Constraint Synthesis for Environment Modeling in Functional Verification

Jun Yuan Ken Albin
Motorola Inc.
Austin, TX 78729
jun.yuan,ken.albin@mot.com

Adnan Aziz
University of Texas at Austin
Austin, TX 78712
adnan@ece.utexas.edu

Carl Pixley
Synopsys
Hillsboro, OR 97124
cpixley@synopsys.com

ABSTRACT

Modeling design environment with constraints instead of a traditional testbench is advantageous in a hybrid verification framework that encompasses simulation and formal verification. This movement is gaining popularity in industry and sparks research in the constraint-based environment modeling and stimulus generation problem. We present an approach, called *constraint synthesis*, to this problem. Constraint synthesis falls in the general category of parametric Boolean equation solving but is novel in utilizing don't care information unique to hardware constraints and heuristic variable removal to simplify the solution. Experimental results have demonstrated the effectiveness of the proposed approach.

Categories and Subject Descriptors

J.6 [**Computer-aided engineering**]: Verification

General Terms

Algorithm, Verification

Keywords

Constraint solving, Simulation vector generation

1. INTRODUCTION

Constraint-based verification is the idea of defining an environment for the Design Under Verification (DUV) by using constraints. These constraints can take several forms such as Boolean formulas whose variables reference inputs and state bits in the design or in auxiliary finite state machines, or in the form of temporal logic expressions. An environment is often called a testbench or bus functional model in conventional simulation. It is used to inject inputs into a design possibly reacting to the design's outputs or to monitor the outputs of the design. An environment is also necessary for a formal analysis of the design [3]. One of the key advantages of using constraints to model environments is its generation/monitor duality [8]. This duality means that the very same syntax can be used to monitor the interaction between designs and to drive inputs to a design fragment. Several commercial tools such as Vera and Verisity use constraints to help define a testbench [9]. In addition, Yuan *et al.* [13], Kukula and Shiple [4] and Shimizu and Dill [10] have presented algorithms to implement constraints as stimulus generators for simulation.

In our experience, sometimes hundreds of constraints are used to model the environment of a commercial DUV. This requires that the stimulus generator be able to handle high complexity. In addition, so as not to inordinately slow down simulation, the generator must solve the constraints every clock cycle very quickly, depending upon the value of the state-holding variables sampled from the DUV. Since the general SAT problem is known to be NP-hard, this stresses the constraint solving engine. One way to solve this problem is to build Binary Decision Diagrams for the conjunction of the constraints. To keep BDD sizes small, various techniques have been proposed, e.g, the *hold-constraint extraction* in [12] aimed at conjoining as few constraints as necessary, and the *range-preserving* simplification of constraints in [5].

The current paper shows a more efficient alternative to Yuan et al. [13] for stimulus generation from constraints. Although falling into the general category of parametric Boolean equation solving (e.g., [11, 7, 1, 2, 6]), this approach is novel in two aspects: first, it simplifies the solution by utilizing don't care information present in hardware constraints involving both input and state variables and in multi-level logic; second, further optimization is achieved by heuristically removing parametric variables. This approach is also related to Kukula and Shiple [4] in that it can be used to build a hardware circuit that emits correct inputs (if there are any) for any assignment of state variables.

The remainder of this paper is organized as follows: Section 2 introduces hardware design constraints and the problem of constructing *general (parametric) solutions* for Boolean functions. In Section 3, we present the main algorithm and the optimization techniques. We show experimental results in Section 4 and summarize in Section 5.

2. PRELIMINARIES

In this paper, we are concerned with the problem of solving constraints in the form of Boolean formulas that capture the interaction between a design and its environment. Take

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2003, June 2–6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006 ...$5.00.

a bus interface as an example: a typical environment constraint would be "the *transaction start* input (ts) is asserted only if the design is in the *address idle* state", or in formula

$$ts \rightarrow (\text{addr\_state} == \text{ADDR\_IDLE}).$$

Generally, environment constraints can be viewed as rules that relate the design's inputs to its state. Constraints depending on state information are general enough to describe a rich class of scenarios. For example, the temporal behavior of inputs can be modeled by constraints that use auxiliary variables to remember past states.

Our focus will be on constraint solving through the *synthesis* of constraints. Given a constraint over some input and state variables, constraint synthesis is the problem of finding a *general solution* to the inputs under all states for which a solution exists. Formally, let $f(X, Y)$ be the constraint of concern where $X = \{x_1, \ldots, x_n\}$ is the set of inputs and Y the state variables; the general solution to f is a substitution $\sigma = \{\sigma_1, \ldots, \sigma_n\}$ for X, such that for any state for which the equation $f = 1$ has a solution, $f(\sigma_1, \ldots, \sigma_n, Y)$ reduces to constant 1.

We call the states for which $f = 1$ has a solution the *legal states*. Whether a state s is legal or not can be quickly determined by checking if $s \in \exists_X f$. Having a general solution literally means *all* input vectors allowed under the current state are embodied by the solution. For this reason constraint synthesis provides a parametric solution and thus can be used to generate symbolic stimulus in formal verification. In contrast, the method in [13] is explicit and only applicable in traditional simulation.

3. CONSTRAINT SYNTHESIS

Constraint synthesis falls in the general category of Boolean equation solving, of which a representative approach called the *Boolean Unification* [11, 7] can be traced back to Boole himself. However, the result of Boolean Unification is not directly applicable in our case: we wish to consider don't cares unique to state-dependent hardware constraints and use them to optimize the general solution to the constraints. We also intend to minimize the variable count in the solution so as to further simplify the solution. In this section, we show how these objectives are achieved.

3.1 The Core Procedure

Let f be a constraint of input variables $X = \{x_1, \ldots, x_n\}$ and state variables Y. Denote the *projection* of f onto $\{x_i, \ldots, x_n, Y\}$ by f^i, which is given by the existential quantification $\exists_{x_1,\ldots,x_{i-1}} f$, for $1 \leq i \leq n$. Note the special case $f^1 = f$. It is not hard to see the following results:

1. Given a constraint f and a state, either all, or none, of the projections $f^1, \ldots, f^n$ are satisfiable.
2. Under a legal state, any input assignment satisfying f^i can be extended to an input assignment satisfying f^j, for all $j < i$.

Therefore, we can construct a solution for f by successively solving for inputs in the projections, starting from f^n which contains only x_n and the state variables, then applying the solution of x_n to f^{n-1}, in which the only inputs are x_n, x_{n-1}, and solving for x_{n-1}, and so on. Since $f^1 = f$, when we have solved x_1, we have also obtained a solution for f. So what's left to be decided is how we compute each input in a projection.

First we note the following decomposition of a function g over an orthogonal basis $\{g_x g_{\overline{x}}, g_x \overline{g_{\overline{x}}}, \overline{g_x} g_{\overline{x}}\}$:

$$g = g_x g_{\overline{x}} + g_x \overline{g_{\overline{x}}} x + \overline{g_x} g_{\overline{x}} \overline{x}. \tag{1}$$

Suppose a solution exists for $\exists_x g$. Denote it by a vector α. Then a satisfying assignment to x in g is chosen as

$$x = \begin{cases} u & \text{if } \alpha \in g_x g_{\overline{x}} \\ 1 & \text{if } \alpha \in g_x \overline{g_{\overline{x}}} \\ 0 & \text{if } \alpha \in \overline{g_x} g_{\overline{x}}. \end{cases} \tag{2}$$

This solution to x is *general* in that u, a free variable, has the options of assigning either 0 or 1 to x whenever it is allowed. However, an extra degree of freedom can be introduced if we consider the states where no input combinations are allowed by the constraint. For example, a designer may deem certain configurations of the control registers as illegal or undefined which should be warned against once they are realized by the design. As stated earlier, these states can be easily detected. Knowing this, we can use the don't care condition $\overline{g_x}\,\overline{g_{\overline{x}}}$ for optimization. We defer the details to the next subsection.

Returning to solving inputs in the projections, we see that the above analysis applies by corresponding x_i, the variable to be solved in f_i, to x in g, and f^{i+1} to $\exists_x g$ since $f^{i+1} = \exists_{x_i} f^i$. Therefore, according to Equation (2) and the don't care condition, we come to the following general solution for x_i:

$$\sigma_i(x_i) := (f^i_{x_i} f^i_{\overline{x_i}}) u_i + (f^i_{x_i} \overline{f^i_{\overline{x_i}}}) + (\overline{f^i_{x_i}}\,\overline{f^i_{\overline{x_i}}}) d_i. \tag{3}$$

Note u_i is the free variable introduced for x_i, and d_i can be any function that takes advantage of the don't care condition. Perceiving the σ_i's as logic circuits with inputs u_i's and outputs x_i's. A general solution to $f(x_1, \ldots, x_n, Y)$ can then be constructed as a multi-level and multi-output circuit that is the cascade of σ_i's according to the following rules:

1. x_1 is the output of σ_1
2. x_i is the output of σ_i, whose input u_j is connected to x_j for $1 \leq j < i$
3. all state inputs to σ_i for $1 \leq i < n$ are connected to the corresponding state lines from the design

Figure 1 gives an example of a 3-output solution circuit for a 3-input constraint, where $\sigma_1, \sigma_2, \sigma_3$ are the solutions, and u_1, u_2, u_3 and Y are the input and state variables, respectively. The following theorem summarizes the property of

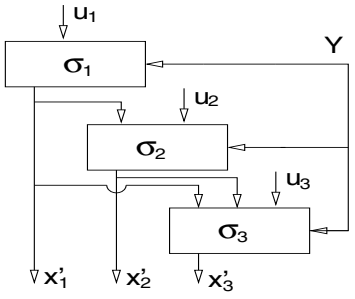

Figure 1: Example: Synthesis of a 3-input constraint

constraint synthesis we discussed so far.

THEOREM 1. *The solutions σ_i's connected as above defines a mapping $\sigma : B^n \mapsto B^n$, such that for any legal state and any pair of input vectors $\alpha, \alpha' \in B^n$ and $\sigma(\alpha) = \alpha'$, we have*

1. α' *satisfies* f, *and*
2. *if* α *satisfies* f, *then* $\alpha' = \alpha$.

3.2 Optimizations

We discuss two types of optimizations: one through the use of don't care information, the other with the elimination of variables introduced in the derivation of the general solution.

3.2.1 Don't Cares

The don't care optimization arises from the introduction of the term

$$\overline{f^i_{\overline{x_i}} f^i_{x_i}} d_i \tag{4}$$

in the solution for each input x_i in (3). The careset actually has two sources, the legal states and the limited input combinations observed by the general solutions σ_i's. For example, in Figure 1, the first two inputs of the circuit σ_3 are constrained by the outputs of σ_1 and σ_2.

We know for a constraint $f(X, Y)$ (recall X and Y are the input and respectively state variables) the legal state space is given by $\exists_X f$. In addition, under a legal state s, the only input patterns (excluding the free variable u_i) allowed at a solution σ_i are these satisfying the projection f^{i+1} under s, i.e., $\exists_{x_i} f^i|_s$. Overall, the careset over the input-state space is the product of the two which reduces to $\exists_{x_i} f^i$. The inversion of this, the don't care condition, equals the condition for d_i in (4). This is no accident but a result of the synthesis.

To take advantage of this careset information and to realize d_i, we simplify the solution in (3) by first setting $d_i = 1$ which gives

$$\sigma_i(x_i) := f^i_{x_i} u_i + \overline{f^i_{\overline{x_i}}}. \tag{5}$$

This reduction is intended to minimize intermediate BDD operations which tend to explode. Then we optimize the result again with respect to the careset $\exists_{x_i} f^i$ using the BDD *restrict* operator [1].

One may be tempted to *collapse* the solutions for simplification, for example, in Figure 1, by substituting σ_1 for the input to σ_2, then σ_1 and the new σ_2 for the inputs to σ_3. This is the case in standard Boolean Unification, such as the one in [7]. In a sense, this also realizes the input restriction we exploited above. But this *enforcement* of the restriction is to be distinguished from the optimization with respect to the restriction. Further, as suggested by experiment results, this recursive substitution tends to pose complexity problems in the BDD implementation.

3.2.2 Variable Removal

The problematic recursive substitution has one good effect: it eliminates the variables being substituted. However, the same effect can be achieved simply by recycling the input x_i whenever a free variable u_i is needed. This is sound due to the cascade of individual solutions shown in Figure 1, in which each u_i can be safely replaced by x_i.

Furthermore, the x_i's do not always have to be reintroduced. It is well known that function f is independent of variable x iff the *Boolean difference* $f_x \overline{f_{\overline{x}}} + \overline{f_x} f_{\overline{x}}$ (denoted by $\partial f/\partial x$) is 0. Applying this to the solution in (3), we have the following condition for the removal of x_i:

$$(f^i_{x_i} f^i_{\overline{x_i}} = 0) \bigwedge (\overline{f^i_{\overline{x_i}} f^i_{x_i}} (\partial d_i/\partial u_i) = 0). \tag{6}$$

```
 /* GetRset() returns variables satisfying (7) */
ConSynth(f, r_set) {
  if (f contains no input variables)
     return;
  if (r_set == nil)
     r_set = GetRset(f);
  if (r_set is empty) {
     let x be the input in f with
        the highest rank (in BDD var. ordering);
  else {
     let x ∈r_set be such that ∃_x f has the
        largest removable set;
     r_set = GetRset(∃_x f);
  }
  σ_x = x f_x + f_x̄;
  σ_x = BDD_Restrict(σ_x, ∃_x f);

  ConSynth(∃_x f, r_set);
}
```

Figure 2: Constrain synthesis.

As we have chosen d_i as 1 before the careset optimization, this condition can be simplified to

$$f^i_{x_i} f^i_{\overline{x_i}} = 0. \tag{7}$$

If this condition is met, the solution in (3) reduces to

$$\sigma_i(x_i) := \overline{f^i_{\overline{x_i}}}, \tag{8}$$

which is subjected to further optimization using the careset $\exists_{x_i} f^i$.

The satisfiability of the condition in (7) is determined by the constraint and the order in which the variables are solved. So instead of following the arbitrary order $x_n, \ldots, x_1$, we use the following heuristic to select a variable to solve in the current constraint projection f^i: first compute the *removable set* of variables that meet the condition in (7); if this set is not empty, choose from it a variable x such that $\exists_x f^i$ has the largest removable set; else, choose the highest ranked (in BDD variable ordering) input in f^i. Note that since the removable sets shrink monotonically (because for any variable ordering, $f^i_{x_i} f^i_{\overline{x_i}} = 0$ only if $f_{x_i} f_{\overline{x_i}} = 0$ initially) and they are relatively small comparing to the whole set of inputs, finding a good ordering incurs only an insignificant overhead.

3.3 The Overall Algorithm

The algorithm that combines synthesis and optimization is given in Figure 2. Initially, the argument for the removable set, *r_set*, is passed in as *nil*; *GetRset(f)* computes the removable set of f; and the general solution for input x is computed as σ_x.

4. EXPERIMENTAL RESULTS

We implemented the proposed algorithm and two related methods for evaluation. Examples with constraints are from real designs we collected. Each example has up to a few hundreds of constraints that involve from hundreds to over a thousand input and state variables. The actual numbers of variables in the examples range from a few to over 10 thousands.

	SimGen		*BU*		*Consynth*		
circuit	time	size	time	size	time	size	removed
des1	19.0	2105	t/o	–	29.0	3814	21/115
des2	27.0	2345	34.0	1599	30.0	822	238/423
des3	12.0	2437	14.0	1347	14.0	1217	6/174
des4	63.0	2847	88.0	15675	69.0	1521	11/134
des5	15.0	4679	92.0	16056	40.0	1641	15/63
des6	73.0	21822	t/o	–	81.0	8108	132/689
des7	237.0	28430	t/o	–	279.0	18480	23/207
des8	141.0	53788	t/o	–	169.0	9968	45/283

Table 1: Complexity of Solutions

circuit	*SimGen*	*Consynth*	speedup
des1	71.06	7.30	9.73
des2	34.66	11.68	2.97
des3	22.35	10.98	2.03
des4	14.70	2.18	6.74
des5	10.96	5.56	1.97
des6	181.72	6.74	26.96
des7	124.61	14.57	8.55
des8	219.4	9.53	23.02

Table 2: Speedup of Vector Generation

In Table 1, we compare the construction time (in seconds) and size of the BDDs (in number of BDD nodes) representing solutions from three methods: *SimGen* – the vector generation tool from [13], *BU* – the Boolean Unification method from [7], and *Consynth*, the method presented in the current paper. All three share the same flow including design compilation and constraint partitioning, and up to the construction of BDDs representing the constraints. *SimGen* ends after this stage, while *BU* and *Consynth* starts their respective constraint solving processes. As can be seen, in *Consynth*, the extra time spent in constraint solving is modest comparing to time for building the constraint BDDs, indicated by the time used by *SimGen*. Also, the final BDD size is reduced in all but one case. This should also apply to the comparison of our method vs. the method in [4] which would produce BDDs of the same size as those produced by *SimGen*. The classic *BU* approach, with recursive substitution and lacking the optimizations proposed, tends to generate larger BDDs and actually in half of the cases timed out (t/o) after 30 minutes.

The last column in the table reports the result of variable removal, where an entry m/n means m out of n input variables are removed from the final solution.

In Table 2, we report the results of using our method in vector generation in comparison with *SimGen*. The first two columns give the time (in seconds) spent by each method on vector generation in simulation. Each simulation runs for 10000 clock cycles. The third column reports the speedup using *Consynth*. We have already seen the reduction of BDD size in *Consynth*. But this is not the main factor that attributes to the order of magnitude improvement of the vector generation speed. When used as an explicit vector generator, *Consynth* uses time linear in the *number of variables* in the solution, whereas *SimGen* uses time linear in the *number of BDD nodes*.[1] This contrast is especially noticeable when BDD size gets large in both methods.

5. SUMMARY

We have described a method of solving hardware constraints by using Boolean equation solving techniques enhanced by don't care optimization and variable removal. We have shown improvements over similar approaches. In the case of input vector generation, we have achieved speedups of an order of magnitude in large designs.

Using constraints for environment modeling is an effective alternative to the traditional testbench approach in functional verification. Constraint synthesis facilitates the application of this methodology in areas where parametric or synthesizable constraint solutions are required, e.g., in model checking and emulation.

[1]Since state variables are assigned by the design, one cannot walk randomly down the BDD to generate vectors; instead, a bottom-up *weight* computation (in time linear to the BDD size) has to be performed to guide the walk [13].

6. REFERENCES

[1] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[2] M. Fujita, Y. Tamiya, Y. Kukimoto, and K.-C. Chen. Application of Boolean Unification to Combinational Logic Synthesis. *Proceedings of International Conference on Computer-Aided Design*, pages 510–513, 1991.

[3] M. Kaufmann, A. Martin, and C. Pixley. Design Constraints in Symbolic Model Checking. In *Proceedings of the Computer Aided Verification Conference*, 1998.

[4] J.H. Kukula and T.R. Shiple. Building Circuits From Relations. In *Proceedings of the Computer Aided Verification Conference*, 2000.

[5] H.H. Kwak, I.-H. Moon, J.H. Kukula, and T.R. Shiple. Combinational equivalence checking through function transformation. *Proceedings of International Conference on Computer-Aided Design*, pages 526–533, 2002.

[6] C-J. H. Segar M. D. Aagaard, R. B. Jones. Formal verification using parametric represenations of boolean constraints. *Proceedings of the Design Automation Conference*, 1996.

[7] U. Martin and T. Nipkow. Boolean unification - the story so far. In *Journal of Symbolic Computation*, volume 7, pages 275–293, 1989.

[8] C. Pixley. Integrating Model Checking Into the Semiconductor Design Flow. *Computer Design's Electronic Systems journal*, pages 67–74, March 1999.

[9] S. Regimbal, J-F. Lemire, Y. Savaria, G. Bois, E-M. Aboulhamid, and A. Baron. Applying Aspect-Oriented Programming to Hardware Verification with e. *Proceedings of HDLCON*, 2002.

[10] Kanna Shimizu and David Dill. Deriving a simulation input generator and a coverage metric from a formal specification. *Proceedings of the Design Automation Conference*, pages 801–806, June 2002.

[11] W. Büttner and H. Simonis. Embedding boolean expressions into logic programming. In *Journal of Symbolic Computation*, volume 4, pages 191–205, 1987.

[12] J. Yuan, A. Aziz K. Albin, and C. Pixley. Simplifying boolean constraint solving for random simulation-vector generation. *Proceedings of International Conference on Computer-Aided Design*, pages 123–127, 2002.

[13] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. *Proceedings of International Conference on Computer-Aided Design*, pages 584–589, 1999.