2015 16th International Workshop on Microprocessor and SOC Test and Verification

Performance of a SystemVerilog Sudoku Solver with VCS

Jeremy Ridgeway
Avago Technologies, Ltd.
Ph: +1 970-288-5211
Email: Jeremy.Ridgeway@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 solvers efficiency.**

I. 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 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 [8], [10]. The game board follows strict rules that are easy to parameterize and scale. Large puzzles provide significant challenges for satisfiability (SAT) solvers [7], [9]. 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 three public domain SAT solvers, mathsat5 [2], yices [5], and Z3 [4].

II. Sudoku

Sudoku is an implementation of the Latin-square model. A Latin-square is an $n \times n$ array of cells populated with n^2

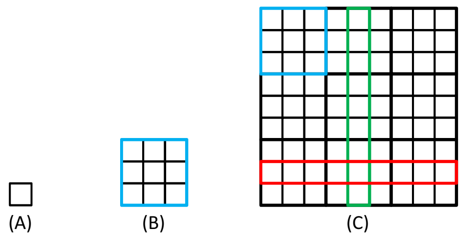
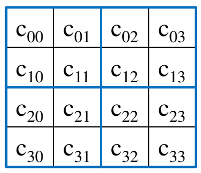


Fig. 1: Sudoku game model: (A) single cell, (B) Latin-square, and (C) full grid of rows (red), columns (green), squares (blue).

different symbols each occurring exactly once in every row and column [14]. The blue square in Fig. 1-(B) and again in Fig. 1-(C) is one Latin-square. 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 Fig. 1-(C). Each of n^2 different symbols occur exactly once in each row, column, and Latin-square of the board [15].

A. Sudoku Cell 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}

Fig. 2: A 2×2 Latin-square Sudoku game board: 4×4 cells.

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 Fig. 2 is labelled according to row and column position: $C_{\text{row col}}$. 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. 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:

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

SystemVerilog specifies that all constraints on a given random variable must be considered simultaneously in a conjunctive fashion [6]. Therefore, when an instance of the cell is randomized, the constraint solver performs a composition and pre-processing step to construct a Boolean formula, simplifies that to conjunctive normal form (CNF), then employs SAT and satisfiability modulo theory (SMT) techniques to solve. The general CNF structure is given in (2).

$$\phi = \bigwedge_{i=1}^{L} \bigvee_{j_i=1}^{K_j} l_{j_i} \tag{2}$$

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), $c_i = \vee_{j_i} l_{j_i}$ is a single Boolean *clause*. A propositional formula, ϕ, is the conjunction (logical AND) of all its clauses, $\wedge_i c_i$. A CNF formula is essentially a formula in product-of-sums Boolean logic form.

TABLE I: Derivation of contiguous range term to literals.

`val inside {[1:9]}` ⇒	
⇒ `(1 <= val <= 9)`	
⇒ `((1 <= val) && (val <= 9))`	Distribution
⇒ `!(!(1 <= val) || !(val <= 9))`	De Morgan's
⇒ `!(1 > val) && !(val > 9)`	De Morgan's

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`**.

The Boolean formula for `val` from (1) and Table I consists of two clauses, $\phi = c_0 \;\&\&\; c_1$. Each clause contains a single negative literal, $c_0 \leftrightarrow !l_0$ and $c_1 \leftrightarrow !l_1$. The literals are related to predicates and the Boolean quantities as:

$$!l_0 \longleftrightarrow !p_0 \qquad p_0 \longleftrightarrow \texttt{(1 > val)}$$
$$!l_1 \longleftrightarrow !p_1 \qquad p_1 \longleftrightarrow \texttt{(val > 9)}.$$

To constrain all cells to a valid range for an $n \times n$ Sudoku game $n^2 * n^2 = n^4$ constraints of the form (1) are required. In other words, $2 * n^4$ unit clauses must be solved simultaneously. For a 2×2 Latin-square Sudoku game, 32 clauses ensure each cell contains a valid value.

B. Sudoku Array Constraints

In addition to the cell constraint, the cell's value must be unique in each of three arrays: row, column, and Latin-square.

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). Assume each cell in the game board is modelled as a class with randomized data member, `val`. Then, in general, for an $n \times n$ Latin-square Sudoku game, the constraints for each cell within the row set of constraints is:

Listing 1: Sudoku row constraints.

```
1 for(int row=0; row < n^2; row+=1)
2   for(int c=0; c < n^2; c+=1)
3     for(int i=0; i < n^2; i+=1)
4       c != i -> c_row_c.val != c_row_i.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, as in Listing 1, to ensure the cell has a unique value in the row.

Listing 1, line 1, considers one row at a time, the `row`th row. Line 2, considers one cell at a time, the `c`th cell within the row. The innermost loop, lines 3-4, compares the cell currently under examination, `c`, with the `i`th cell in the row. Either the current cell <u>is</u> 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.

In general, the number of inequalities in a complete set per row is $(n^2 - 1) * n^2$. An inequality should not be composed for the current cell versus the current cell in each row, thus -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, Listing 1, line 4, some parasitic literals are included due to the implication operator, as in Table II.

TABLE II: Parasitic literals (i == j) in array constraints.

i != j → $C_{\text{row\_i}}$`.val` != $C_{\text{row\_j}}$`.val` ⇒
⇒ !(i != j) || ($C_{\text{row\_i}}$`.val` != $C_{\text{row\_j}}$`.val`)
⇒ (i == j) || ($C_{\text{row\_i}}$`.val` != $C_{\text{row\_j}}$`.val`)

At randomization time, each loop is unrolled to compose a propositional CNF 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.

The set of column constraints are composed in the same way as Listing 1, replacing `row` with the column index. However, the composition of the set of Latin-square constraints must track both current row and column, Listing 2.

Listing 2: Sudoku Latin-square constraints.

```
1 for(int sq=0; sq < n^2; sq+=1)
2  for(int r = ((sq % n) * n);
3          r < (((sq % n) * n) + n); r++)
4   for(int c = ((sq / n) * n);
5           c < (((sq / n) * n) + n); c++)
6    for(int i=0; i < n; i++)
7     for(int j=0; j < n; j++) {
8        int ri = (i%n) + ((sq % n) * n);
```

```
 9          int cj = (j%n) + ((sq / n) * n);
10          (r != ri || c != cj) ->
11             c_r_c.val != c_ri_ci.val; }
```

C. Testing Methodology

We have composed two basic test benches. The object-oriented (OOP) test bench constrains cell values separately from the row, column, and Latin-square inequality sets 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.[1]

For both OOP and FLAT, a single Sudoku game board was instantiated (with no hints–no cell had an initial value) per simulation, and solved in a single **randomize**() step. The size of the OOP game board was determined at compile time via macro definition. All game board sizes in the range $n = \{[2 : 7]\}$, resulting in Sudoku boards from 4×4 to 49×49, were simulated. However, at most, two 49×49 Sudoku boards were solved per test set. As such, their solving times have largely been omitted from the results in Table IV at the end of this paper. Additionally, FLAT test benches were compared against public SAT solvers mathsat5 [2], yices [5], and Z3 [4]. For these, the set of SystemVerilog constraints were written in the solvers' input language, SMT2 [1], and only the random seed provided on the command-line (base options otherwise).

All tests were executed on a compute farm of identical machines, each containing Intel Xenon processors with 64 cores and 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). CPU times reported reflect only simulation time, compilation was performed separately. We assume the overhead for normal SystemVerilog simulation is considered minimal and therefore not taken into account.

Finally, post-processing scripts, in Perl [13], verified correctness on completed Sudoku game boards. Solutions between optimizations in the same test bench were compared (e.g., OOP Basic vs. Optimization 1), 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 1. 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.

III. Object Oriented Game Board (OOP)

The architecture for the object oriented game board, refer to Fig. 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 current value.

Listing 3: Single cell in the Sudoku Game Board.

```
1 class scell;
2  rand int val;
3  int max; // can optimize
4  constraint c {val inside {[1:max]};}
5  function new(int m); max=m; endfunction
6 endclass
```

Given an $n \times n$ Latin-square Sudoku game board, each cell's 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.

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

Listing 4: Single Sudoku array constraint.

```
1 class slist;
2  rand scell cel[];
3  constraint valid {
4   foreach(cel[i])
5    foreach(cel[j])
6     i != j -> cel[i].val != cel[j].val;}
7 endclass
```

The same constraint may be applied over the array regardless of the Sudoku constraint set. This is because the same goal is always achieved: no cells in the cel array, line 2, may have an equivalent value. The key, then, is to build the cel array properly. We do so in Listing 5 by simultaneously employing Listings 1 and 2 in the build function. The macro, `NUM_ROWS, is set on the compile command-line.

Listing 5: Sudoku game board.

```
 1 class sgrid;
 2  const int n2 = `NUM_ROWS;
 3  rand slist row[n2], col[n2], box[n2];
 4  function void build();
 5   for(int r=0; r<n2; r++)  // row
 6    for(int c=0; c<n2; c++) // col
 7    begin// Row & column Latin-sq indexes
 8     int rs = ((r/n)*n)+(c/n);
 9     int cs = ((r%n)*n)+(c%n);
10     row[r].cel[c] = new(n2);
11     col[c].cel[r] = row[r].cel[c];
12     box[rs].cel[cs] = row[r].cel[c];
13    end
14  endfunction
15 endclass
```

[1]The **unique** constraint was not employed because the OOP test bench maintained an array of references, an invalid unique constraint construct.

The total number of clauses required to solve each Basic OOP Sudoku board is given in Table IV. We compared constraint solving performance between VCS-2011 [11] and VCS-2014 [12]. Both axes in Fig. 3 represent CPU solving time. For points above the center line, the x-axis performance was better; below the center line, the y-axis. It is clear that VCS-2014 performs far better then VCS-2011 over the same test bench in nearly every simulation.

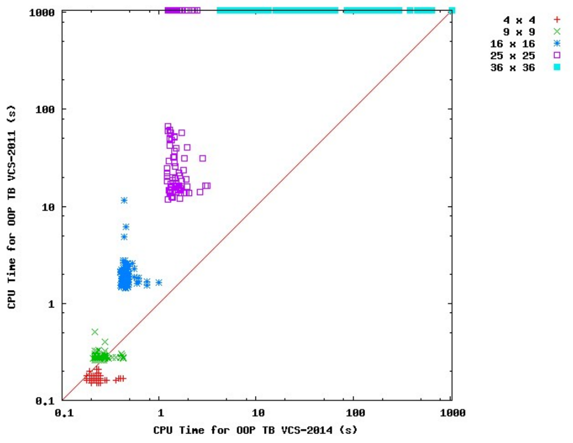


Fig. 3: VCS-2014 outperforms VCS-2011 in nearly all cases.

A. *OOP optimization 1: if-else vs. implication*

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

Referring to the row constraint in Listing 1, we showed that parasitic literals are composed during the unrolling of the **foreach** loops. At line 4, an implication operator is used when comparing loop indexes. From Table II, the implication composes two literals. During CNF formula composition, 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 (e.g., 0 == 1 exists exactly once) and (b) solve the literals immediately, with negligible impact on performance. We test this theory by replacing the implication in Listing 4, line 6, with the if-else construction shown below.

```
if(i != j) cel[i].val != cel[j].val;
```

The SystemVerilog standard states that the "if-else style constraint declarations are equivalent to implications" [6]. Therefore we suppose that an if-statement should be as efficient as the implication. Further, we make the assumption that clauses are not composed when the if-statement, above, is not satisfied, resulting in a reduction in the CNF formula.

Comparing test benches Basic OOP VCS-2014 against Optimization 1 in Fig. 4 and Table IV shows very little difference in solving time. As such, the if-else construct seems to have only a small effect. For VCS, neither implication nor if-else have significant benefit over the other. Although, from Table IV, solving if-else appears to be more consistent in time as the formula grows.

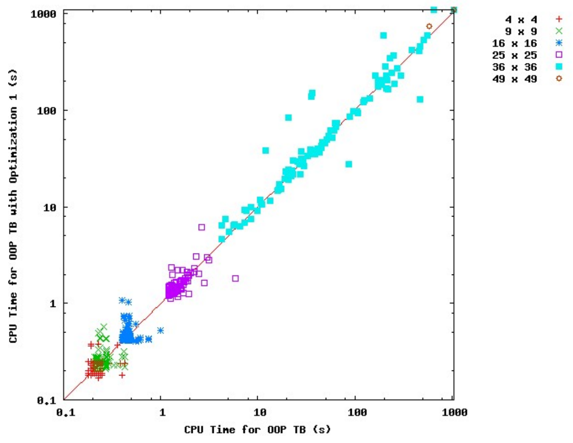


Fig. 4: Solving Times for OOP Test Bench using if-else construct.

B. *OOP optimization 2: constant vs. parameter in the cell*

Question 3. Are constants more efficient than variables?

The range constraint in `scell` class in Listing 3, line 4, sets the maximum range with a class variable. Depending on how the solver operates, each time the clause `val <= max` is evaluated a memory access may be required on `max`. However, Latin-square and Sudoku rules dictate that the maximum cannot change for a single game board. Therefore, it is feasible to modify this constraint to use a constant value.

SystemVerilog provides two options for constant values [6]. First, a constant class variable may be indicated using a keyword: **const int** `max`. Fig. 5 compares the Basic OOP test bench against Optimization 2 (constant).

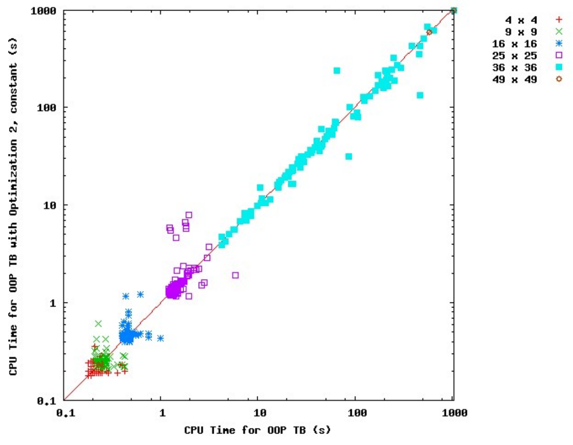


Fig. 5: Solving Times for OOP Test Bench with constant `m_max` class member.

However, the **const** keyword may or may not imply an immediate-type CPU instruction (value encoded within the instruction). The implementation is simulator-dependent. Therefore, for the second constant, we can force the immediate-type by using a compile-time macro: `val >= 0 && val <=`NUM_ROWS`. Fig. 6 compares the Basic OOP test bench against Optimization 2 (parameter).

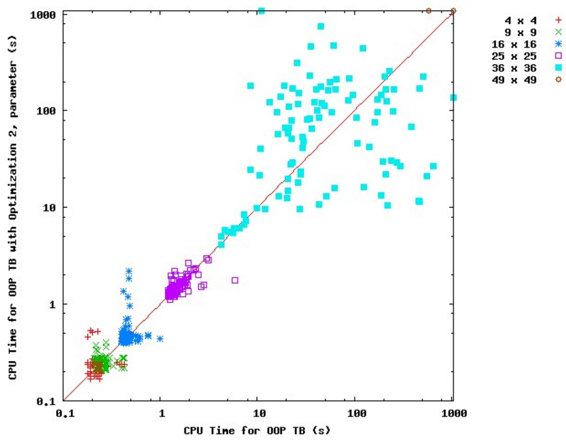


Fig. 6: Solving Times for OOP Test Bench with parameter `m_max`.

Comparing the metrics for Optimization 2 in both cases from Table IV, we see that even though Fig. 6 varies wildly on the larger game board sizes, the overall performance is very similar. As such, we suggest that the VCS constraint solver is about as efficient in either variable, constant, and parameter accesses in constraints.

C. OOP optimization 3: reduction of array duplicates

Question 4. Does removing duplicates from the row/column/square constraint improve efficiency?

From Listing 1, we can see that duplicate constraint clauses will be composed. For a 2×2 Latin-square Sudoku game, the following is a *complete set* of inequalities for row 0:

$$\begin{aligned}&C_{00}\texttt{.val!} = C_{01}\texttt{.val \&\&}\ C_{00}\texttt{.val!} = C_{02}\texttt{.val \&\&}\\&C_{00}\texttt{.val!} = C_{03}\texttt{.val \&\&}\\&C_{01}\texttt{.val!} = C_{00}\texttt{.val \&\&}\ C_{01}\texttt{.val!} = C_{02}\texttt{.val \&\&}\\&C_{01}\texttt{.val!} = C_{03}\texttt{.val \&\&}\\&C_{02}\texttt{.val!} = C_{00}\texttt{.val \&\&}\ C_{02}\texttt{.val!} = C_{01}\texttt{.val \&\&}\\&C_{02}\texttt{.val!} = C_{03}\texttt{.val.}\end{aligned} \tag{3}$$

However, as the **foreach** loops are always incrementing over the array (i.e., 0th cell before 1st cell before 2nd cell, etc.), we can take advantage of this construct to reduce the number of clauses generated by rewriting line 6 in Listing 4.

```
if(j > i) cel[i].val != cel[j].val;
```

In other words, if current cell under examination, `j`, is in a greater position than the `i`th cell for comparison, then no inequality is necessary because it was already composed by a previous loop iteration. Then for row 0, (3) becomes a *reduced set* of inequalities:

$$\begin{aligned}&C_{00}\texttt{.val!} = C_{01}\texttt{.val \&\&}\ C_{00}\texttt{.val!} = C_{02}\texttt{.val \&\&}\\&C_{00}\texttt{.val!} = C_{03}\texttt{.val \&\&}\ C_{01}\texttt{.val!} = C_{02}\texttt{.val \&\&}\\&C_{01}\texttt{.val!} = C_{03}\texttt{.val \&\&}\ C_{02}\texttt{.val!} = C_{03}\texttt{.val.}\end{aligned} \tag{4}$$

For this test, we used the if-construct in the constraint because from Optimization 1 we noticed the VCS solver seems more consistent in time. From Table IV, Optimization 3 should compose half the number of clauses compared to the Basic OOP test bench. We compared all three storage types, local variable, **const** variable (Fig. 7), and parameter in Table IV.[2]

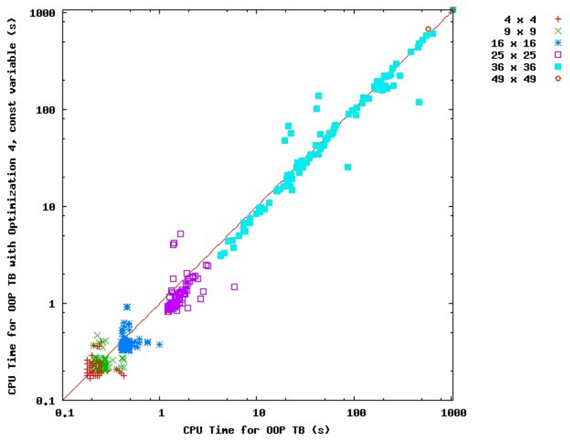


Fig. 7: Solving Times for OOP Test Bench with reduced inequalities, constant class variable.

Our intuition was that reducing the clausal explosion directly in the constraint would have a significant effect on the solving time required. That does not appear to be true. In fact, all OOP Optimizations proved largely ineffectual for VCS.

IV. Flat Game Board (FLAT)

We theorized that a flat constant game board would have the best constraint solving performance. To that end, we wrote a Perl5 [13] script to generate four sets of flat game board tests.

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 in Listing 6.

Listing 6: Flat Sudoku Game Board 2×2.

```
class sgrid_4x4;
 rand int c_1_1; // row 1, col 1
 ...
 constraint cells_valid {
  c_1_1 inside {[1:4]}; ...  }
 constraint rows_valid {
```

[2]Figures for local variable and parameter are similar to Fig. 7, but have been omitted due to space considerations.

```
  c_1_1 != c_1_2; ...  }
 constraint cols_valid {
  c_1_1 != c_2_1; ...  }
 constraint sqrs_valid {
  c_1_1 != c_1_2; ...
  c_2_1 != c_1_1; ...  }
endclass
```

Generation of constraints for row and column, Listing 1, and Latin-square, Listing 2, were handled in the script itself. The testing methodology was the same as the OOP game board.

The **Full FLAT** game board is similar to Basic OOP test bench with one difference: parasitic literals, as described in section II-B, Table II, were not generated. As such, the number of clauses in Table IV in OOP Basic VCS-2014 and FLAT Basic sections differ somewhat. However, the solving times required are very nearly the same in all cases.

The **Re-ordered** FLAT game board directly tests if the constraint solver is able to re-order and strip out duplicate clauses on its own. In this game board, we have re-ordered each clause so that it matches a duplicate clause if one exists. The solving metrics in Table IV indicate no significant difference between the FLAT re-ordered test bench and the OOP basic test bench.

The **Array Reduction** FLAT game board extends clausal re-ordering by actually removing the duplicates within each array (row, column, and Latin-square). The metrics in Table IV show the number of clauses FLAT versus Basic have been reduced. Still, the solving efficiency has not changed much.

The **Unique** FLAT game board extends array reduction by removing duplicate clauses from any constraint block. Thus, only unique clauses exist. The metrics in Table IV show an even further reduction in the CNF clauses, but, again, no significant reduction in solving efficiency. In fact, we were unable to find a significant difference in solving time on any FLAT versus OOP test bench; they both appear equally efficient for VCS.

Finally, a flat game board has a nearly one-to-one mapping to SMT2 constraint language, the common input to public domain SAT solvers (refer to Listing 7 for a 2×2 board).

Listing 7: Flat Sudoku Game Board 2×2 in SMT2.

```
(declare-fun c_1_1 () Int) ...
; cells valid
(assert (>= c_1_1 1))
(assert (<= c_1_1 4)) ...
; rows valid
(assert (not (= c_1_1 c_1_2))) ...
; ... cols valid, sqrs valid ...
(check-sat) ; Satisfy?
(get-model) ; Get cell values
```

We employed public domain solvers mathsat5 [2], yices [5] and Z3 [4], executing on the same machine farm and under the same constraints as the SystemVerilog simulators. All three solvers are highly considered in the Boolean Satisfiability community and often compete in the SMT-COMP competition [3]. Table III compares average solving time per game board size for each solver. Starting at the 16×16 game board, no public domain solver was able complete within the given 40 minute time window. VCS outperformed all three SAT solvers.

TABLE III: Ave. Solving Time (s) VCS vs. SMT.

	OOP		SMT		
Grid	Basic	Opt 3. const	MSAT4	Yices	Z3
4x4	0.23	0.22	0.02	0.00	0.01
9x9	0.26	0.24	21.99	7.70	174.64
16x16	0.47	0.39	N/A	N/A	N/A
25x25	1.60	1.25	N/A	N/A	N/A

V. Conclusions

We set out to characterize the VCS constraint solver based on a few typical situations. We took advantage of Sudoku's ability to saturate the solver with clauses due to its exponential scalability. However, we were unable to find any compelling reason that one constraint composition was more efficient than another. Our conclusion is that, for Synopsys VCS, the readability of the constraint block is overwhelmingly more important than its structure for time-efficiency purposes.

References

[1] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[2] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 7795 of *LNCS*. Springer, 2013.

[3] D. R. Cok, A. Griggio, R. Bruttomesso, and M. Deters. The 2012 SMT competition. In *SMT at The International Joint Conference on Automated Reasoning (IJCAR)*, pages 131–142, 2012.

[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] B. Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer, 2014.

[6] IEEE Computer Society. *SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, 1800-2012 edition, 2012.

[7] G. Kwon and H. Jain. Optimized CNF encoding for sudoku puzzles. In *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 1–5, 2006.

[8] I. Lynce and J. Ouaknine. Sudoku as a SAT problem. In *International Symposium on Artificial Intelligence and Mathematicsc (ISAIM)*, 2006.

[9] U. Pfeiffer, T. Karnagel, and G. Scheffler. A sudoku-solver for large puzzles using SAT. In *Logic for Programming Artificial Intelligence and Reasoning (LAPR) Short Papers*, pages 52–57, 2010.

[10] H. Simonis. Sudoku as a constraint problem. In *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27, 2005.

[11] Synopsys, Inc. *VCS(R)MX/VCS MXi(TM) User Guide*, E-2011.03 edition, 2011.

[12] Synopsys, Inc. *VCS(R)MX/VCS MXi(TM) User Guide*, J-2014.12 edition, 2014.

[13] L. Wall. The perl programming language, 2006.

[14] Wikipedia. Latin square — wikipedia, the free encyclopedia, 2015. [Online; accessed 3-September-2015].

[15] Wikipedia. Sudoku — wikipedia, the free encyclopedia, 2015. [Online; accessed 3-September-2015].

TABLE IV: Clause count and metrics for OOP and FLAT Test Benches.

		Number of Clauses						Seeds	VCS Solving Time (seconds)		
Test Set	Grid	Cell	Row	Col	Square	Total	Reduced	Solved	Min	Ave	Max
OOP Basic VCS-2011	4x4	32	64	64	64	224	-	100	0.15	0.17	0.21
	9x9	162	729	729	729	2,349	-	100	0.26	0.28	0.51
	16x16	512	4,096	4,096	4,096	12,800	-	100	1.43	2.03	11.53
	25x25	1,250	15,625	15,625	15,625	48,125	-	63	11.79	25.58	66.34
	36x36	2,592	46,656	46,656	46,656	142,560	-	0	-	-	-
OOP Basic VCS-2014	4x4	32	64	64	64	224	-	100	0.18	0.23	0.43
	9x9	162	729	729	729	2,349	-	100	0.21	0.26	0.43
	16x16	512	4,096	4,096	4,096	12,800	-	100	0.40	0.47	0.99
	25x25	1,250	15,625	15,625	15,625	48,125	-	100	1.20	1.60	5.94
	36x36	2,592	46,656	46,656	46,656	142,560	-	100	4.29	109.32	1,040.92
	49x49	4,802	117,649	117,649	117,649	357,749	-	1	587.48	587.48	587.48
OOP Optimize 1 if-else	4x4	32	48	48	48	176	21.43%	100	0.17	0.21	0.39
	9x9	162	648	648	648	2,106	10.34%	100	0.22	0.25	0.74
	16x16	512	3,840	3,840	3,840	12,032	6.00%	100	0.41	0.50	2.42
	25x25	1,250	15,000	15,000	15,000	46,250	3.90%	100	1.16	1.69	3.00
	36x36	2,592	45,360	45,360	45,360	138,672	2.73%	100	4.32	108.74	1,548.41
OOP Optimize 2 Constant	4x4	32	64	64	64	224	-	100	0.18	0.22	0.36
	9x9	162	729	729	729	2,349	-	100	0.21	0.26	0.61
	16x16	512	4,096	4,096	4,096	12,800	-	100	0.40	0.48	1.23
	25x25	1,250	15,625	15,625	15,625	48,125	-	100	1.16	1.86	7.86
	36x36	2,592	46,656	46,656	46,656	142,560	-	100	3.93	104.48	986.74
OOP Optimize 2 Parameter	4x4	32	64	64	64	224	-	100	0.17	0.23	0.53
	9x9	162	729	729	729	2,349	-	100	0.21	0.25	0.40
	16x16	512	4,096	4,096	4,096	12,800	-	100	0.39	0.52	2.21
	25x25	1,250	15,625	15,625	15,625	48,125	-	100	1.13	1.55	3.00
	36x36	2,592	46,656	46,656	46,656	142,560	-	100	4.12	108.94	1,070.65
OOP Optimize 3 Variable	4x4	32	24	24	24	104	53.57%	100	0.17	0.22	0.29
	9x9	162	324	324	324	1,134	51.72%	100	0.20	0.24	0.55
	16x16	512	1,920	1,920	1,920	6,272	51.00%	100	0.33	0.39	0.92
	25x25	1,250	7,500	7,500	7,500	23,750	50.65%	100	0.79	1.27	4.63
	36x36	2,592	22,480	22,480	22,480	70,632	50.45%	100	3.26	107.08	1,041.81
OOP Optimize 3 Constant	4x4	32	24	24	24	104	53.57%	100	0.17	0.22	0.40
	9x9	162	324	324	324	1,134	51.72%	100	0.20	0.24	0.47
	16x16	512	1,920	1,920	1,920	6,272	51.00%	100	0.32	0.39	0.91
	25x25	1,250	7,500	7,500	7,500	23,750	50.65%	100	0.82	1.25	5.24
	36x36	2,592	22,480	22,480	22,480	70,632	50.45%	100	3.12	104.32	1,060.59
OOP Optimize 3 Parameter	4x4	32	24	24	24	104	53.57%	100	0.17	0.23	0.58
	9x9	162	324	324	324	1,134	51.72%	100	0.21	0.28	1.55
	16x16	512	1,920	1,920	1,920	6,272	51.00%	100	0.32	0.40	1.84
	25x25	1,250	7,500	7,500	7,500	23,750	50.65%	100	0.76	1.27	6.58
	36x36	2,592	22,480	22,480	22,480	70,632	50.45%	100	2.76	114.09	1,340.97
FLAT Basic	4x4	32	48	48	48	176	-	100	0.18	0.28	1.36
	9x9	162	648	648	648	2,106	-	100	0.20	0.25	0.70
	16x16	512	3,840	3,840	3,840	12,032	-	100	0.41	0.58	2.79
	25x25	1,250	15,000	15,000	15,000	46,250	-	100	1.19	1.90	5.17
	36x36	2,592	45,360	45,360	45,360	138,672	-	100	12.61	137.72	746.75
FLAT Re-ordered	4x4	32	48	48	48	176	-	100	0.17	0.21	0.66
	9x9	162	648	648	648	2,106	-	100	0.21	0.25	0.35
	16x16	512	3,840	3,840	3,840	12,032	-	100	0.42	0.51	1.27
	25x25	1,250	15,000	15,000	15,000	46,250	-	100	1.34	2.11	5.70
	36x36	2,592	45,360	45,360	45,360	138,672	-	100	13.50	158.19	716.66
FLAT Array Reduction	4x4	32	24	24	24	104	40.91%	100	0.18	0.26	1.47
	9x9	162	324	324	324	1,134	46.15%	100	0.19	0.32	1.50
	16x16	512	1,920	1,920	1,920	6,272	47.87%	100	0.33	0.38	0.64
	25x25	1,250	7,500	7,500	7,500	23,750	48.65%	100	0.79	1.46	6.36
	36x36	2,592	22,680	22,680	22,680	70,632	49.07%	100	8.22	127.85	716.66
FLAT Unique	4x4	32	24	24	8	88	50.00%	100	0.17	0.22	0.41
	9x9	162	324	324	162	972	53.85%	100	0.19	0.22	0.51
	16x16	512	1,920	1,920	1,152	5,504	54.26%	100	0.32	0.42	1.55
	25x25	1,250	7,500	7,500	5,000	21,250	54.05%	100	0.80	1.61	7.58
	36x36	2,592	22,680	22,680	16,200	64,152	53.74%	100	8.60	148.51	1,138.90