienced time-out or memory-out, this was noted in the graphs but not incorporated into minimum, maximum, and average computed solving times.

Question 6: Object oriented or flat constraints?

Our intuition states that constraints well within the single class scope should be more efficient than crossing instance boundaries. This is tested in the remainder of the paper.

Uniqueness Constraint

As stated throughout sections 2 and 3, a Sudoku cell value must be unique in the row, column, and Latin-square, simultaneously. SystemVerilog defines a built-in uniqueness constraint:

uniqueness_constraint ::= unique { open_range_list } [11].

It would seem natural to apply the uniqueness constraint to an array of cell class instances (refer to the `cell` class from section 2, page 4). However, SystemVerilog restricts the open_range_list, above, to a list where each element is one of:

- "A scalar variable of integral type,"
- "An unpacked array whose leaf element is integral, or a slice of such a variable" [11].

A class instance reference is not an integral type. Therefore, we cannot use the unique constraint for the object-oriented test benches. Uniqueness may be employed in the FLAT test benches because all random variables exist within a single class. However, in an effort to keep test bench comparisons as similar as possible, since we do not use the unique constraint in the OOP test benches we do not use it in the FLAT test benches either.

4. Object Oriented Game Board (OOP)

The architecture for the object oriented game board, refer to Figure 1, has three components: a cell class, a list class containing an array of cells, and a game board class.

The cell class requires two members, a maximum range value and a current value.

```
class scell; // single cell
    rand int m_val;
    int m_max; // can optimize
    constraint c { m_val inside {[1:m_max]}; }
    function new(int max);
        m_max = max;
    endfunction
endclass
```

Given an $n \times n$ Latin-square Sudoku game board, each cell maximum value is set at construction by the list class such that `max` = n^2. In the `scell` class, above, the maximum value used in the constraint is a non-constant variable. Therefore, the constraint solver must perform at least one memory access to retrieve the value `m_max` during solving.

One constraint block, with two unit clauses, is required per cell instance to ensure random selection from a valid contiguous range. For an $n \times n$ Latin-square Sudoku game board, there are $n^2 * n^2$ cells instantiated. Thus, $2 * n^2 * n^2$ clauses are always required.

The list class models each set of Sudoku constraints: row, column, and Latin-square.

```
class slist; // single list
    rand scell cel[];
    constraint valid {
        foreach(cel[i])
            foreach(cel[j])
                i != j -> cel[i].m_val != cel[j].m_val; // can optimize
    }
endclass
```

The same constraint may be applied over the array regardless of the Sudoku constraint set. This is because the `slist cel` array contains cell references from a row, column, or Latin-square and the constraint equation must ensure the same result: no cells in the array may have an equivalent value. The key, then, is to construct the `slist cel` array references properly.

The Sudoku game board class, `sgrid`, instantiates each of cell and sets reference in each of the constraint sets.

```
class sgrid; // single grid
    local rand slist m_row[`NUM_ROWS];
    local rand slist m_col[`NUM_ROWS];
    local rand slist m_box[`NUM_ROWS];
    function void build();
        for(int row = 0; row < `NUM_ROWS; row++) begin
            for(int col = 0; col < `NUM_ROWS; col++) begin
```

```
                int box_row = ((row / n) * n) + (col / n);
                int box_col = ((row % n) * n) + (col % n);

                // instantiate all cells in the m_row arrays
                m_row[row].cel[col] = new(`NUM_ROWS);

                // set references to cells in m_col and m_box arrays
                m_col[col].cel[row] = m_row[row].cel[col];
                m_box[box_row].cel[box_col] = m_row[row].cel[col];
            end
        end
    endfunction
endclass
```

Equations (3), (5), and (6) are employed simultaneously in the build function to instantiate and assign cell instance references for rows, columns, and Latin squares, respectively. Finally, given an $n \times n$ Latin-square Sudoku game board, each cell maximum value is set to n^2 at instantiation while the `NUM_ROWS macro is set to n^2 at compile-time.

The constraint block in the list generates more than the complete set of constraints to solve the Sudoku board. The implication operator composes one clause of two literals per iteration of the inner foreach loop, regardless of the equivalency of `i` and `j`. Two kinds of clauses are composed:

$i = j$: `(i == i) || ~~(cel[i].m_val != cel[i].m_val)~~` **(7)**

$i \neq j$: `~~(i == j)~~ || (cel[i].m_val != cel[j].m_val)` **(8)**

If the `i` and `j` iterators are equivalent then the first literal in the composed clause is solved while the second is disregarded, indicated in (7) by the strike-through. When the `i` and `j` iterators are not equivalent, the second literal is solved and the first disregarded, as in (8). Regardless, a clause will be generated when the foreach loops are unrolled. Therefore, the number of clauses per row, column, and square is $n^2 * n^2 * n^2 = n^6$. Table 5 shows the total number of clauses the basic OOP test bench composes at randomization time.[1]

Table 5: Number of Clauses in Basic OOP Test Bench.

Latin-Square $n \times n$	Sudoku Game Board	Number of Cells	Number of Clauses				
			Cell (range)	Row	Column	Square	Total
2×2	4×4	16	32	64	64	64	224
3×3	9×9	81	162	729	729	729	2,349
4×4	16×16	256	512	4,096	4,096	4,096	12,800
5×5	25×25	625	1,250	15,625	15,625	15,625	48,125
6×6	36×36	1296	2,592	46,656	46,656	46,656	142,560
7×7	49×49	2401	4,802	117,649	117,649	117,649	357,749

[1] This is an absolute number barring optimizations in simulator-specific pre-solve processing.

Table 6 presents a summary of solving time per Sudoku Game Board size using the basic OOP test bench. Metrics for both VCS-2014 [8] and VCS-2011 [14] are listed, and Figure 3 details per-seed comparisons. Notice that VCS-2014 outperforms VCS-2011 in nearly every case.

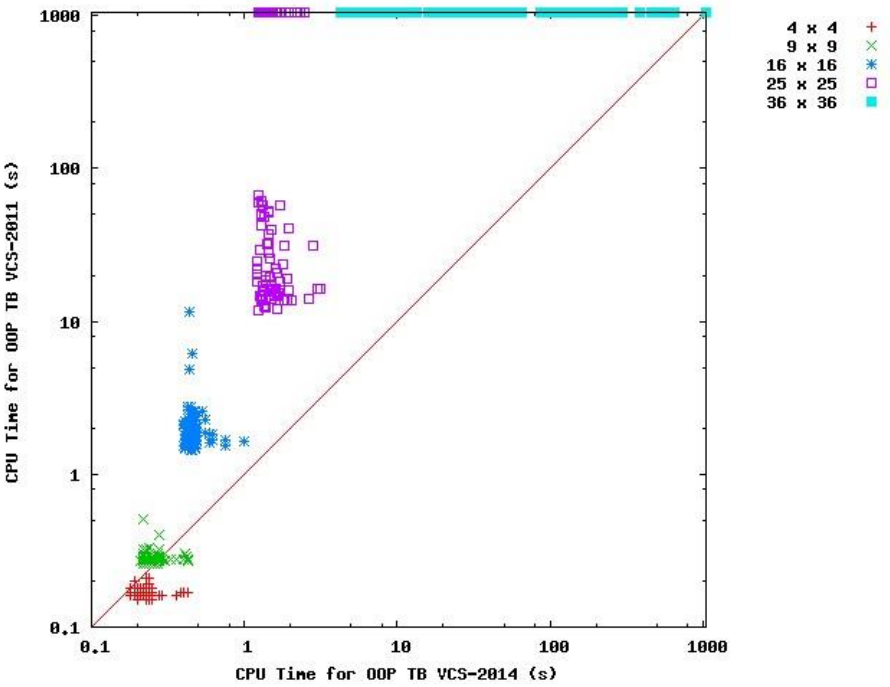

Figure 3: VCS-2014 outperforms VCS-2011 in nearly all cases.

This Basic OOP test bench executed with VCS-2014 is the one all optimizations (also executed with VCS-2014) are compared against. For smaller game board sizes, it appears the number of clauses has little effect on solving time. However, the more than 48,000 clauses in the 25×25 game board finally consistently requires more than one second. The spread of solving time from minimum to maximum is 5.94s-1.20s = 4.74s. Notice that that this spread increases dramatically as the game board size further increases, until only a single 49×49 instance is solved.

Table 6: Solving Time Summary for Basic OOP Test Bench.

	Sudoku Game Board	No. Seeds Solved	VCS Solving Time (s)		
			Min	Average	Max
VCS-2014	4×4	100	0.18	0.23	0.43
	9×9	100	0.21	0.26	0.43
	16×16	100	0.40	0.47	0.99
	25×25	100	1.20	1.60	5.94
	36×36	100	4.29	109.32	1,040.92
	49×49	1	587.48	587.48	587.48
VCS-2011	4×4	100	0.15	0.17	0.21
	9×9	100	0.26	0.28	0.51
	16×16	100	1.43	2.03	11.53

25×25	63	11.79	25.58	66.34
36×36	0	-	-	-
49×49	0	-	-	-

OOP Optimization 1: if-else constraint in place of implication

In section 3, "Dissecting Sudoku Constraints", we showed that parasitic literals are composed during the unrolling of the constraints. Recall from the derivation of the implication operator in Table 3 that the looping iterators resolve to the following clause (e.g. in a row constraint):

```
(i == j) || (cel[i].m_val != cel[j].m_val).
```

We stated that both iterators, `i` and `j`, are replaced with integer values. An efficient solver should be able to (a) maintain only unique copies of any (`i == j`) literal (i.e., 0 == 1 exists exactly once in the propositional formula to solve) and (b) solve the literals immediately, with negligible impact on performance.

We can model this scenario by replacing the implication operator with an if-else statement. As per the SystemVerilog standard [11]:

if (expression) constraint_set [else constraint_set].

The standard says that the if-else statement and the implication offer equivalent functionality. However, whereas the implication is reduced to a Boolean quantity ($a \rightarrow b \Longrightarrow !a||b$), the standard says: if the expression is true then the first constraint_set must be satisfied [11].

We test the theory that the constraint solver, in a pre-processing step, may reduce the number of generated clauses (and possibly no patristic literals) during unrolling. This would imply that, in effect, the expression can be solved immediately and only unit clauses composed. Therefore, in this OOP test bench we have re-written the list's valid constraint as follows:

```
constraint valid {
    foreach(cel[i])
        foreach(cel[j])
            if(i != j) // optimization 1: if-else instead of implication
                cel[i].m_val != cel[j].m_val;
}
```

If our theory is correct, then an inequality should be composed *only* when the iterators `i` and `j` are not equivalent. The result is that the number of clauses generated should be $((n^2 - 1) * n^2) * n^2$. The expected number of clauses to solve is listed inTable 7.

Table 7: Expected Number of Classes in OOP Test Bench using if-else Construct

Sudoku Game Board	Number of Cells	Number of Clauses					
		Cell (range)	Row	Column	Square	Total	Reduction

4 × 4	16	32	48	48	48	176	21.43%
9 × 9	81	162	648	648	648	2,106	10.34%
16 × 16	256	512	3,840	3,840	3,840	12,032	6.00%
25 × 25	625	1,250	15,000	15,000	15,000	46,250	3.90%
36 × 36	1296	2,592	45,360	45,360	45,360	138,672	2.73%
49 × 49	2401	4,802	115,248	115,248	115,248	350,546	2.01%

Referring to Table 8 and Figure 4 we see that either approach has only a small effect on the constraint solver's time required for each Sudoku game board size. Table 8 shows that, in all cases Sudoku game board sizes from 4 × 4 to 36 × 36 the maximum solving time has been reduced. More importantly, the spread from minimum to maximum solving times has been flattened. That is, overall, the formula was more consistently solved near the average time. For example, the 25 × 25 game board in the basic OOP test bench had range spread (max-min) of 4.74s, whereas in Table 8, the same game board size indicates a spread of only 1.84s.

Table 8: Solving Time Summary for OOP Test Bench using if-else Construct.

Sudoku Game Board	No. Seeds Solved	VCS Solving Time (s)		
		Min	Average	Max
4 × 4	100	0.17	0.21	0.39
9 × 9	100	0.22	0.25	0.74
16 × 16	100	0.41	0.50	2.42
25 × 25	100	1.16	1.69	3.00
36 × 36	100	4.32	108.74	1,548.41
49 × 49	1	2,039.92	2,039.92	2,039.92

Figure 4 compares the solving time required per Sudoku game board size and seed. Dots above the middle line indicate the Basic OOP test bench outperformed the OOP test bench with Optimization 1. While there are outliers for each game size, in general the solving time required is very similar with implication (x-axis) versus if-else (y-axis). Overall, the if-else construct has a small effect on the solving time required. This seems to imply that the if-else construct may differ slightly but has a similar impact on solver performance when compared to the implication construct. For VCS, the answer to **Question 2**, from page 9, seems to be: it doesn't matter much, but if-else does appear to be more consistent in solving time required.

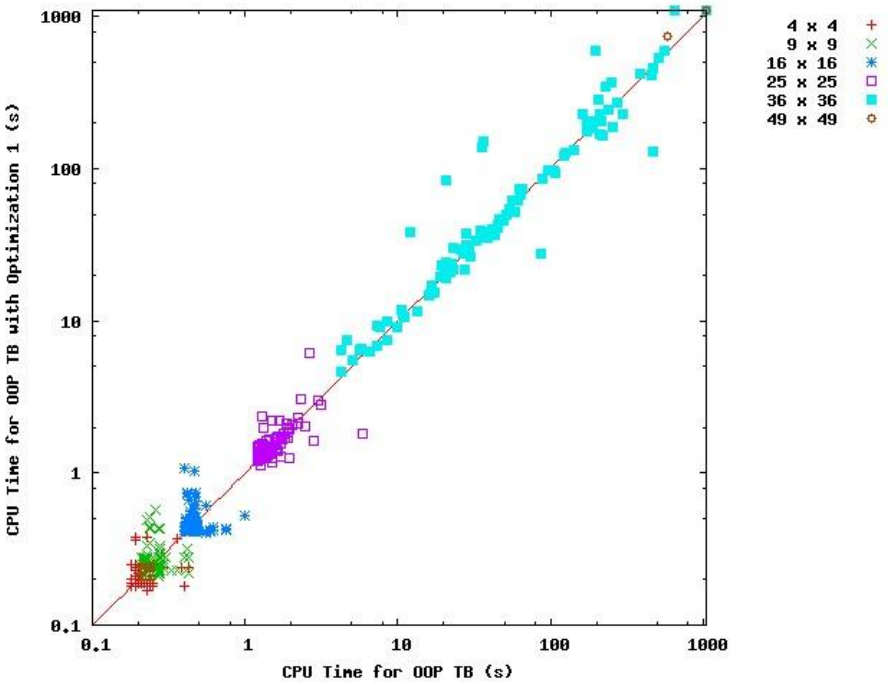

Figure 4: Solving Times for OOP Test Bench using if-else construct.

OOP Optimization 2: constant or parameter in cell constraint

The range constraint employed in the `scell` class, section 4, page 12, sets the constraint maximum using a class variable whose value is set at construction. Depending on how the solver approaches this constraint, each time the clause `m_val <= m_max` is evaluated a memory access may be required on `m_max`. However, according to Latin-square and Sudoku rules, a change in the maximum cell value implies a change to the structure of the game board. Therefore, it is feasible to modify the cell range constraint to use a constant value.

SystemVerilog provides two options for constant values [11]. In the first, a constant class variable may be employed by attributing the `m_max` variable with the keyword **`const`**.

```
class scell;
    rand int m_val;
    const int m_max;
    constraint c { m_val > 0 && m_val <= m_max; }
    function new(int max);
        m_max = max;
    endfunction
endclass
```

A constant class variable tells the SystemVerilog compiler that once the value is set at construction it cannot change throughout the life of the scell *instance*. Therefore write access is not required for `m_max` and, thus, it can reside directly in immediate-type CPU instructions (value encoded within the instruction). However, as the implementation of **`const`** variables is simulator-proprietary (the SystemVerilog standard provides no guidance), we can force an

immediate instruction by using a number or compile-time macro in the place of the `m_max` class variable.

```
class scell; // single cell
    rand int m_val;
    `define m_max `NUM_ROWS // used also in slist
    constraint c { m_val > 0 && m_val <= `m_max; }
    function new(); endfunction
endclass
```

Whereas the constant class variable requires no change (no write) throughout the life of the instance, the macro requires no change throughout the life of *all* instances in the simulation. This is not an issue for Sudoku since all cells have the same range constraint always.

For Table 9, we have *not* employed optimization 1. In other words, the implication operator is used in the `slist` constraint block for both cases. It is interesting note, in Figure 5, that while the solving times required for **`const`** versus the basic OOP test bench are very similar, the spread from minimum to maximum has increased for some game board sizes.

Table 9: Solving Time Summary for OOP Test Bench against different class storage types (optimization 2).

	Sudoku Game Board	No. Seeds Solved	VCS Solving Time (s)		
			Min	Average	Max
Constanat	4 × 4	100	0.18	0.22	0.36
	9 × 9	100	0.21	0.26	0.61
	16 × 16	100	0.40	0.48	1.23
	25 × 25	100	1.16	1.86	7.86
	36 × 36	100	3.93	104.48	986.74
	49 × 49	1	595.58	595.58	595.58
Parameter	4 × 4	100	0.17	0.23	0.53
	9 × 9	100	0.21	0.25	0.40
	16 × 16	100	0.39	0.52	2.21
	25 × 25	100	1.13	1.55	3.00
	36 × 36	100	4.12	108.94	1,070.65
	49 × 49	0	-	-	-

However, referring to Table 9 and Figure 6, when the constraint was changed to a parameter, the solving times seem to go haywire. Indeed, no 49 × 49 game boards were solved with the parameter option. Recall that the solutions found per game board size and per seed were verified identical via post-processing script. Therefore, even though solving times for Figure 6 are distributed over a greater area, the final result is an identical solution. We can attribute the differences in solving time to *how* the solver arrived at the final solution. The fact that the solutions are identical, the choices made during solving may differ with constant values versus memory accesses on the `m_max` parameter. The mean over all seeds, however, is very similar between `m_max` as constant storage class or parameter.

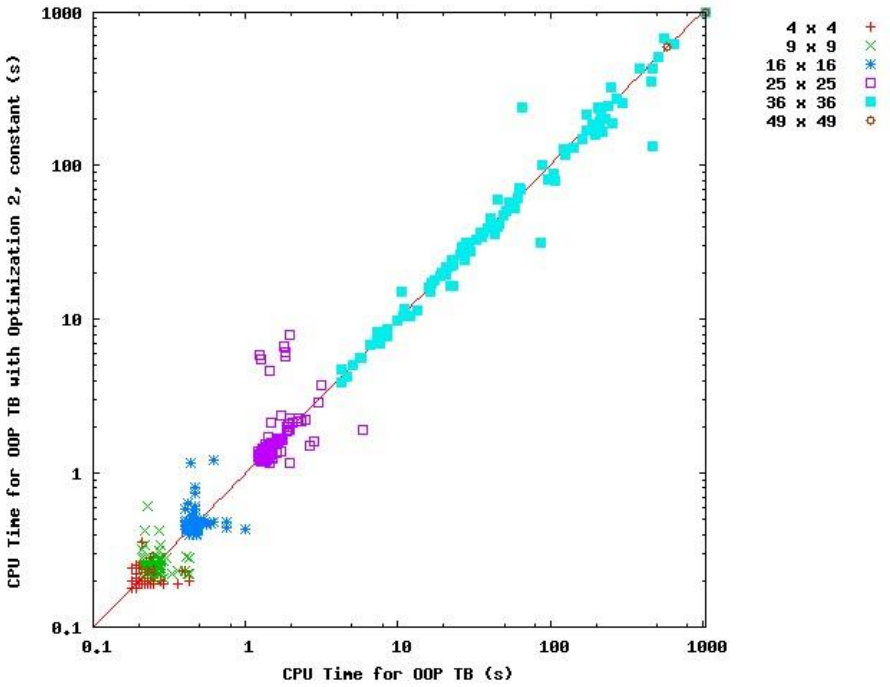

Figure 5: Solving Times for OOP Test Bench with constant `m_max` class member.

With respect to **Question 1**, page 8, the answer seems to be: it doesn't matter much. If anything, it looks like the VCS constraint solver favors variables over constants. This, however, does not even hold for every Sudoku game board size. Compare Figure 6 to Figure 5 for the 25×25 game board size. All dots are tightly coupled near the diagonal line. This shows that both storage classes are approximately equivalent.

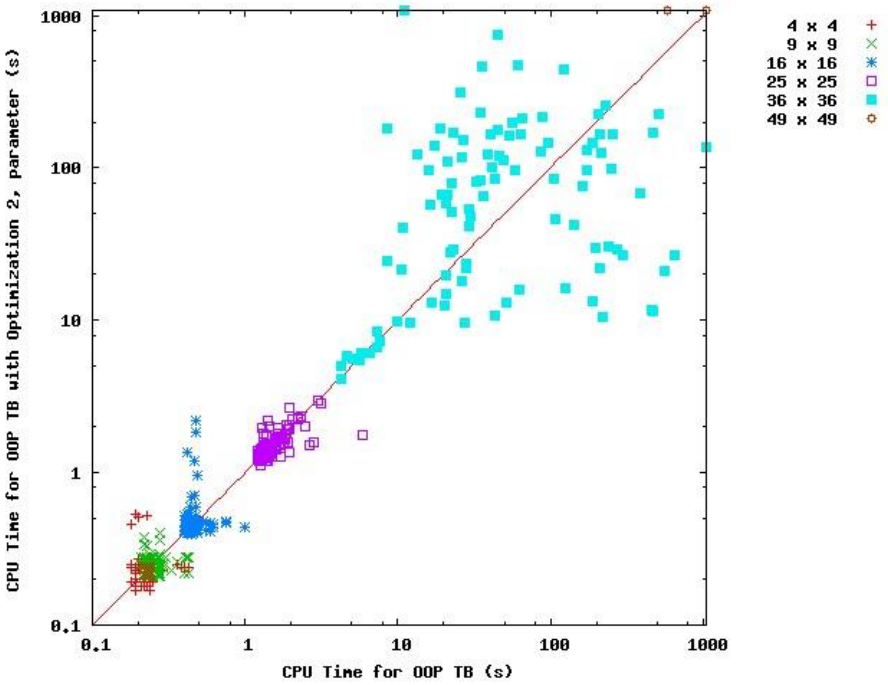

Figure 6: Solving Times for OOP Test Bench with parameter `m_max`.

OOP Optimization 3: combine optimization 1 & 2

For OOP test bench optimization 3, we combine both optimization 1, if-else construct, and optimization 2, constant and parameter for cell range maximum. Note that the number of clauses in this test bench should match Table 7. Solving time summary in Table 10 shows that optimization 3 has very similar timing metrics compared to optimization 2.

Table 10: Solving Time Summary for OOP Test Bench with Optimizations 1 and 2.

	Sudoku Game Board	No. Seeds Solved	VCS Solving Time (s)		
			Min	Average	Max
Constanat	4 × 4	100	0.18	0.22	0.28
	9 × 9	100	0.21	0.26	0.96
	16 × 16	100	0.40	0.47	1.00
	25 × 25	100	1.18	1.60	5.07
	36 × 36	100	4.42	110.15	1,051.10
	49 × 49	1	631.72	631.72	631.72
Parameter	4 × 4	100	0.17	0.23	0.53
	9 × 9	100	0.21	0.27	1.60
	16 × 16	100	0.39	0.53	2.32
	25 × 25	100	1.09	1.63	5.82
	36 × 36	100	3.98	110.48	992.43
	49 × 49	-	-	-	-

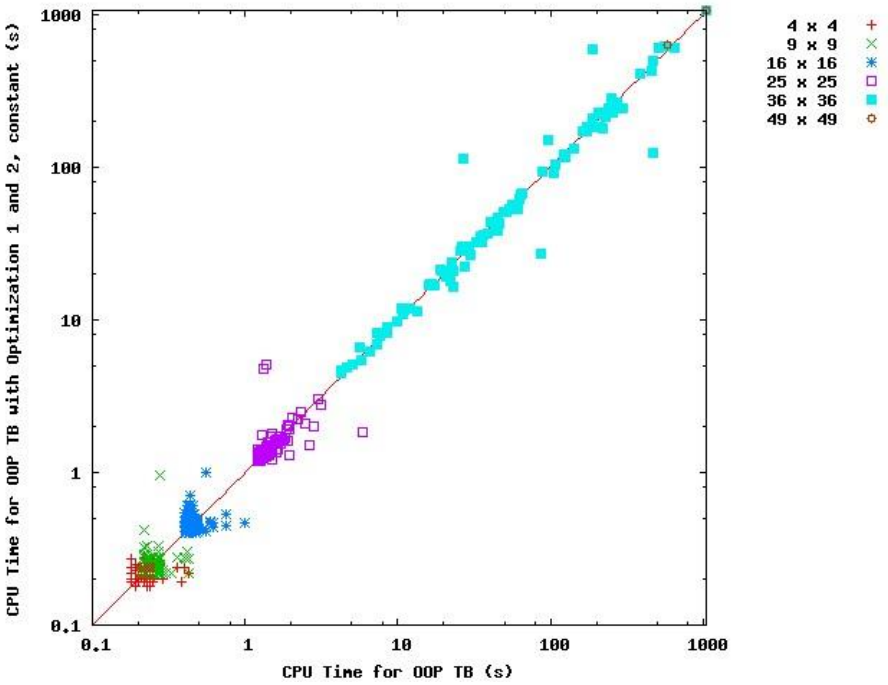

Figure 7: Solving Times for OOP Test Bench using optimizations 1 and 2, constant class member.

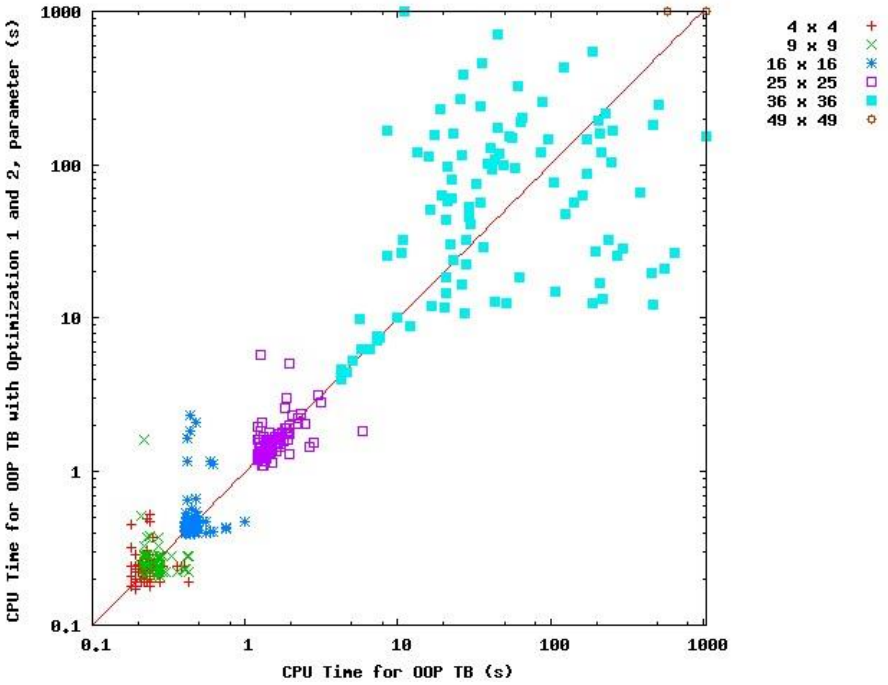

Figure 8: Solving Times for OOP Test Bench using optimizations 1 and 2, parameter.

Figure 7 for optimization 3 is very similar to Figure 5 for optimization 2. The same is true for Figure 8 and Figure 6. However, in Figure 9, we compare the local const class member versus a parameter. With the small Sudoku game board sizes, it appears that the parameter outperforms the const. The 36×36 game board appears to have approximately equal number on either side

of the diagonal line. We can conclude that combining the if-else construct with the constant storage class has very little additional effect on solving time.

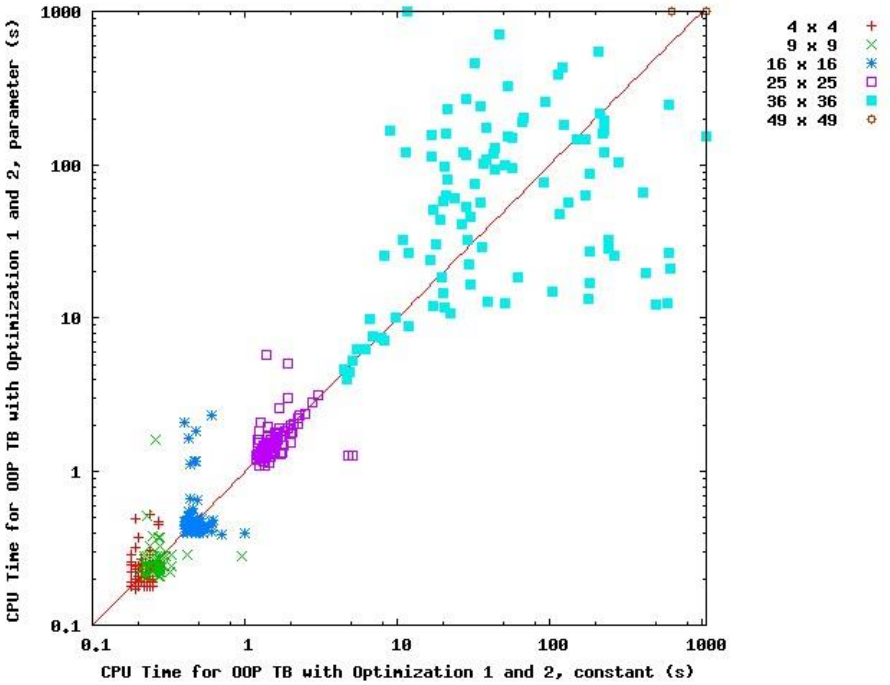

Figure 9: Solving Times for OOP Test Bench comparing storage types, parameter vs constant.

Regarding **Questions 1**, **2**, and **3**, pages 8 and 9, it again looks as though VCS constraint solver can easily handle all storage class kinds when utilizing the if-else construct. Thus, between the various optimizations, we lean toward disregarding storage class and if-else construct. The readability of the constraint block having higher importance.

OOP Optimization 4: reduction of array duplicates

In section 3, starting on page 7, we showed that array constraint inequalities are duplicated within the same constraint set array: row, column, and Latin-square. We can reduce the set of inequalities that require solving with a simple change to the constraint.

Referring to (3), we see the foreach loops operating in an always increasing fashion (i.e., 0th cell before 1st cell before 2nd cell and so on). We assume that regardless of when and where the constraint is composed (**randomize** is called) during simulation, this ordering will always be followed. Then we can say that if the current cell under examination `c` is in a greater position than the `i`th cell for comparison then we need no inequality. Basically, the inequality was already composed by the previous iteration of the middle loop.

```
for(int row = 0; row < n^2; row += 1)
    for(int c = 0; c < n^2; c += 1)
        for(int i = 0; i < n^2; i += 1)
            i > c -> c_row c.val != c_row i.val;
```
(9)

Then, for row 0, the following sets of inequalities are composed. This technique results, as shown below, in six inequalities, or six fewer than the complete set for row 0 from (4).

$$C_{00}.val != C_{01}.val \;\&\&\; C_{00}.val != C_{02}.val \;\&\&\; C_{00}.val != C_{03}.val \;\&\&$$
$$C_{01}.val != C_{02}.val \;\&\&\; C_{01}.val != C_{03}.val \;\&\&$$
$$C_{02}.val != C_{03}.val.$$

Because equations for row (3), column (5), and Latin-square (6) were employed to *construct* the OOP game board array sets, we need only change the list's valid constraint to apply the optimization to each.

```
constraint valid {
    foreach(cel[i])
        foreach(cel[j])
            if(j > i) // optimization 4: > instead of !=
                cel[i].m_val != cel[j].m_val;
}
```

We are using the if-else construct, with empty else, above because we noticed from Optimization 1 that the VCS solver seems to be more consistent in time. Together, Optimization 4 should require only half the number of clauses that need be solved to find a valid solution to the Sudoku game board, as shown in Table 11.

Table 11: Expected Number of Clauses for Array Reduction OOP Test Bench.

Sudoku Game Board	Number of Cells	Number of Clauses					
		Cell (range)	Row	Column	Square	Total	Reduction
4 × 4	16	32	24	24	24	104	53.57%
9 × 9	81	162	324	324	324	1,134	51.72%
16 × 16	256	512	1,920	1,920	1,920	6,272	51.00%
25 × 25	625	1,250	7,500	7,500	7,500	23,750	50.65%
36 × 36	1296	2,592	22,680	22,680	22,680	70,632	50.45%
49 × 49	2401	4,802	57,624	57,624	57,624	177,674	50.34%

For this test, we employed all three storage types for the maximum range value constraint: local variable, const variable, and parameter. The solving time summary is presented in Table 12 and time comparisons in the figures following.

Table 12: Solving Time Summary for Array Reduction OOP Test Bench.

	Sudoku Game Board	No. Seeds Solved	VCS Solving Time (s)		
			Min	Average	Max
Variable	4 × 4	100	0.17	0.22	0.29
	9 × 9	100	0.20	0.24	0.55
	16 × 16	100	0.33	0.39	0.92
	25 × 25	100	0.79	1.27	4.63
	36 × 36	100	3.26	107.08	1,041.81

	49 × 49	1	585.06	585.06	585.06
Constanat	4 × 4	100	0.17	0.22	0.40
	9 × 9	100	0.20	0.24	0.47
	16 × 16	100	0.32	0.39	0.91
	25 × 25	100	0.82	1.25	5.24
	36 × 36	100	3.12	104.32	1,060.59
	49 × 49	1	679.28	679.28	679.28
Parameter	4 × 4	100	0.17	0.23	0.58
	9 × 9	100	0.21	0.28	1.55
	16 × 16	100	0.32	0.40	1.84
	25 × 25	100	0.76	1.27	6.58
	36 × 36	100	2.76	114.09	1,340.97
	49 × 49	-	-	-	-

Figure 11, Figure 12, and Figure 13 show the solving comparison versus the Basic OOP test bench. Note, however, that the solution solved does *not match* the solution for the same game board size and seed in the Basic OOP test bench.

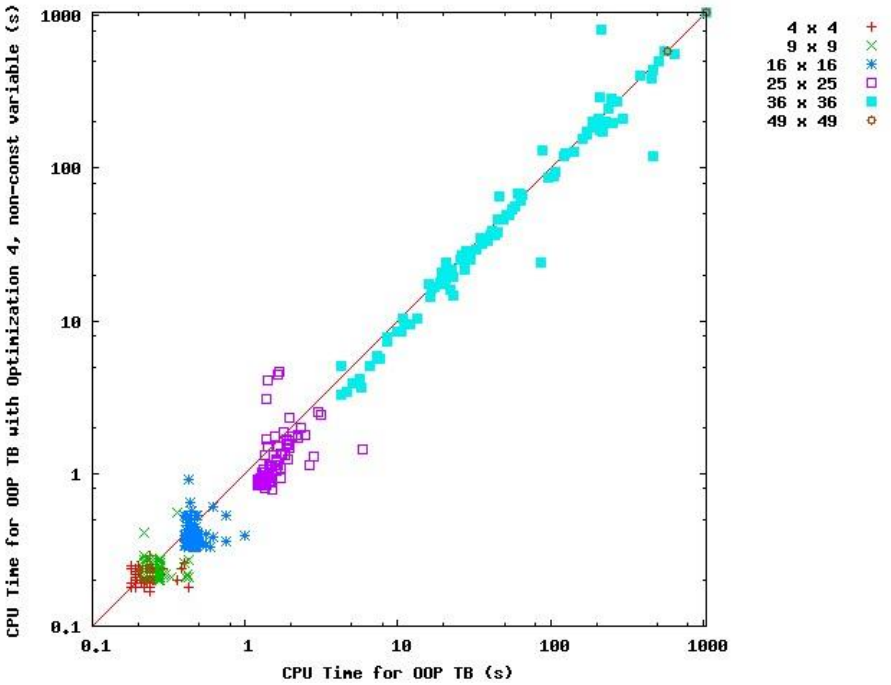

Figure 10: Solving Times for OOP Test Bench with reduced inequalities, variable class member.

Regarding **Question 4**, page 10. Our intuition told us that focusing on reducing the clausal explosion as much as possible would greatly enhance the solving time metric. However, even though the number of clauses to solve was reduced by more than half (the parasitic literals still exist, however), the metrics have not shifted versus the Basic OOP test bench. It is also interesting to compare the solving summary in Table 12, Parameter section, with its FLAT test bench equivalent, Table 14, Reduced section. The FLAT test bench is able to solve two 49 × 49 Sudoku game boards versus none for the OOP test bench. Otherwise, all FLAT timing metrics

are worse than their corresponding OOP metrics. Our conclusion to **Question 4** is no, reducing the number of clauses in the propositional formula in this manner does not have the desired effect. Again, we return to readability of the constraint block having more importance than structure.

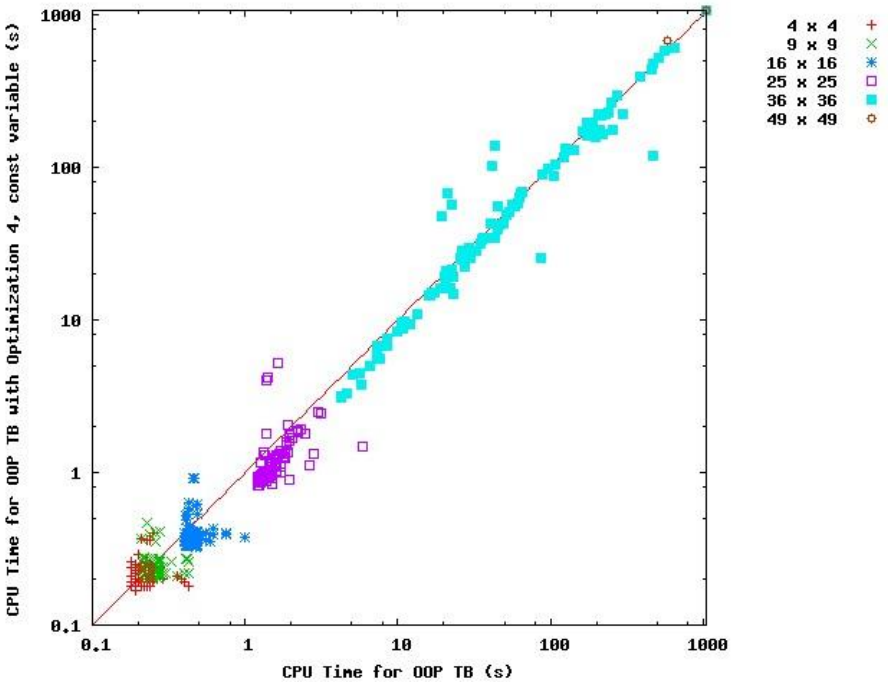

Figure 11: Solving Times for OOP Test Bench with reduced inequalities, constant class member.

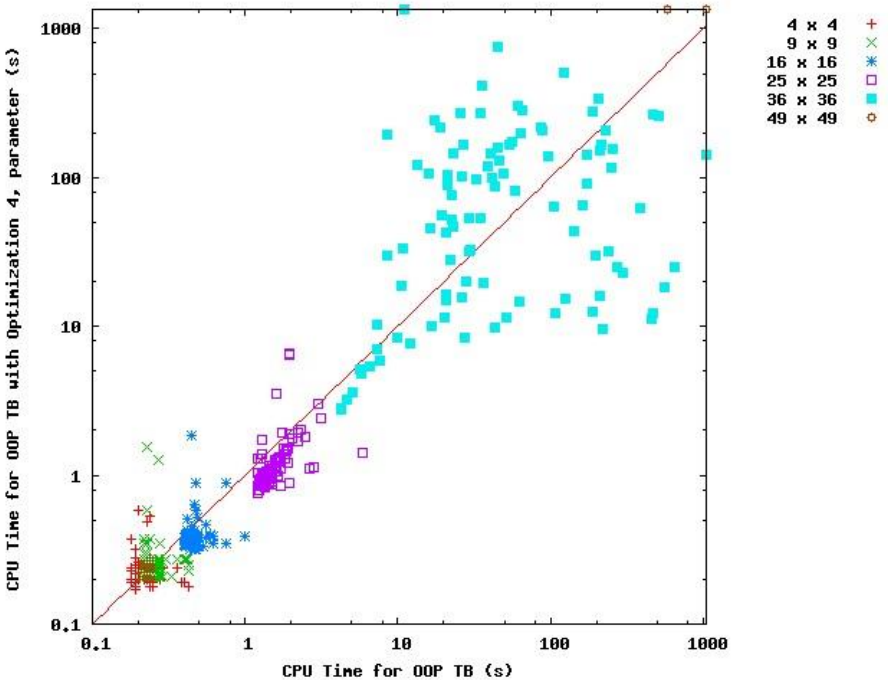

Figure 12: Solving Times for OOP Test Bench with reduced inequalities, constant parameter.

5. Flat Game board (FLAT)

We theorized that a flat constant game board would prove to have the shortest constraint solving times. To that end, we wrote a Perl [13] script to generate four sets of flat game boards:

- Full – similar to the OOP game board,
- Reorder,
- Reduction – similar to the OOP optimization 4 game board,
- Unique.

Each test bench contained exactly one class definition per game board size. All random variables and constraints were fully contained within the class. No cross-reference or inheritance was employed. For example, a snippet of the Sudoku game board for $n = 2$ is shown below.

```
class sgrid_4x4;
    rand int c_1_1; // row 1, column 1
    …
    rand int c_4_4; // row 4, column 4

    constraint cells_valid {
        c_1_1 inside {[1:4]};
        …
        c_4_4 inside {[1:4]};
    }
    constraint rows_valid {
        c_1_1 != c_1_2;
        …
    }
    constraint cols_valid {
        c_1_1 != c_2_1;
        …
    }
    constraint squares_valid {
        c_1_1 != c_1_2;
        …
        c_2_1 != c_1_1;
        …
    }
endclass
```

Due to the size of the game board, it was infeasible to construct them by hand. As such, we wrote a Perl script that employs each of the equations for row (3), column (5), and Latin-square (6) constraint sets to output the constraint explicitly. The simulation methodology was exactly the same as OOP game board.

Because each class was generated from a script—using an internal model as its basis—we also output the SMT2 constraint language. For example, a snippet for the Sudoku game board $n = 2$ is shown below for SMT2:

```
; Grid 4x4 – variable declarations
(declare-fun c_1_1 () Int) ; row 1, column 1
…
```

```
(declare-fun c_4_4 () Int) ; row 4, column 4

; cells valid
(assert (>= c_1_1 1))
(assert (<= c_1_1 4))
…
(assert (>= c_4_4 1))
(assert (<= c_4_4 4))

; rows valid
(assert (not (= c_1_1 c_1_2)))
…

; columns valid
(assert (not (= c_1_1 c_2_1)))
…

; squares valid
(assert (not (= c_1_1 c_1_2)))
…
(assert (not (= c_2_1 c_1_1)))
…

; Satisfy?
(check-sat)
(get-model)
(exit)
```

We employed public domain solvers mathsat5 and yices, executing the one the same machine farm and under the same constraints as the SystemVerilog simulators [5], [6]. Both solvers are highly considered in the Boolean Satisfiability community and compete in the annual SMT-COMP competition [15].

FLAT Optimization 1: reorder array inequalities

During pre-processing the solver should be able to identify identical clauses, disallowing duplicates. We tested this theory by isolating the effect of duplicate inequalities by reordering them during game board generation so that they identically match their duplicate. In (4), we showed that within the same array, duplicate inequalities are composed with the terms reversed:

$$C_{00} != C_{01} \&\& C_{01} != C_{00} \ldots$$

As the constraints are generated by the script, we check to see if a duplicate exists either in the natural ordering or in the reverse ordering. If so, then the duplicate is output instead of the reverse. Thus, the above constraint is composed as:

$$C_{00} != C_{01} \&\& C_{00} != C_{01} \ldots$$

The point is to see if reordering affects solver performance. We theorized that the solver should already be reordering constraints as they are parsed. If true, then our reordering should have negligible effect on solver performance when compared to the basic FLAT test bench.

FLAT Optimization 2: reduction of array duplicates

In this optimization we completely removed duplicates *within* an array of constraints. A modification to the script implemented OOP Optimization 4, on page 23. Therefore, the generated set of constraints per row, column, and square contained no duplicates.

FLAT Optimization 3: unique inequalities

Finally, we took FLAT Optimization 2 one step further by removing *all* duplicate inequalities from the generated game board. A modification to the script hashed both the forward and reverse ordered inequality for Sudoku the game board as a whole. Only those inequalities that were unique in the game bard were generated.

FLAT Test Bench Results

Table 13, below, identifies the exact number of clauses presented in each FLAT test bench. The cell range constraint is composed as a SystemVerilog inside constraint. According to Table 1, each inside constraint requires exactly two clauses. This calculation is indicated in the Cell column. The remainder of the columns represent absolute counts of unit clauses in the generated by the scripts.

Table 13: Clause Count Generated by FLAT Test Bench Generation Script.

	Sudoku Game Board	Number of Cells	Number of Clauses					
			Cell (range)	Row	Column	Square	Total	Reduction
Flat	4 × 4	16	32	48	48	48	176	-
	9 × 9	81	162	648	648	648	2,106	-
	16 × 16	256	512	3,840	3,840	3,840	12,032	-
	25 × 25	625	1,250	15,000	15,000	15,000	46,250	-
	36 × 36	1,296	2,592	45,360	45,360	45,360	138,672	-
	49 × 49	2,401	4,802	115,248	115,248	11,5248	350,546	-
Reordered	4 × 4	16	32	48	48	48	176	-
	9 × 9	81	162	648	648	648	2,106	-
	16 × 16	256	512	3,840	3,840	3,840	12,032	-
	25 × 25	625	1,250	15,000	15,000	15,000	46,250	-
	36 × 36	1,296	2,592	45,360	45,360	45,360	138,672	-
	49 × 49	2,401	4,802	115,248	115,248	115,248	350,546	-
Reduction	4 × 4	16	32	24	24	24	104	40.91%
	9 × 9	81	162	324	324	324	1,134	46.15%
	16 × 16	256	512	1,920	1,920	1,920	6,272	47.87%
	25 × 25	625	1,250	7,500	7,500	7,500	23,750	48.65%
	36 × 36	1,296	2,592	22,680	22,680	22,680	70,632	49.07%
	49 × 49	2,401	4,802	57,624	57,624	57,624	177,674	49.32%
Unique	4 × 4	16	32	24	24	8	88	50.00%
	9 × 9	81	162	324	324	162	972	53.85%
	16 × 16	256	512	1,920	1,920	1,152	5,504	54.26%
	25 × 25	625	1,250	7,500	7,500	5,000	21,250	54.05%
	36 × 36	1,296	2,592	22,680	22,680	16,200	64,152	53.74%

49 × 49	2,401	4,802	57,624	57,624	43,218	163,268	53.42%

Table 14, below, presents the time summary for the VCS solver. In nearly every case, Flat, Reordered, Reduced, and Unique, the average solving time was worse than the Basic OOP test bench. Two notes regarding Table 14. First, the maximum time for larger solved puzzles (25 × 25 and above) was reduced. Specifically, the Reduced maximum time for 36 × 36 was nearly cut in half compared to the Basic OOP test bench. Regarding **Question 5**, page 10, we cannot conclude that over the aggregate, removing duplicate clauses has a noticeable effect on solving time for the VCS constraint solver.

Table 14: Solving Time Summary for all FLAT Test Bench Techniques.

	Sudoku Game Board	No. Seeds Solved	VCS Solving Time (s)		
			Min	Average	Max
Flat	4 × 4	100	0.18	0.28	1.36
	9 × 9	100	0.20	0.25	0.70
	16 × 16	100	0.41	0.58	2.79
	25 × 25	100	1.19	1.90	5.17
	36 × 36	100	12.61	137.72	746.75
	49 × 49	2	1,664.83	1,700.46	1,736.10
Reordered	4 × 4	100	0.17	0.21	0.66
	9 × 9	100	0.21	0.25	0.35
	16 × 16	100	0.42	0.51	1.27
	25 × 25	100	1.34	2.11	5.70
	36 × 36	100	13.50	158.19	716.66
	49 × 49	3	1,100.11	1,501.66	1,931.08
Reduced	4 × 4	100	0.18	0.26	1.47
	9 × 9	100	0.19	0.32	1.50
	16 × 16	100	0.33	0.38	0.65
	25 × 25	100	0.79	1.46	6.36
	36 × 36	100	8.22	127.85	551.55
	49 × 49	2	1,177.81	1,644.47	2,111.13
Unique	4 × 4	100	0.17	0.22	0.41
	9 × 9	100	0.19	0.22	0.51
	16 × 16	100	0.32	0.42	1.55
	25 × 25	100	0.80	1.61	7.58
	36 × 36	100	8.60	148.51	1,138.90
	49 × 49	2	919.23	1,459.09	1,998.92

The puzzle solutions to the FLAT test benches are not identical to their OOP counterparts. As such, comparison on an individual seed basis is not very meaningful. Nonetheless, we have plotted the Basic OOP test bench versus FLAT unique tes tbench in Figure 13.

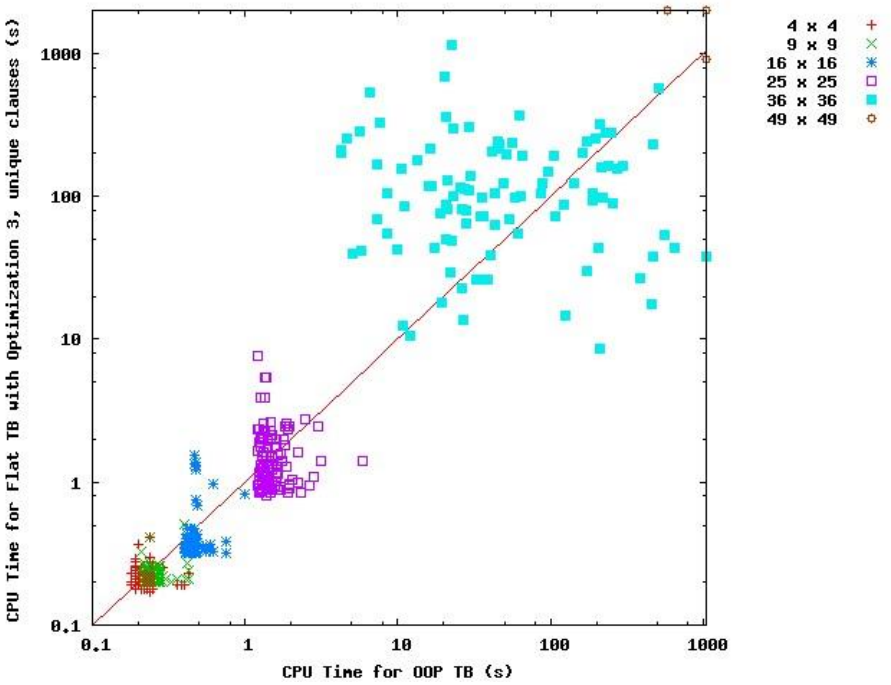

Figure 13: Solving times comparing Basic OOP TB vs Flat TB with optimization 3.

Finally, in Table 15, we compare the VCS average solving time on FLAT test benches versus their identical compositions in the SMT2 language and solved by publicly available solvers Mathsat5 and Yices. The table only lists up to 16 × 16 because neither public domain solvers solved anything larger on any seed. It is dramatic the difference between VCS and these solvers.

Table 15: Solving Time Summary VCS versus Mathsat5 and Yices.

	Sudoku Game Board	No. Seeds Solved			Average Solving Time (s)		
		VCS	MSAT5	YICES	VCS	MSAT5	YICES
Flat	4 × 4	100	100	100	0.28	0.02	0.00
	9 × 9	100	100	100	0.25	24.93	26.73
	16 × 16	100	0	0	0.58	-	-
Re-order	4 × 4	100	100	100	0.21	0.02	0.00
	9 × 9	100	100	100	0.25	21.55	6.67
	16 × 16	100	0	0	0.51	-	-
Re-duced	4 × 4	100	100	100	0.26	0.02	0.00
	9 × 9	100	100	100	0.32	30.07	3.86
	16 × 16	100	0	1	0.38	-	1,562.66
Uni-que	4 × 4	100	100	100	0.22	0.02	0.00
	9 × 9	100	100	100	0.22	21.99	7.70
	16 × 16	100	0	0	0.42	-	-

6. Conclusions

We set out to characterize the VCS constraint solver based on a few typical situations. Perhaps the object-oriented versus the flat architecture comparison is most important:

Question 6: Object oriented or flat constraints?

Inheritance and cross-referencing hierarchical paths is a common occurring in constraint random verification. Intuition states that a flat test bench should be a bit faster for solving propositional formulae. With our testing of VCS, we have found no noticeable difference in time metric between OOP and FLAT test benches. However, the FLAT test bench solved two or three 49×49 Sudoku game board puzzles where the OOP test bench solved at most one. This seems to imply that the capacity of the constraint solver at these larger propositional formula sizes does improve with constants, even if the solving time does not.

The following questions focused on the storage class of bounds (not the random variable itself):

Question 1: What is performance impact when n is a number versus n is a variable?
Question 3: Are constants more efficient than variables?

Intuitively, we assumed that constants variables or parameters would have an impact on simulator's constraint solving time. However, with our testing of VCS, we found no noticeable difference in time metric between non-constant variables, constant variables, and constant parameters.

The following question focused on the structure of the constraint block:

Question 2: Is it better to use an implication or an if-else constraint?

We theorized that the SystemVerilog analyzer or some pre-processing step would be able to filter out the constraint if the if-expression failed, since the iterator arguments are numbers. With VCS, we found that either construct is performs about the same in solving time for both OOP and FLAT test benches.

Finally, the following questions targeted the absolute size of the composed propositional formula to solve:

Question 4: Would removing duplicates from the row/column/square improve efficiency?
Question 5: Would removing all duplicates from the game board improve efficiency?

We presented how the constraint solver may transform a set of SystemVerilog constraints into a propositional formula, then steps it might take to solve the formula. In general, a smaller formula should require less time to solve. This was specifically shown with public domain solvers mathsat5 and yices. However, with VCS our testing showed that our reduction methods have very little effect on the constraint solver, regardless of OOP or FLAT test bench architecture. On the assumption that a propositional formula with fewer clauses should take less time to solve (i.e., its complexity is reduced), then we conclude that the lack of difference in

solving time with VCS was that our reductions simply were not great enough. For example, the number of clauses in the propositional formula to solve increases exponentially as the Sudoku game board size increases (e.g., 9×9 to 16×16). However, our reduction techniques achieved only a linear reduction. The number of clauses in the final formula was still too great to solve in a reasonable amount of time, although we did notice an increased capacity with the FLAT test benches.

Our overall conclusion is that, with Synopsys' VCS constraint solver, readability of the constraint block is overwhelmingly more important than the structure of the block for time-efficiency purposes.

References

[1] I. Lynce and J. Ouaknine, "Sudoku as a SAT Problem," in *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2006.

[2] H. Simonis, "Sudoku as a Constraint Problem," in *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, 2005.

[3] G. Kwon and H. Jain, "Optimized CNF Encoding for Sudoku Puzzles," in *Proceedings of the 13th International Conference on Logic Programming for Artificial Intelligence and Reasoning (LPAR)*, 2006.

[4] U. Pfeiffer, T. Karnagel and G. Scheffler, "A Sudoku-Solver for Large Puzzles using SAT," in *Logic for Programming Artificial Intelligence and Reasoning (LPAR) Short Papers*, 2010.

[5] A. Cimatti, A. Griggio, B. Schaafsma and R. Sebastiani, "The Mathsat5 SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[6] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, 2014.

[7] C. Barrett, R. Sebastiani, S. A. Seshia and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, vol. 185, A. Biere, M. Huele, H. van Maaren and T. Walsch, Eds., IOS Press, 2008, pp. 825-885.

[8] Synopsys, Inc., *VCS Mx / VCS MXi User Guide*, J-2014.12 ed., 2014.

[9] Wikipedia, "Latin Square--Wikipedia, the Free Encyclopedia," [Online]. [Accessed 4 September 2014].

[10] Wikipedia, "Sudoku--Wikipedia, the Free Encyclopedia," [Online]. [Accessed 4 September 2014].

[11] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," New York, 2013.

[12] C. Barrett, P. Fontaine and C. Tinelli, "The SMT-LIB Standard: Version 2.5," 2015.

[13] L. Wall, "The Perl Programming Language," 2006.

[14] Synopsys, Inc., *VCS Mx / VCS MXi User Guide*, E-2011.03 ed., 2011.

[15] D. R. Cok, A. Griggio, R. Bruttomesso and M. Deters, "The 2012 SMT Competition," 2012. [Online]. Available: http://smtcomp.sourceforge.net/2012/reports/SMTCOMP2012.pdf.