Simplifying Boolean Constraint Solving For Random Simulation-Vector Generation

Jun Yuan Ken Albin
Motorola Inc.
Austin, TX 78729
{jun.yuan,ken.albin}@motorola.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

We present an algorithm for simplifying the solution of conjunctive Boolean constraints of state and input variables, in the context of constrained random vector generation using BDDs. The basis of our approach is extraction of "hold-constraints" from constraint system. Hold-constraints are deterministic and trivially resolvable; in addition, they can be used to simplify the original constraints as well as refine the conjunctive partition. Experiments demonstrate significant reduction in the time and space needed for constructing the conjunction BDDs, and the time spent in vector generation during simulation.

1 Introduction

Constrained random vector generation [1, 7, 12] is an important task in simulation — the prevalent form of functional verification of commercial designs [6, 5]. In [12], Yuan *et al.* proposed a method of generating random vectors from a set of constraints, together with an optimization in which the constraints are partitioned into groups with disjoint input variable supports. In this work, we present a technique called *hold-constraint extraction* which is aimed at constraint simplification and further refinement of the partition. The technique is based on the observation that variables in hardware constraints are not homogeneous: state variables, unlike the inputs, are bounded by the design, and often some inputs can be fully specified under certain state valuations, regardless the values of the other inputs. We refer to a relation wherein the inputs only depend upon the state variables as a *hold-constraint*, and the inference thereof the *hold-constraint extraction*. The dichotomy comes from a more specific example of the mentioned dependency, which occurs frequently in writing constraints for hardware designs — under certain condition, an input variable maintains its value from the previous clock cycle, or is simply fixed to a constant. A hold-constraint is either instantly solved, assigning constants to its input variables, or discharged as a tautology. In addition, the disjoint-input-support partitioning can be further refined due to two facts about hold-constraints: (1) they do not need to be conjoined with any other constraints while being solved; and (2) they can be used to simplify the original constraints, and the result often contains fewer input variables. Experiments on several commercial designs demonstrated significant reduction in the time and space needed for constructing BDDs, and the time spent in vector generation during simulation.

The rest of the paper is structured as follows: Section 2 gives the preliminaries on hold-constraint extraction. In Section 3 we briefly describe a syntactical extraction algorithm. A procedure of complete functional extraction of hold-constraints and how they are used to simplify the original constraints are given in Section 4. The discussion of related works is deferred to Section 6. We report experimental results in Section 7 and summarize in Section 8.

2 Preliminaries

A constraint is a Boolean function of input variables X and state variables Y. The onset of a constraint represents the legal state-dependent input space regarding this constraint. The overall legal input space is the intersection of the onsets of all the constraints. In the sequel, we frequently use f, g, h, k, and e as function symbols, y as a state variable, s_i, s_j as states, i.e., minterms of state variables, S as a set of states, and x, v as input variables.

Definition 1 An input variable x in the support of f is *positively* (resp. *negatively*) *bounded* with respect to a set of states S if in all minterms in $f^{on} \cdot S$, it is the case that x evaluates to 1 (resp. 0).

More intuitively, x is *positively bounded* with respect to S if $f \rightarrow (S \rightarrow (x = 1))$, and x is *negatively bounded* with respect to S if $f \rightarrow (S \rightarrow (x = 0))$.

Definition 2 A *hold-constraint* on input variable x is a constraint $e(x, Y)$ in which x is positively or negatively bounded with respect to a nonempty set of states.

All hold-constraints can be written in the following *normal* form:

$$k \rightarrow (x = g) \tag{1}$$

where x is the input variable, and k, g are Boolean functions depending only on Y (the state variables), which we call the *condition* and *assignment* respectively. Note both k and g can be constants with the exception that k can not be 0.

We can infer a hold-constraint $k \rightarrow (x = g)$ from f iff the following *implication requirement* is met:

$$f \rightarrow (k \rightarrow (x = g)) \tag{2}$$

However, we immediately realize that an arbitrary hold-constraint would be able to meet the requirement as long as the condition k and the constraint f do not overlap. For a meaningful inference, we must also enforce the *nonvacuousness requirement*:

$$f \cdot k \neq 0 \tag{3}$$

Definition 3 A hold-constraint $k \rightarrow (x = g)$ is *extractable* from constraint f if the two satisfy the implication and nonvacuousness requirements.

In the coming sections, we present how hold-constraints can be extracted by both syntactical and functional means.

0-7803-7607-2/02/$17.00

3 Syntactical Extraction

Syntactical extraction consists of a conjunctive decomposition phase followed by a matching phase in which each of the resulting conjuncts is checked to see whether it transforms to a hold-constraint in the normal form. The following is the macro-code of the syntactical extraction algorithm.

1. Conjunctively decompose the constraint according to a set of syntactical rules
2. For each conjunct f:
   (a) If f does not match a pattern that is transformable to a disjunction $k+h$, skip f
   (b) If k depends on input variables, swap k and h
   (c) If k depends on input variables, skip f
   (d) If h matches a pattern that is transformable to $x=g$, where x is an input variable and g does not depend on input variables, then add $k \rightarrow (x=g)$ to the hold-constraint set

The above extraction satisfies the implication and nonvacuousness requirements. Its time and space complexities are both linear to the size of the constraint formulas. However, it is incomplete due to the limitation of its rule-based approach.

4 Functional Extraction

4.1 Condition and Extraction

We begin the description of a complete functional extraction algorithm with the introduction of *prime* hold-constraints.

Definition 4 A hold-constraint $k \rightarrow (x=g)$ is said to be *prime* if g is a constant, and the onset of k is a singleton, i.e., contains exactly one state.

The following lemma follows from Definitions 1 and 4.

Lemma 1 For any constraint f, for every input variable x and state s_i such that x is bounded in f with respect to $\{s_i\}$, the prime hold-constraint below is *extractable* from f.

$$s_i \rightarrow (x=b_i)$$

where b_i is 1 if x is positively bounded, and 0 otherwise.

The theorem below indicates that the conjunction of the prime hold-constraints obtained as in Lemma 1 on an input variable is a hold-constraint on the same variable, and the converse is also true.

Theorem 1 The conjunction of a set of prime hold-constraints on x with mutually exclusive conditions is a hold-constraint, and vice versa. That is,

$$\bigwedge_{i=1}^{l} \big(s_i \rightarrow (x=b_i)\big) \Leftrightarrow k \rightarrow (x=g) \tag{4}$$

where $s_i \wedge s_j = 0$ for $i \neq j$, $b_i \in \{0,1\}$; k and g can be derived from the prime hold-constraints as

$$k = \bigvee_{i=1}^{l} s_i,\ g \in [g^{on}, \overline{g^{off}}],\ g^{on} = \bigvee_{i=1}^{l} (b_i \cdot s_i),\ g^{off} = \bigvee_{i=1}^{l} (\overline{b_i} \cdot s_i), \tag{5}$$

and the prime hold-constraints can be derived from k and g as in the set

$$\{(s_i, b_i) \mid 1 \leq i \leq |k|,\ s_i \in k,\ b_i = g_{s_i}\}. \tag{6}$$

where $|k|$ is the number of minterms in the onset of k.

Because f implies and intersects with the condition of every prime hold-constraint on x as obtained in Lemma 1, it must also imply the conjunction of the said prime hold-constraint and intersect with the union of the said conditions. Therefore, if all of the prime hold-constraints are extractable from f, the hold-constraint conjunctively constructed in Theorem 1 is also extractable from f.

Now we derive the procedure that actually "computes" the construction in Theorem 1. By abuse of notation, we denote the *set difference* operator by "$-$", and the set of input variables $X - \{x\}$ by x'. The following is true for any constraint f:

1. $(\exists_{x'} f)_x - (\exists_{x'} f)_{\overline{x}}$ is the set of all the states with respect to which x is positively bounded
2. $(\exists_{x'} f)_{\overline{x}} - (\exists_{x'} f)_x$ is the set of all the states with respect to which x is negatively bounded

The union of the above two disjoint state sets is the Boolean differential denoted by $\partial(\exists_{x'} f)/\partial x$. The conjunction of the prime hold-constraints conditioned on this set is the *complete* hold-constraint on x, since it includes all the states for which x is bounded. The derivation of such a hold-constraint is formalized in the theorem below, which follows naturally from Theorem 1 and the above analysis.

Theorem 2 The complete hold-constraint of f on x is

$$k \rightarrow x = [g^{on}, \overline{g^{off}}]$$

where

$$k = \frac{\partial(\exists_{x'} f)}{\partial x},\ g^{on} = k \cdot (\exists_{x'} f)_x,\ g^{off} = k \cdot (\exists_{x'} f)_{\overline{x}}.$$

Obviously, this hold-constraint is nonvacuous iff

$$\frac{\partial(\exists_{x'} f)}{\partial x} \neq 0$$

Any function ψ, such that $\psi(g,c) \cdot c = g \cdot c$, can be used to select a g from the interval $[g^{on}, \overline{g^{off}}]$ for the above extraction. Note the careset $c = g^{on} + g^{off} = k$. We choose the BDD *Restrict* function [4, 10] since it is efficient and usually decreases the BDD size and does not introduce new variables from the careset to the result.

Note the careset for $x=g$ is also k due to the hold-constraint. However, since $k \leq \exists_X f$, function g does not need to be simplified with respect to f.

It is clear that the above extraction is unique with respect to x and f up to the selection of ψ. Although the extraction is complete, the BDD representation of k can still be optimized with regard to f using BDD *Restrict*. Since $k \leq \exists_X f$, and due to properties of BDD *Restrict*, the onset of k only increase in the optimization, and increment only comes from $\overline{f}$. Thus all the side effect is the addition of "vacuous" prime hold-constraints, which does not destroy the completeness property of the extraction.

4.2 Constraint Simplification

A hold-constraint can be used to simplify the constraint from which it is extracted by applying the *conditional substitution*, as defined below.

Definition 5 Let e be a hold-constraint $k \rightarrow (x=g)$. The *conditional substitution* of e on a Boolean function f, written $\tau(f,e)$, is

$$\tau(f,e) = k \cdot f_{x:=g} + \overline{k} \cdot f$$

where $f_{x:=g}$ is the substitution of variable x with function g.

Conditional substitution often simplifies a function and removes the variable being substituted. For example, let $f = y + x + v$ and $e := \overline{y} \to \overline{x}$. Then $\tau(f,e) = \overline{y} \cdot (y+v) + y \cdot (y+x+v) = y+v$. On the other hand, BDD *Restrict*, although widely used as a simplification function, is sensitive to variable ordering and may not improve the result: if x is the top variable, then restricting f on e returns f itself.

It turns out the conditional substitution is a don't care optimization function, just like BDD *Restrict*, but insensitive to variable ordering. It also possesses other nice properties:

Property 1: $\tau(f,e) \cdot e = f \cdot e$, i.e., $\tau(f,e)$ is a function equivalent to f in the careset e.

Property 2: $\tau(f,e)$ decreases the "diversity" of x in f, i.e., $\partial f/\partial x$, by the amount k. This implies that $\tau(f,e)$ is independent of x iff $k \geq \partial f/\partial x$

Property 3: If $\tau(f,e)$ is independent of x and $f \to e$, then $\tau(f,e) = \exists_x f$.

Property 4: If there exists a careset optimization function $\psi(f,e)$ that does not depend on x then it must be $\tau(f,e)$.

Note the last property makes conditional substitution a better choice than BDD *Restrict* in regard to input variable removal.

4.3 Recursive Extraction

Extracted hold-constraints can be used in extracting more hold-constraints which would otherwise be impossible. The complete and nonvacuous extraction for a set of constraints can be done using the procedure in Theorem 2 on the conjunction of the constraints. While we exclude conjoining the original constraints due to efficiency concerns, we can try to extract from the conjunction of the concerned constraint and the already extracted hold-constraints, whose size is usually small. In fact, we can even avoid explicitly conjoining a hold-constraint with a constraint due to the following theorem.

Theorem 3 For any hold-constraint $e := k \to (x = g)$, and Boolean function f, an input variable $v \neq x$ is extractable from $\tau(f,e)$ iff it is extractable from $f \cdot e$, or more precisely,

$$\frac{\partial\left(\exists_{v'} \tau(f,e)\right)}{\partial v} = \frac{\partial\left(\exists_{v'} (f \cdot e)\right)}{\partial v}$$

where $v' = X - \{v\}$.

The above theorem implies that, given a constraint and a hold-constraint, conditional substitution is an exact method for finding hold-constraints for input variables other than the one being substituted.

Take the same example from Section 4.2. Conditional substitution $\tau(f,e) = \overline{y} \cdot (y+v) + y \cdot (y+x+v)$ results in $y+v$, which is another hold-constraint. As expected from Theorem 3, the conjunction $f \cdot e = y + \overline{x} \cdot v$ yields the same two hold-constraints. Whereas $f \cdot e$ gives a function that is more complicated than $\tau(f,e)$, and BDD *Restrict* of f with respect to e returns f, which does not allow the extraction of the second hold-constraint.

Unfortunately, Theorem 3 cannot be extended to substitution of more than one hold-constraints. However, extraction preceded by multiple substitutions has shown in our experiments to be effective in extracting more hold-constraints.

5 The Overall Algorithm

The extraction and simplification algorithm is given in Figure 1. We always perform the syntactical extraction first because it is fast, and it simplifies the constraint formula prior to BDD building. The ensuing functional extraction is iterative. In the first iteration, the extraction is done for each constraint. Subsequently, if there are extractions from the last iteration, they are substituted in to find more extractions for the input variables that do not have an extraction yet. At the end of each iteration, the new extractions are used to simplify the remaining constraints. The procedure will terminate because the number of input variables in each constraint is finite.

```
extract(C) {
    E = currE = ∅;
    (C,E) = synt_extract(C);
    do {
        prevE = currE;
        currE = ∅;
        foreach f in C {
            f'=cond_subst(f, prevE);
            foreach not-yet-extracted input variable x in f' {
                e_x = func_extract(f', x);
                if (e_x ≠ nil) currE = currE ∪ {e_x};
            }
        }
        E = E ∪ currE;
        C = simplify(C, currE);
    } while (currE ≠ ∅)
}
```

Figure 1: Hold-constraint extraction

6 Related Works

The idea of extracting hold-constraints stemmed from our observation of real-life design constraints in which inputs are very commonly assigned values stored in memory elements. A syntactical extraction was the natural choice at the conception of this idea. The attempt on a functional extraction was inspired by the *state assignment extraction* work of Yang, Simmons, Bryant, and O'Hallaron [11]. The key result of their work is as follows:

Theorem 4 Let f be a Boolean formula, then

$$f \Leftrightarrow (x \in g) \cdot h$$

where x is a variable whose possible values are in the set L, and $h = \exists_x f$ and $g = \psi(t,h)$; ψ is a simplification function which uses the careset h to minimize t. The relation $t \subseteq h^{on} \times L$ is computed as:

$$\bigvee_{l \in L} \left(ITE(f|_{v \leftarrow l}, \{l\}, \emptyset)\right)$$

If t is also a partial function, i.e., each minterm in h^{on} correspondents to a unique value in L, then g is a function, and

$$f \Leftrightarrow (x = g) \cdot h$$

However, there is no distinction of state and input variables in their work and the assignments are unconditional. We attempted to modify the above approach to meet our needs in the "natural" way as given by the following theorem.

Theorem 5 Let f be a Boolean formula, $k = \overline{\forall_x f}$ and $h = \exists_x f$. Let ψ be a simplification function which uses the careset h to minimize t. Let $e = f \cdot k$ and $g = \psi(e_x, \exists_x e)$. Then

$$f \Leftrightarrow (k \to (x = g)) \cdot h.$$

We needed to make sure k and g do not depend on any input variables by applying don't care optimization. Even so, we failed to obtain some obviously extractable hold-constraints, for example, $y_1 + x_1$ in the constraint

$$f = (y_1 + x_1) \cdot (y_2 \cdot x_2 + x_1) \tag{7}$$

where y_1, y_2 are the state variables, and x_1, x_2 the inputs.

It turns out that the above method works only if f has a *conjunctive bi-decomposition* such that the intended input variable and the rest of the input variables belong to different conjuncts. This is an obvious limitation.

Bertacco and Damiani [2] proposed a method to build the decomposition tree for a Boolean function from its BDD representation. Their method has a similar restriction that the variable supports of the components be disjoint, and is therefore not suitable for our application.

An earlier work by McMillan [8] gives similar results as those of Yang *et al*. His method uses the BDD *constrain* operator, and can factor out *dependent variables* from Boolean functions. However, because the dependency is unconditional (i.e., in our case, for all state valuations), the method can not be adopted for a complete extraction, either. The example in (7) above also showcases the inability of this method to extract all hold-constraints.

It can be proved that a hold-constraint on x is extractable from f iff there exists a conjunctive bi-decomposition of f such that one conjunct depends on x and some state variables, and the other can depend on all the variables. Our test in Theorem 2 detects exactly such a decomposition.

The closest works on similar decompositions are those of Bochmann, Dresig, and Steinbach [3], and Mishchenco, Steinbach, and Perkowski [9], from the logic synthesis and optimization community. They proposed a set of criteria for various *groupabilities*, including one that tests whether two (disjoint) groups of variables can be separated in a conjunctive bi-decomposition. This seemed to match our need of checking if an input variable can be factored out from a constraint. However, the grouping also depends on a third group of variables that is shared by the conjuncts. In our case, this can be any subset of the set of state variables. Therefore, it can take multiple (in the worst case, exponential to the number of state variables) groupability tests to decide if a hold constraint exists for an input variable. In contrast, our approach needs just one test.

7 Experiments

7.1 Impact on building conjunction BDDs

The experiments are intended to compare the effect of hold-constraint extraction on building BDDs for the partitioned constraints. Six commercial designs are used in the experiments. Four configurations are compared, namely

no-extraction:	with no extraction
syntactical:	with the syntactical extraction
functional1:	with the nonrecursive functional extraction
no-extraction:	with the recursive functional extraction.

Table 1 demonstrates the effect of the three types of extractions. For this experiment only, the functional extractions are run without first applying syntactical extraction in order to perform the sanity check that the former always subsumes the latter. Columns 1 and 2 give the numbers of constraints and input variables of the designs, respectively. The *#e_c* and *#e_i* columns give the numbers of constraints and input variables with extractions, respectively. As can be seen, the functional extractions always perform better, and a recursive extraction is more powerful than a nonrecursive extraction.

Tables 2 through 5 compare the results of building BDDs for the partitioned constraints. The reported times and BDD node counts include times and BDD nodes used in extraction. Dynamic variable reordering is enabled in all examples except in *rio*, where a fixed order is used to avoid BDD blowup. Table 2 shows the results of building conjunction BDDs without any extraction. Column 1 is the number of BDD conjunction operations performed during partitioning. Column 2 shows the number of resulting parts. Column 3, 4, and 5 show the peak number of BDD nodes, number of BDD nodes in the result, and the time for building the BDDs, respectively.

	circuit stats		syntactical		functional1		functional2	
circuit	#cons	#input	#e_c	#e_i	#e_c	#e_i	#e_c	#e_i
mmq	117	207	18	25	42	49	77	53
qbc	93	174	58	169	75	174	75	174
qpag	215	283	149	282	173	282	197	283
qpcu	109	34	75	34	93	34	93	34
rio	198	371	80	283	89	289	96	292
sbs	108	423	95	422	96	423	97	423

Table 1: Result of extractions

circuit	#conj	#part	peak	result	time
mmq	92	26	82782	24535	17.0
qbc	60	34	10220	2689	2.5
qpag	187	29	968856	142943	272.4
qpcu	94	11	14308	4563	1.0
rio	133	66	1299984	375723	3.3
sbs	69	40	12264	2940	1.5

Table 2: Building conjunction BDDs (*no extraction*)

The impact of extractions on BDD building is shown in Table 3, 4 and 5, respectively for *syntactical*, *functional1* and *functional2*. First, it should be clarified that the sharp increase of number of parts between the functional and the syntactical extractions is partially due to the fact that the former extracts signals bit by bit, while the latter can extract bus signals at once. Therefore, the effect of partition refinement is more proportionally represented in the number of conjunctions performed among the simplified versions of the original constraints (note we do not conjoin the hold-constraints). It can be seen that as the extraction gets more powerful, the number of parts increases. Although the number of conjunctions decreases quickly in *syntactical* and more in *functional1*, we observed that *functional2* does not improve the number further, although it has the most extractions. As a result, BDD sizes in *functional2* increase slightly over those in *functional1*. Obviously, simplification from the extra extractions in *functional2* in our examples did not remove more input variables to refine the partition, although in theory it could. Overall, functional extractions have large improvement over its syntactical counterpart. In all examples, extracting hold-constraints has a clear advantage in time (up to 23 times faster) and space (up to 7 times smaller) usages over not extracting such constraints.

7.2 Impact on Simulation

Simulation directly benefits from finer partitions which usually result in smaller conjunction BDDs. Furthermore, hold-constraints by themselves also contribute to the speedup of vector generation since they can produce quick solution to input variables. Table 6 shows the impact of the extractions on simulation. Each design is simulated three times, with runs of 1000 cycles, using randomly generated inputs from the conjunction BDDs. Nonrecursive extraction was used in this experiment. Column 1 and 2 report the average times spent in simulation generation from BDDs with and

circuit	#conj	#part	peak	result	time
mmq	80	37	32704	9501	8.9
qbc	19	77	9198	1754	0.1
qpag	65	156	155344	89195	61.4
qpcu	24	84	4088	983	0.0
rio	97	127	1040396	149152	1.5
sbs	10	104	15330	1663	0.1

Table 3: Building conjunction BDDs (*syntactical*)

circuit	#conj	#part	peak	result	time
mmq	72	58	33726	9529	6.2
qbc	10	92	5110	1294	0.0
qpag	60	216	47012	20613	12.3
qpcu	17	136	4088	801	0.0
rio	94	140	1072460	142997	1.5
sbs	10	503	34748	1612	0.1

Table 4: Building conjunction BDDs (*functional1*)

circuit	#conj	#part	peak	result	time
mmq	72	58	33726	9529	6.2
qbc	10	113	6132	1428	0.0
qpag	60	268	79716	28221	26.7
qpcu	17	146	13286	1042	0.0
rio	94	144	1226400	143071	1.6
sbs	10	528	49056	1812	0.1

Table 5: Building conjunction BDDs (*functional2*)

circuit	without extract	with extraction	speedup
mmq	6.98	4.57	1.53
qbc	1.55	1.24	1.25
qpag	6.46	2.57	2.51
qpcu	0.49	0.55	0.89
rio	302.63	132.00	2.30
sbs	2.12	1.98	1.07

Table 6: Impact on simulation generation

without extraction, respectively. Column 3 gives the ratio of generation time without extract to that with extraction. The speedup is more proportional to the reduction in BDD size when the conjunction BDDs get more complex, for example, in *rio*, *qpag* and *mmq*. For smaller BDDs, the overhead of handling finer partitions may offsets the size reduction, which results in increase of generation time in *qpcu*. Nonetheless, our concern is more with the long generation time from large conjunction BDDs, in which cases we achieved a speedup ratio of about 2.5.

8 Summary

We have presented a method for simplifying constraint solving in random simulation-vector generation. The source of the simplification is the refining of constraint partition by extracting deterministic assignments to input variables. The result is a faster construction and smaller size of BDD representation of the constraints. Simulation vector generation time is also reduced due to the smaller BDD size, and the constant assignments from the fast solution of hold-constraints.

References

[1] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd. Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator. *IBM Systems Journal*, 30(4):527–538, July 1991.

[2] V. Bertacco and M. Damiani. The Disjunctive Decomposition of Logic Functions. *Proc. Intl. Conf. on Computer-Aided Design*, pages 78–82, 1997.

[3] D. Bochmann, F. Dresig, and B. Steinbach. A New Decomposition Method for Multilevel Circuit Design. *Proc. European Design Automation Conf.*, pages 374–377, 1991.

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

[5] S. Devadas, A. Ghosh, and K Keutzer. An observablity-based code coverage metric for functional simulation. *Proc. of the Design Automation Conf.*, pages 418–425, 1996.

[6] Daniel Geist et al. A Methodology For the Verification of a "System On Chip". *Proc. of the Design Automation Conf.*, pages 574–579, 1999.

[7] J. Freeman, R. Duerden, C. Taylor, and M. Miller. The 68060 microprocessor functional design and verification methodology. In *On-Chip Systems Design Conference*, pages 10.1–10.14, 1995.

[8] K. L. McMillan. A conjunctively decomposed boolean representation for symbolic model checking. *Proc. of the Computer Aided Verification Conf.*, 1996.

[9] A. Mishchenco, B. Steinbach, and M. Perkowski. An Algorithm for Bi-Decomposition of Logic Functions. *Proc. of the Design Automation Conf.*, 2001.

[10] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic Minimization of BDDs Using Don't Cares. In *Proc. of the Design Automation Conf.*, San Diego, CA, June 1994.

[11] B. Yang, R. Simmons, R.R. Bryant, and D.R. O'Hallaron. Optimizing Symbolic Model Checking for Constraint-rich Models. *Proc. of the Computer Aided Verification Conf.*, 1999.

[12] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. *Proc. Intl. Conf. on Computer-Aided Design*, 1999.