Application of Formal Word-Level Analysis to Constrained Random Simulation (Tool Paper)

Hyondeuk Kim[1,2], Hoonsang Jin[2], Kavita Ravi[2], Petr Spacek[2], John Pierce[2], Bob Kurshan[2], and Fabio Somenzi[1]

[1] University of Colorado at Boulder
[2] Cadence Design Systems

Abstract. Constrained random simulation is supported by constraint solvers integrated within simulators. These constraint solvers need to be fast and memory efficient to maintain simulation performance. Binary Decision Diagrams (BDDs) have been successfully applied to represent constraints in this context. However, BDDs are vulnerable to size explosions depending on the constraints they are representing and the number of Boolean variables appearing in them. In this paper, we present a word-level analysis tool *DomRed* to reduce the number of Boolean variables required to represent constraints by reducing the domain of constraint variables. *DomRed* employs static analysis techniques to obtain these reductions. We present experimental results to illustrate the impact of this tool.

1 Introduction

Constrained random simulation is in increasing demand with hardware designers and verification engineers. As the name indicates, it is the simulation of a design under specified constraints. The user is required to capture the behavior of the environment of the design as constraints and the simulation tools simulate the design under these constraints with the aid of constraint solvers embedded in them. Commercial tools, such as Specman, have been popular for providing this capability. To address the need for constrained random simulation, modern hardware description languages (HDL), such as System Verilog, have incorporated constraint specification as part of their syntax.

The overwhelming benefit of constrained random simulation over the traditional writing of testbenches is the automation. Once the constraints are specified, the constraint solver in the simulator enumerates the valid scenarios instead of a human. Further, by specifying weights on the search space, the user can indicate whether the constrained space should be sampled uniformly or specific areas should be focused on.

Given that constraint solving comprises the bulk of constrained random simulation time, the efficiency and performance of constraint solvers is critical. Traditional constraint solving techniques, such as integer linear programming and constraint programming, far lag the performance of simulators. Boolean engines, e.g., BDDs, have been applied quite successfully to this problem[3] by taking advantage of the finite state nature of HDL constraints. More recently, Kitchen and Keuhlmann[2] have provided a word-level technique based on Markov-chain Monte Carlo methods. The scalability of this technique to industrial strength designs is yet to be proven.

In our constraint solver, *ValueGen*, we have incorporated both BDD and SAT-based Boolean engines. BDDs provide the advantage of fast generation of uniformly distributed solutions. However, some constraints have very large BDDs that cause memory explosion during simulation. SAT solvers are less vulnerable to size explosion. On the other hand, each solution generation could be exponentially slower than BDDs.

In this paper, we present a word-level pre-processor, *DomRed*, that *ValueGen* applies to the constraints to reduce the size of their representation in the Boolean engines. The pre-processing is a static analysis technique that uses an SMT-like framework. *DomRed* combines a SAT solver and a linear arithmetic solver that handles primarily integer difference logic, with a minor extension to positive and negative coefficient inequalities. The input to the tool is a Boolean combination of linear arithmetic constraints and bit-vector constraints. The output is a set of variables and their reduced domains. The constraints with reduced-domain variables are then passed on to the Boolean engines, resulting in smaller Boolean representations for constraint solving. We present experimental results of applying *DomRed* within *ValueGen* on our simulation testcases.

2 Constraint Solving in Simulation

Constraints are Boolean combinations of linear arithmetic and bit-vector expressions on design variables. The expressiveness of the specified constraints is limited by the HDL being used. For example, a System Verilog constraint is

```
constraint c1 {src_addr >= 0 && src_addr < 65536 &&
               payload_len >= 0 && payload_len < 4096 &&
               dest_addr - src_addr >= 4096 && dest_addr < 65536}
```

Constraint solving is the task of generating values for the design variables that satisfy the constraints. In the above example, `src_addr` $= 512$, `payload_len` $= 1024$, `dest_addr` $= 4608$ is a set of legal values. Our constraint solver, *ValueGen*, is invoked dynamically during simulation i.e., every time the simulator encounters a user call to generate new values for variables appearing in constraints, the simulator calls the constraint solver. Tight integration is required between the two to maintain efficiency.

Constraints are typically written on the inputs of the design and may depend on some internal design signals (*state* variables). During constraint solving, the solver is required to generate values that satisfy both the constraints as well as the states values.

Each set of related HDL constraints, when encountered, is parsed by the simulator, and sent to *ValueGen* through a word-level API along with the state values. Internally, *ValueGen* maintains a applies several optimizations at the word-level, including partitioning based on non-overlapping variable support and constant propagation. Finally, it bit-blasts the word-level constraints and calls the Boolean engines (BDD or SAT) on the Boolean representation.

The optimizations in *ValueGen* result from syntactic and very minor semantic analysis of the constraints. They do not include the ability to deduce that the tightest ranges of `dest_addr` and `src_addr` in the above example. *DomRed* addresses exactly this deficiency. It extracts a subset of invariants from semantic analysis of the constraints. If an invariant yields variable bound reductions, then the reduced number of bits are applied to encode the respective variables, the default number of bits are used otherwise.

3 *DomRed*: Technical Details

ValueGen provides *DomRed* with a quantifier-free first order logic formula with linear arithmetic constraints. An *LA* constraint is of the form $a_1x_1 + \ldots + a_nx_n \bowtie c$, where $\bowtie \in \{=, \leq, <, >, \geq, \neq\}$. A *difference* constraint is a special case of an LA constraint whose form is $x_i - x_j \bowtie c$. A positive-(negative-)inequality is another special case of an LA constraint where $\forall i.a_i \geq 0, x_i \geq 0, c \geq 0$ ($\forall i.a_i \leq 0, x_i \leq 0, c \leq 0$). We are working on the extension to bit-vector constraints.

As in the SMT-framework, the first order logic formula is abstracted conservatively into a propositional formula and given to the SAT solver. The SAT solver extracts a set of level-zero assignments, which corresponds to a set of LA constraints. From this set, we gather *difference* constraints, analyze them with the Bellman-Ford algorithm described in [1] and derive reduced bounds for the variable domains if possible. Among the LA constraints left over, positive- and negative-coeffient inequalites may also yield reduced upper (lower) bounds of x_i equal to c/a_i. The remaining LA constraints are conservatively marked as not yielding any domain reduction.

Example: Users commonly declare design inputs as `int`, meaning a 32-bit finite integer, causing the Boolean representation of the example in Section 2 to contain 96 bits. In applying *DomRed*, the equality constraint is translated into two inequalities in the usual manner. Inequalities are encoded with one bit each in the SAT solver. All these bits appear in the set of level-zero assignments. Since they all correspond to difference constraints, the Bellman-Ford algorithm yields the intervals $[0, 61439]$ for `src_addr`, $[0, 4095]$ for `payload_len` and $[4096, 65535]$ for `dest_addr`. The Boolean encoding will then require 16, 12 and 16 bits respectively, totalling 44 bits in the resulting Boolean expression (more than 2X reduction).

DomRed may also indicate to *ValueGen* that the constraints are infeasible (over-constrained situation) if the SAT solver or the LA solver detects it. This is of great value to *ValueGen* since it can avoid building the Boolean representations altogether.

4 Experimental Results

We integrated our tool *DomRed* into *ValueGen*, which, in turn, is integrated with our simulator. Our benchmark set includes both System-C and System Verilog examples. The System-C examples are smaller in size; 40 out of 68 showed improvements, the rest showed no degradation. The detailed table of results is not presented here for lack of space. The System Verilog examples consist of industrial-strength customer benchmarks. Of the 34 System Verilog examples that we experimented with, 11 showed improvement and are presented in Table1, the remaining 23 showed no degradation.

We use three parameters to measure the performance impact of applying *DomRed*—number of bits, CPU times and memory used. *ValueGen* switches between the BDD and SAT solver based on the Boolean representation size to maximize the size constraints

Table 1. Comparison Table of without and with Bound Reduction

Design	Sim. cycles	# of bits			CPU (sec)			MEM (Mbytes)		
		without	with	%	without	with	%	without	with	%
design1	5000000	112	101	10	683.0	549.4	**20**	40.8	34.2	**16**
design2	1000000	335	321	4	325.5	319.2	**2**	70.6	53.9	**24**
design3	50000	491	301	39	412.3	333.4	**19**	103.1	93.5	**9**
design4	1000000	54	40	26	180.9	174.1	**4**	37.2	37.8	-2
design5	1000000	64	60	6	86.1	44.0	**49**	33.2	33.6	-1
design6	1000000	64	60	6	75.9	48.1	**37**	33.2	33.7	-1
design7	1000000	16	14	12	340.2	344.6	-1	37.0	33.8	**9**
design8	44000	7	5	29	967.2	966.7	0	115.0	116.4	-1
design9	400000	8484	8428	1	607.1	559.6	**8**	62.3	62.0	0
design10	40	160	97	39	648.5	603.3	**7**	809.1	756.2	**7**
design11	2500	374	335	10	234.6	186.3	**21**	370.7	282.1	**24**

that can be solved and optimize the speed of constraint solving (better with BDDs). Our experimental results show the improvement over the default optimized algorithm. However, this makes comparing the Boolean representation sizes harder since different solvers may be used when *DomRed* is applied. We are working on addressing this problem to obtain a tighter comparison.

Column 1 of Table1 specifies the design, Column 2 shows the number of simulation cycles, Columns 3–5 show the reduction of the number of bits in the constraints. Note that the number of bits is measured for the constraints only and the design may have several thousand more bits. Columns 6–8 show the CPU times and Columns 9–11 the memory reduction. The time taken by *DomRed* is negligibly small and hence, not presented here. The CPU time includes simulation time only in 2/11 cases, hence the CPU time improvement for most examples is for constraint solving alone.

The table shows that the reduction in the number of bits is sometimes substantial, upto 39%. Smaller constraints yield better CPU times and memory reductions. Given that *DomRed* takes negligible time, 11/34 examples show improvement on applying *DomRed* and the remaining 23 examples are no worse off, we conclude that *DomRed* is a cheap preprocessing technique and that it is always beneficial to apply it. These results are encouraging and as part of future work, we hope to apply more powerful static analysis to reduce the size of the Boolean representation even further.

References

[1] H. Kim and F. Somenzi. Finite instantiations for integer difference logic. In *Formal Methods in Computer Aided Design (FMCAD'06)*, pages 31–38, San Jose, CA, Nov. 2006.

[2] N. Kitchen and A. Kuelhmann. Stimulus generation for constrainted random simulation. In *ICCAD*, 2007.

[3] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using bdds. In *ICCAD*, pages 584–590, 1999.