Interpolating Functions from Large Boolean Relations

Jie-Hong R. Jiang, Hsuan-Po Lin, and Wei-Lun Hung

Department of Electrical Engineering/Graduate Institute of Electronics Engineering
National Taiwan University, Taipei 10617, Taiwan

ABSTRACT

Boolean relations are an important tool in system synthesis and verification to characterize solutions to a set of Boolean constraints. For physical realization as hardware, a deterministic function often has to be extracted from a relation. Prior methods however are unlikely to handle large problem instances. From the scalability standpoint this paper demonstrates how interpolation can be exploited to extend determinization capacity. A comparative study is performed on several proposed computation techniques. Experimental results show that Boolean relations with thousands of variables can be effectively determinized and the extracted functional implementations are of reasonable quality.

1. INTRODUCTION

Relations are a powerful tool to represent mappings. Admitting one-to-many mappings, they are strictly more generic than functions. Taking the Boolean mappings $\{(x_1, x_2) \in \mathbb{B}^2\} \to \{(y_1, y_2) \in \mathbb{B}^2\}$ of Figure 1 as an example, we can express the (one-to-one) mapping of (a) using Boolean functions $f_1 = x_1x_2$ and $f_2 = \neg x_1 \neg x_2$ for outputs y_1 and y_2, respectively. On the other hand, there is no similar functional representation for the mapping of (b) due to the one-to-many mapping under $(x_1, x_2) = (0, 1)$. However, this mapping can be specified by the relation (with characteristic function) $R = \neg x_1 \neg x_2 \neg y_1 y_2 \vee \neg x_1 x_2 \neg y_1 \neg y_2 \vee \neg x_1 x_2 y_1 y_2 \vee x_1 \neg x_2 \neg y_1 \neg y_2 \vee x_1 x_2 y_1 \neg y_2$.

Owing to their generality, relations can be exploited to specify the permissible behavior of a design. For instance, the behavior of a system can be specified using relations as constraints over its input stimuli, state transitions, and output responses. Moreover, the flexibility of a circuit can be naturally characterized by a relation. In fact, relations subsume the conventional notion of don't-cares. To see it, assume Figure 1 (b) to be a relaxed permissible mapping of (a). That is, under input $(x_1, x_2) = (0, 1)$ the output (y_1, y_2) can be $(1, 1)$ in addition to $(0, 0)$. This flexibility is not expressible using the conventional don't-care representation, and it can be useful in circuit optimization. By trimming off the output choice $(0, 0)$ under input $(0, 1)$, the resulting mapping in (c) has new output functions $f_1 = x_2$ and $f_2 = \neg x_1$, simpler than those of (a).

Compared with relations, functions, though more restrictive, are often closer to physical realization due to their deterministic nature. Therefore conversions between relations and functions are usually indispensable. To name two examples, in reachability analysis, the transition functions of a state transition system are often converted to a transition relation to abstract away input variables; in circuit synthesis, optimal functions can be extracted from a relation representing some specification or permissible behavior.

Whereas converting functions to relations is straightforward, converting relations to functions involves much complication. Among many possibilities, relation-to-function conversion in its two extremes can be *range-preserving*, where all possible output responses under every input stimulus[1] are produced (with the help of parametric variables), and can be *deterministically reducing*, where only one output response under every input stimulus is produced (without the need of parametric variables). Both extremes have important applications in system synthesis and verification. The former is particularly useful in verification. As the constraints specified by a relation are preserved, the conversion helps create a testbench to generate simulation stimuli [10, 22, 21] mimicking the constrained system environment. The later on the other hand is particularly useful in synthesis [4]. As the synthesized components are typically much compact than those with range preserved [2], it is attractive for generating the final implementation. This paper is concerned with the latter, in particular, determinizing a relation and extracting functional implementation in the Boolean domain.

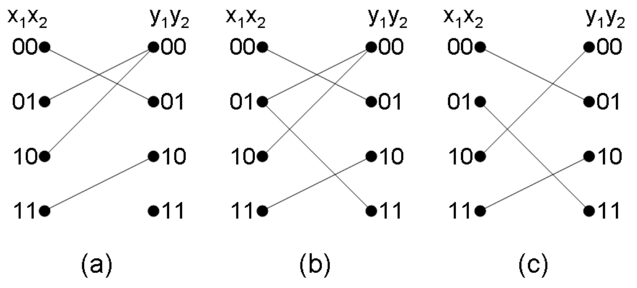

Figure 1: Boolean mappings.

Brayton and Somenzi [4] were among the first to observe the utility of Boolean relations in logic minimization. Boolean relations are useful not only in characterizing circuit flexibilities [20, 15], but also in characterizing solutions to design specifications/constraints [2]. There were intensive efforts focusing on the exact or heuristic optimization of functions implementing a given relation. Such optimization objectives, for instance, can be in terms of two-level logic minimization under the SOP representation [5, 7, 14, 9, 19], or in term of some polynomial functions over BDD sizes [1]. State-of-the-art methods, such as [19, 1], are based on decision diagrams. As BDDs are not memory efficient in representing large Boolean functions, these methods have intrinsic limitations and are not scalable to large problem instances. There has been growing need for scalable determinization of large Boolean relations. Synthesis from specifications shows one such example [2]. The quest for scalable determinization methods remains.

From the scalability standpoint, we seek reasonable representation of large Boolean functions and, in particular, use

[1]In some occasions input stimuli are unimportant and their correspondences with output responses need not be preserved.

and-inverter graphs (AIGs) (see, e.g., [17]) as the underlying data structure. Due to their simple and multi-level nature, AIGs are often much compact and are closer to final logic implementation than the two-level SOP form. Moreover they provide a convenient interface with SAT solvers in terms of conversion to CNFs and representation of interpolants [18]. Therefore, unlike previous efforts on relation solving, our objective is to convert a large relation to a set of functions with reasonable quality. Similar attempts were pursued recently in other efforts of scalable logic synthesis, e.g., [11, 12, 13, 16].

Our main exploration includes 1) exploiting *interpolation* [6] for Boolean relation determinization and function extraction, 2) studying expansion- and substitution-based quantifications with reuse, and 3) showing support minimization in the interpolation framework. A comparative empirical study is performed on various computation schemes. Experimental results suggest that interpolation is essential to scalable relation determinization and function extraction. Boolean relations with thousands of variables can be determinized effectively and the extracted functions are typically of reasonable quality compared with their respective reference models.

This paper is organized as follows. Essential backgrounds are given in Section 2. Section 3 presents the main results on relation determinization and function extraction; Section 4 discusses function simplification. Experimental results and discussions are given in Section 5. Finally, Section 6 concludes the paper and outlines some future work.

2. PRELIMINARIES

As a notational convention, substituting function g for variable v in function f is denoted as $f[v/g]$.

2.1 Boolean Relation

A relation $R \subseteq X \times Y$ can be equivalently represented as a characteristic function $R : X \times Y \to \mathbb{B}$ such that $(a, b) \in R$ for $a \in X, b \in Y$ if and only if $R(a, b) = 1$.

Definition 1. *A relation $R : X \times Y \to \mathbb{B}$ is* total *(in X) if $\forall a \in X, \exists b \in Y.R(a, b) = 1$. Otherwise, R is* partial.

Unless otherwise said we shall assume that a relation $R \subseteq X \times Y$ is total in X.

Definition 2. *Given a partial relation $R : X \times Y \to \mathbb{B}$, an (input) assignment $a \in X$ is* undefined *if no (output) assignment $b \in Y$ makes $R(a, b) = 1$.*

This paper assumes that X is the *input space* $\mathbb{B}^n$ spanned by *input variables* $\vec{x} = (x_1, \dots, x_n)$ and Y is the *output space* $\mathbb{B}^m$ spanned by *output variables* $\vec{y} = (y_1, \dots, y_m)$.

Given a Boolean relation $R : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}$ with input variables $\vec{x} = (x_1, \dots, x_n)$ and output variables $\vec{y} = (y_1, \dots, y_m)$, we seek a *functional implementation* $\vec{f} = (f_1, \dots, f_m)$ with $f_i : \mathbb{B}^n \to \mathbb{B}$ such that

$$D = \bigwedge_{i=1}^{m} (y_i \equiv f_i(\vec{x}))$$

is contained by R, i.e., the implication $D \Rightarrow R$ holds. Equivalently, the relation after substituting f_i for y_i

$$R[y_1/f_1, \dots, y_m/f_m]$$

equals constant 1.

Note that the above relation D is a *deterministic* relation, i.e.,

$$\forall a \in X, \forall b, b' \in Y.((D(a, b) \wedge D(a, b')) \Rightarrow (b = b')).$$

Therefore seeking a functional implementation of a total relation can be considered as determinizing the relation. On the other hand, any deterministic total relation has a unique functional implementation.

2.2 Satisfiability and Interpolation

The reader is referred to prior work [11] for a brief introduction to SAT solving and circuit-to-CNF conversion, which are essential to our development. To introduce terminology and convention for later use, we restate the following theorem.

Theorem 1 (Craig Interpolation Theorem). *[6] Given two Boolean formulas ϕ_A and ϕ_B, with $\phi_A \wedge \phi_B$ unsatisfiable, then there exists a Boolean formula ψ_A referring only to the common variables of ϕ_A and ϕ_B such that $\phi_A \Rightarrow \psi_A$ and $\psi_A \wedge \phi_B$ is unsatisfiable.*

The Boolean formula ψ_A is referred to as the *interpolant* of ϕ_A with respect to ϕ_B. Modern SAT solvers can be extended to construct interpolants from resolution refutations [18].

In the sequel, we shall assume that Boolean relations, functions, and interpolants are represented using AIGs.

3. RELATIONS TO FUNCTIONS

3.1 Single-Output Relation

We consider first the functional implementation of a single-output relation $R(\vec{x}, y)$ with y the only output variable.

3.1.1 Total Relation

Proposition 1. *A relation $R(\vec{x}, y)$ is total if and only if the conjunction of $\neg R(\vec{x}, 0)$ and $\neg R(\vec{x}, 1)$ is unsatisfiable.*

Theorem 2. *Given a single-output total relation $R(\vec{x}, y)$, the interpolant ψ_A of the refutation of*

$$\neg R(\vec{x}, 0) \wedge \neg R(\vec{x}, 1) \quad (1)$$

with $\phi_A = \neg R(\vec{x}, 0)$ and $\phi_B = \neg R(\vec{x}, 1)$ corresponds to a functional implementation of R.

Proof. Since R is total, Formula (1) is unsatisfiable by Proposition 1. That is, the set $\{a \in X \mid R(a, 0) = 0 \text{ and } R(a, 1) = 0\}$ is empty. Hence ϕ_A (respectively ϕ_B) characterizes the set $S_A = \{a \in X \mid R(a, 1) = 1 \text{ and } R(a, 0) = 0\}$ (respectively $S_B = \{a \in X \mid R(a, 0) = 1 \text{ and } R(a, 1) = 0\}$). As $\phi_A \Rightarrow \psi_A$ and $\psi_A \Rightarrow \neg\phi_B$, the interpolant ψ_A maps every element of S_A to 1, every element of S_B to 0, and every other element to either 0 or 1. Let D be $(y \equiv \psi_A)$. Then $D \Rightarrow R$. ■

Therefore interpolation can be seen as a way to exploit flexibility for function derivation without explicitly computing don't cares.

Corollary 1. *Given a single-output total relation R, both $R(\vec{x}, 1)$ and $\neg R(\vec{x}, 0)$ are legitimate functional implementation of R.*

Proof. Let $\phi_A = \neg R(\vec{x}, 0)$ and $\phi_B = \neg R(\vec{x}, 1)$. Because $\phi_A \Rightarrow R(\vec{x}, 1)$ and $R(\vec{x}, 1) \Rightarrow \neg\phi_B$, $R(\vec{x}, 1)$ is a legitimate interpolant. Similarly, $\neg R(\vec{x}, 0)$ is a legitimate interpolant, too. ■

In fact, the cofactored relations $R(\vec{x}, 1)$ and $\neg R(\vec{x}, 0)$ are the largest (weakest) and smallest (strongest) interpolants, respectively, in terms of solution spaces. Therefore to derive a functional implementation of a single-output total relation, interpolation is unnecessary. However practical experience suggests that functional implementations obtained through interpolation are often much simpler in AIG representation.

3.1.2 Partial Relation

Note that Theorem 2 works only for *total* relations because *partial* relations make Formula (1) satisfiable. To handle partial relations, we treat undefined input assignments as don't-care conditions (this treatment is legitimate provided that the undefined input assignments can never be activated) and define complete totalization as follows.

Definition 3. *Given a single-output partial relation R, its* complete totalization *is the new relation*

$$T(\vec{x}, y) = R(\vec{x}, y) \vee \forall y. \neg R(\vec{x}, y). \qquad (2)$$

Note that $T = R$ if and only if R is total.

Accordingly Theorem 2 is applicable to a totalized relation T with

$$\begin{aligned} \phi_A &= \neg T(\vec{x}, 0) \text{ and} && (3)\\ \phi_B &= \neg T(\vec{x}, 1), && (4) \end{aligned}$$

which can be further simplified to

$$\begin{aligned} \phi_A &= \neg R(\vec{x}, 0) \wedge R(\vec{x}, 1) \text{ and} && (5)\\ \phi_B &= \neg R(\vec{x}, 1) \wedge R(\vec{x}, 0). && (6) \end{aligned}$$

Observe that the conjunction of Formulas (5) and (6) is trivially unsatisfiable. Further, either of $\neg R(\vec{x}, 0)$ and $R(\vec{x}, 1)$ is a legitimate interpolant. Therefore, as long as the undefined input assignments of R are never activated, the interpolant is a legitimate functional implementation of R. (This fact will play a role in the development of Section 3.2.2.)

Given a (partial or total) relation R with y being the only output variable, in the sequel we let $FI(y, R)$ denote a functional implementation of y with respect to R. Among many possibilities, $FI(y, R)$ can be derived through the interpolation of Formulas (5) and (6).

3.2 Multiple-Output Relation

We now turn attention to the functional implementation of a multiple-output relation $R(\vec{x}, y_1, \ldots, y_m)$ with $m > 1$. In essence, we intend to reduce the problem so as to apply the previous determinization of single-output relations.

A determinization procedure contains two phases: The first phase reduces the number of output variables; the second phase extracts functional implementation. We study two determinization procedures with different ways of reducing the number of output variables. One is through existential quantification; the other is through substitution.

3.2.1 Determinization via Expansion Reduction

As a notational convention, we let $R^{(i)}$ denote $\exists y_m, \ldots, y_i.R$ for $1 \leq i \leq m$. Through standard existential quantification by formula expansion, i.e., $\exists x.\varphi = \varphi[x/0] \vee \varphi[x/1]$ for some formula φ and variable x, one can reduce a multiple-output relation R to a single-output relation $R^{(2)}$.

In the first phase, $R^{(i)}$ can be computed iteratively as follows.

$$\begin{aligned} R^{(m)} &= \exists y_m.R\\ &\vdots\\ R^{(i)} &= \exists y_i.R^{(i+1)}\\ &\vdots\\ R^{(2)} &= \exists y_2.R^{(3)} \end{aligned}$$

for $i = m-1, \ldots, 2$.

In the second phase, functional implementations of all output variables can be obtained through the following iterative calculation.

$$\begin{aligned} f_1 &= FI(y_1, R^{(2)})\\ &\vdots\\ f_i &= FI(y_i, R^{(i+1)}[y_1/f_1, \ldots, y_{i-1}/f_{i-1}])\\ &\vdots\\ f_m &= FI(y_m, R[y_1/f_1, \ldots, y_{m-1}/f_{m-1}]) \end{aligned}$$

for $i = 2, \ldots, m-1$.

The above procedure is similar to prior work (see, e.g., [19]) with some subtle differences: First, the quantification results of the first phase are reused in the second-phase computation. It reduces the number of quantifications from $O(m^2)$ to $O(m)$. Second, interpolation is the key element in the computation and AIGs are the underlying data structure.

3.2.2 Determinization via Substitution Reduction

Alternatively the solution to the determinization of a single-output relation can be generalized as follows. Each time we treat all except one of the output variables as the input variables. Thereby we see a single-output relation rather than a multiple-output relation. For example, let y_m be the only output variable and treat $y_1, \ldots, y_{m-1}$ be additional input variables. In the enlarged input space (spanned by $y_1, \ldots, y_{m-1}$ as well as $\vec{x}$), however, R may not be total even though it is total in the original input space X. Let $f'_m = FI(y_m, R)$, obtained through interpolation mentioned in Section 3.1.2. Note that since f'_m depends not only on $\vec{x}$, but also on $y_1, \ldots, y_{m-1}$, it is not readily a functional implementation of y_m.

In the first phase, the number of output variables can be iteratively reduced through the following procedure.

$$\begin{aligned} f'_m &= FI(y_m, R)\\ R^{\langle m\rangle} &= R[y_m/f'_m]\\ &\vdots\\ f'_i &= FI(y_i, R^{\langle i+1\rangle})\\ R^{\langle i\rangle} &= R^{\langle i+1\rangle}[y_i/f'_i]\\ &\vdots\\ f'_2 &= FI(y_2, R^{\langle 3\rangle})\\ R^{\langle 2\rangle} &= R^{\langle 3\rangle}[y_2/f'_2] \end{aligned}$$

for $i = m-1, \ldots, 2$.

In the second phase, the functional implementations can be obtained through the following iterative calculation.

$$\begin{aligned} f_1 &= FI(y_1, R^{\langle 2\rangle})\\ &\vdots\\ f_i &= FI(y_i, R^{\langle i+1\rangle}[y_1/f_1, \ldots, y_{i-1}/f_{i-1}])\\ &\vdots\\ f_m &= FI(y_m, R[y_1/f_1, \ldots, y_{m-1}/f_{m-1}]) \end{aligned}$$

for $i = 2, \ldots, m-1$.

The following fact can be shown.

Lemma 1. *[8] Given a relation R and $f'_m = FI(y_m, R)$, the equality $R[y_m/f'_m] = \exists y_m.R$ holds.*

It may be surprising, at first glance, that any $f'_m = FI(y_m, R)$ results in the same $R[y_m/f'_m]$. This fact is true however and a detailed exposition can be found in the work [8]. By induction on $i = m, \ldots, 2$ using Lemma 1, one can further claim that $R^{\langle i \rangle} = R^{(i)}$.

Note that the above computation implicitly relies on the don't-care assumption of partial relations. This assumption is indeed legitimate because the don't cares for deriving f'_i can never be activated when substituting f'_i for y_i in $R^{\langle i+1 \rangle}$.

Comparing $R^{(i)}$ of Section 3.2.1 and $R^{\langle i \rangle}$ of Section 3.2.2, one may notice that the AIG of $R^{(i)}$ is in general wider in width but shallower in depth, and, in contrast, that of $R^{\langle i \rangle}$ narrower but deeper.

As an implementation technicality, relations $R^{(i)}$ (similarly $R^{\langle i \rangle}$) can be stored in the same AIG manger. So structurally equivalent nodes are hashed together, and logic sharing is possible among relations $R^{(i)}$ (similarly $R^{\langle i \rangle}$).

3.3 Deterministic Relation

We consider the special case of extracting functions from a deterministic relation.

LEMMA 2. *Given a deterministic relation $D(\vec{x}, \vec{y})$ total in the input space X with*

$$D = \bigwedge_{i=1}^{m} (y_i \equiv f_i), \tag{7}$$

let

$$\phi_A = D(\vec{x}, y_1, \ldots, y_{i-1}, 1, y_{i+1}, \ldots, y_m) \text{ and } \tag{8}$$

$$\phi_B = D(\vec{x}, y'_1, \ldots, y'_{i-1}, 0, y'_{i+1}, \ldots, y'_m), \tag{9}$$

where y and y' are independent variables. Then the interpolant of ϕ_A with respect to ϕ_B is functionally equivalent to f_i.

PROOF. Since D is deterministic and total in X, for every $a \in X$ there exists a unique $b \in Y$ such that $D(a, b) = 1$. It follows that the formulas

$$\exists y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m . D[y_i/0] \tag{10}$$

and

$$\exists y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m . D[y_i/1] \tag{11}$$

must induce a partition on the input space X, and thus the interpolant of ϕ_A with respect to ϕ_B must logically equivalent to Formula (11), which is unique. ■

Back to the computation of Section 3.2.2, let

$$D = R \wedge \bigwedge_{i} (y_i \equiv f'_i). \tag{12}$$

Since the relation D is deterministic, the computation of Lemma 2 can be applied to compute f_i. The strengths of this new second-phase computation are twofold: First, no substitution is needed, in contrast to the second-phase computation of Section 3.2.2. Hence the formula sizes of ϕ_A and ϕ_B in interpolant computation do not grow, unlike the previous second-phase computation. As interpolant sizes are more or less proportional to the formula sizes of ϕ_A and ϕ_B, this approach is particularly desirable. Second, only functions f'_i, but not relations $R^{\langle i \rangle}$, are needed in the computation. Since the formula sizes of $R^{\langle i \rangle}$ are typically much larger than those of f'_i, this approach saves memory by discharging $R^{\langle i \rangle}$.

4. FUNCTION SIMPLIFICATION

The following lemma can be exploited in reducing the support variables of a functional implementation.

Table 1: Profile of original benchmark circuits.

circuit	(n, m)	orig		
		#n	#l	#v
s5378	(214, 179)	624	12	1570
s9234.1	(247, 211)	1337	25	3065
s13207	(700, 669)	1979	23	3836
s15850	(611, 597)	2648	36	15788
s35932	(1763, 1728)	8820	12	7099
s38584	(1464, 1452)	9664	26	19239
b10	(28, 17)	167	11	159
b11	(38, 31)	482	21	416
b12	(126, 121)	953	16	1639
b13	(63, 53)	231	10	383

LEMMA 3. *For two Boolean formulas ϕ_A and ϕ_B with an unsatisfiable conjunction, there exists an interpolant without referring to variable x_i if and only if the conjunction of $\exists x_i.\phi_A$ and ϕ_B (equivalently the conjunction of ϕ_A and $\exists x_i.\phi_B$) is unsatisfiable.*

PROOF. ($\Longleftarrow$) Assume the conjunction of $\exists x_i.\phi_A$ and ϕ_B (similarly ϕ_A and $\exists x_i.\phi_B$) is unsatisfiable. Since $\phi_A \Rightarrow \exists x_i.\phi_A$ (similarly $\phi_B \Rightarrow \exists x_i.\phi_B$), the conjunction of ϕ_A and ϕ_B is unsatisfiable as well. Also by the common-variable property of Theorem 1, the existence condition holds.

($\Longrightarrow$) Observe that $\exists x_i.\phi_A$ (respectively $\exists x_i.\phi_B$) is the tightest x_i-independent formula that is implied by ϕ_A (respectively ϕ_B). The existence of an interpolant of ϕ_A with respect to ϕ_B without referring to x_i infers the unsatisfiability of the conjunction of $\exists x_i.\phi_A$ and ϕ_B as well as that of ϕ_A and $\exists x_i.\phi_B$. ■

By the lemma, we can possibly knock out some variables from an interpolant.

Note that, in Lemma 3, it suffices to quantify x_i over ϕ_A or ϕ_B even though it is okay to quantify on both. In practice, quantification on just one formula results in smaller interpolants because the unsatisfiability is easier to be shown in SAT solving.

5. EXPERIMENTAL RESULTS

The proposed methods were implemented in the ABC package [3]; the experiments were conducted on a Linux machine with Xeon 3.4GHz CPU and 6Gb RAM.

To prepare Boolean relations, we constructed the transition relations of circuits taken from ISCAS and ITC benchmark suites. Different amounts of don't cares were inserted to the transition relations to introduce nondeterminism. We intended to retrieve a circuit's transition functions in the following experiments.

The original circuits[2] were minimized with the ABC command `dc2`, and so were the AIGs produced during determinization and function extraction. The profile of the original circuits (after the removal of primary-output functions and after `dc2` synthesis) is shown in Table 1, where "(n, m)" denotes the pair of input- and output-variable sizes of the transition relation, "#n" denotes the number of AIG nodes, "#l" AIG logic levels, and "#v" the summation of support variables of all transition functions.

We first study the usefulness of interpolation in contrast to cofactoring, which can be considered as a way of deriving special interpolants as mentioned in Section 3.1.1. In the experiment, a circuit was determinized via expansion reduction, where the functional implementations extracted in the second phase were derived differently using interpolation

[2]Since a circuit's primary-output functions are immaterial to our evaluation, we are concerned only about the sub-circuit responsible for transition functions.

Table 2: Function extraction from relations — without don't care insertion.

circuit	BDD				Xp				St				SD			
	#n	#l	#v	time	#n	#l	#v	time	#n	#l	#v	time	#n	#l	#v	time
s5378	783	10	1561	286.4	1412	25	1561	49.6	1328	23	1561	58.5	1625	34	1561	32.4
s9234.1	—	—	—	—	7837	59	2764	158.6	8015	45	2765	282.4	8637	45	2766	100.7
s13207	—	—	—	—	5772	140	3554	769.3	6625	223	3554	949.5	6642	109	3554	247.2
s15850	—	—	—	—	42622	188	13348	2700.0	42902	153	13318	3029.6	41014	357	13329	404.7
s35932	—	—	—	—	7280	10	6843	4178.5	7310	10	6843	3982.7	7293	12	6843	2039.2
s38584	—	—	—	—	22589	277	17678	5772.8	22691	387	17676	8481.0	17018	178	17676	2438.1
b10	200	10	152	0.1	197	8	152	0.9	231	14	152	1.7	234	14	152	1.0
b11	1301	18	394	0.9	1504	57	394	5.1	1759	55	394	14.9	1959	53	394	8.0
b12	1663	14	1574	56.7	2166	25	1574	24.0	2368	35	1575	78.8	2662	33	1576	38.6
b13	240	10	349	3.1	224	10	349	2.2	222	11	349	3.5	222	11	349	2.7
ratio 1					3.40	4.16	0.91		3.47	4.98	0.91		3.24	4.41	0.91	
ratio 2	1.70	0.89	0.97		2.24	1.79	0.97		2.40	1.97	0.97		2.53	1.90	0.97	
ratio 3	1.00	1.00	1.00		1.31	2.02	1.00		1.41	2.23	1.00		1.48	2.15	1.00	

Table 3: Function extraction from relations — with don't care insertion.

circuit	BDD				Xp				St				SD			
	#n	#l	#v	time	#n	#l	#v	time	#n	#l	#v	time	#n	#l	#v	time
s5378	769	11	1561	200.2	1332	25	1561	49.05	1196	27	1561	60.99	1919	42	1561	32.74
s9234.1	—	—	—	—	7696	55	2765	166.74	8739	64	2764	325.98	11613	99	2927	120.37
s13207	—	—	—	—	5818	202	3554	897.86	6882	228	3554	1062.15	6218	204	3554	287.47
s15850	—	—	—	—	40078	136	13309	2596.94	42097	164	13318	3012.36	41240	212	14276	467.95
s35932	—	—	—	—	7360	25	6843	4811.1	7300	10	6843	7168.53	8823	19	8756	2775.32
s38584	—	—	—	—	23726	331	17676	5476.67	21595	285	17676	8160.28	17708	281	18556	2591.71
b10	199	9	152	0.1	193	8	152	0.99	217	9	152	1.62	239	8	168	1.13
b11	1221	20	394	0.9	1562	52	394	5.5	1638	46	394	15.19	1896	54	394	8.54
b12	1619	15	1574	452.5	2261	23	1574	26.98	2081	24	1574	86.96	2458	25	1575	44.19
b13	243	11	349	1.6	229	12	349	2.45	236	10	349	4.2	232	10	349	2.9
ratio 1					3.35	4.53	0.91		3.42	4.52	0.91		3.43	4.97	0.98	
ratio 2	1.65	0.94	0.97		2.27	1.71	0.97		2.18	1.66	0.97		2.74	1.99	0.97	
ratio 3	1.00	1.00	1.00		1.38	1.82	1.00		1.33	1.76	1.00		1.66	2.11	1.00	

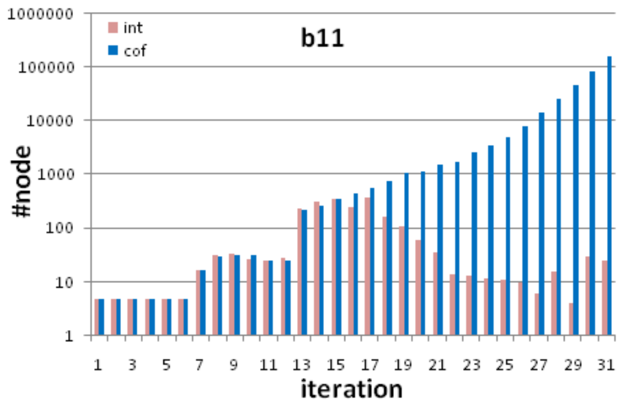

Figure 2: Circuit `b11` determinized by expansion reduction with function extraction by interpolation vs. cofactoring in the second phase.

and cofactoring to compare. Taking circuit `b11` as a typical example, Figure 2 contrasts the difference between the two techniques. As can be seen, by cofactoring, the function sizes grow almost exponentially during the iterative computation; by interpolation, the function sizes remain under control. In fact, derived by cofactoring, say, $R^{(i+1)}$ with $y_i = 1$, function f_i has almost the same size as $R^{(i+1)}$ unless command `dc2` can effectively minimize f_i. However, `dc2` is unlikely to be helpful for large f_i as justified by experiments.

Below we compare different determinization methods, including BDD-based computation, that via expansion reduction (Section 3.2.1), denoted Xp, that via substitution reduction (Section 3.2.2), denoted St, and that via constructing deterministic relation (Section 3.3), denoted SD. Dynamic variable reordering and BDD minimization are applied in BDD-based computation.

Table 2 shows the results of function extraction from relations without don't care insertion. BDD-based computation is not scalable as expected. There are five circuits whose transition relations cannot be built compactly using BDDs under 500K nodes and the computation cannot finish either within 30 hours CPU time or within the memory limitation. Ratio 1 and Ratio 2 are normalized with respect to the data of the original circuits of Table 1, whereas Ratio 3 is normalized with respect to the BDD-based derivation. Ratio 1 covers all the ten circuits, whereas Ratio 2 and Ratio 3 cover only the five circuits that BDD-based derivation can finish.

By Ratio 1, we observe that the derived functions (without further postprocessing to minimize) are about 3-times larger in nodes, 4-times larger in logic levels, and 9% smaller in support sizes. To be shown in Table 4, with postprocessing, the derived functions can be substantially simplified and are comparable to the original sizes. By Ratio 2, we see that even BDD-based derivation may increase circuit sizes by 70% while logic levels are reduced by 11%. By Ratio 3, we see that the results of the SAT-based methods are about 40% larger in nodes and 2-times larger in logic levels.

Table 3 shows the results of function extraction from relations with don't cares inserted. For a circuit with r registers, $\lceil r \cdot 10\% \rceil$ random cubes (conjunction of literals of input and state variables) are created. Each cube represents some don't cares for a randomly selected set of functions. Presumably the more the don't cares are inserted, the simpler the transition functions are extracted. In practice, however, such simplification[3] is not guaranteed (even in BDD-based computation). The reasons can be twofold: Firstly, the simplifi-

[3]Notice that, unlike BDD-based computation, our methods do not explicitly perform don't-care based minimization on the extracted transition functions. The don't-care choices are made implicitly by SAT solving for interpolation.

Table 4: Function extraction from relations — effect of collapse minimization.

circuit	orig						Xp						
	#n	#l	#v	#n_c	#l_c	#v_c	#n	#l	#v	#n_c	#l_c	#v_c	time_c
s5378	624	12	1570	772	11	1561	1412	25	1561	760	11	1561	0.4
s9234.1	1337	25	3065	2791	25	2764	7837	59	2764	2751	22	2764	6.4
s13207	1979	23	3836	2700	20	3554	5772	140	3554	2709	20	3554	5.2
s15850*	2648	36	15788	—	—	—	42622	188	13348	—	—	—	—
s35932	8820	12	7099	7825	9	6843	7280	10	6843	7857	9	6843	39.9
s38584	9664	26	19239	12071	23	17676	22589	277	17678	12132	21	17676	36.5
b10	167	11	159	195	10	152	197	8	152	195	10	152	0.0
b11	482	21	416	1187	21	394	1504	57	394	1226	21	394	0.2
b12	953	16	1639	1556	17	1574	2166	25	1574	1645	16	1574	0.4
b13	231	10	383	237	9	349	224	10	349	237	9	349	0.1
ratio 1	1.00	1.00	1.00	1.21	0.93	0.93	2.02	3.92	0.93	1.22	0.89	0.93	
ratio 2				1.00	1.00	1.00				1.01	0.96	1.00	

cation achieves only local optimums. Secondly, relations with don't cares inserted become more sophisticated and affect interpolant derivation. Nevertheless the results of Table 2 and Table 3 are comparable.

The above experiments of Xp, St, and SD used only light synthesis operations in minimizing the extracted functions. Nonetheless, it is possible to greatly simplify these functions with heavier synthesis operations. To justify such possibilities, we applied ABC command `collapse` once followed by `dc2` twice as postprocessing. Table 4 shows the statistics of extracted functions by Xp for relations without don't care insertion. (Similar results were observed for St and SD, and for cases with don't care insertion.) The postprocessing results of the original functions and extracted functions are shown. This postprocessing time is listed in the last column. Operation `collapse` failed on circuit `s15850`, and the two ratios shown excludes the data of `s15850`. As can be seen, the postprocessing makes the extracted functions comparable to the original ones. Since the postprocessing time is short, our method combined with some powerful synthesis operations can effectively extract simple functions from large relations, where pure BDD-based computation fails. Our method can be used as a way of bypassing the BDD memory explosion problem.

6. CONCLUSIONS AND FUTURE WORK

We have shown that Boolean relations with thousands of variables can be determinized inexpensively using interpolation. The extracted functions from a relation are of reasonable sizes. With such extended capacity, we would anticipate real-world applications, which might in turn enable constraint-based synthesis and verification, synthesis from specifications, and other areas that require solving large Boolean relations.

As we just presented a first step, there remain some obstacles to overcome. In particular, the unpredictability of interpolation prevents relation determinization from being robustly scalable. Moreover, we may need good determinization scheduling and powerful interpolant/AIG minimization techniques, especially under the presence of flexibility.

Acknowledgments

The authors are grateful to Robert Brayton and Alan Mishchenko for valuable discussions. This work was supported in part by NSC grants 95-2218-E-002-064-MY3 and 96-2221-E-002-278-MY3.

REFERENCES

[1] D. Baneres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve Boolean relations. In *Proc. Design Automation Conf.*, pages 416-421, 2004.

[2] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Proc. Design Automation and Test in Europe*, 2007.

[3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*, 2005. `http://www.eecs.berkeley.edu/~alanmi/abc/`

[4] R. Brayton and F. Somenzi. Boolean relations and the incomplete specification of logic networks. In *Proc. Int'l Conf. on Very Large Scale Integration*, 1989.

[5] R. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 316-319, 1989.

[6] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250-268, 1957.

[7] A. Ghosh, S. Devadas, and A. Newton. Heuristic minimization of Boolean relations using testing techniques. In *Proc. Int'l Conf. on Computer Design*, 1990.

[8] J.-H. R. Jiang. Quantifier elimination via functional composition. In *Proc. Int'l Conf. on Computer Aided Verification*, pages 383-397, 2009.

[9] S. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of Boolean relations. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 417-420, 1992.

[10] J. Kukula and T. Shiple. Building circuits from relations. In *Proc. Int'l Conf. on Computer Aided Verification*, pages 113-123, 2000.

[11] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *Proc. Int'l Conf. on Computer-Aided Design*, 2007.

[12] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung. Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In *Proc. Design Automation Conf.*, pages 636-641, 2008.

[13] H.-P. Lin, J.-H. R. Jiang, and R.-R. Lee. To SAT or not to SAT: Ashenhurst decomposition in a large scale. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 32-37, 2008.

[14] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 88-91, 1990.

[15] A. Mishchenko and R. K. Brayton. Simplification of non-deterministic multi-valued networks. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 557-562, 2002.

[16] A. Mishchenko, R. K. Brayton, J.-H. R. Jiang, and S. Jang. Scalable don't care based logic optimization and resynthesis. In *Proc. Int'l Symp. on Field-Programmable Gate Arrays*, pages 151-160, 2009.

[17] A. Mishchenko, S. Chatterjee, J.-H. R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. *ERL Technical Report*, UC Berkeley, March 2005.

[18] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. Int'l Conf. on Computer Aided Verification*, pages 1-13, 2003.

[19] Y.Watanabe and R. Brayton. Heuristic minimization of multi-valued relations. In *IEEE Trans. on Computer-Aided Design*, pages 1458-1472, Oct. 1993.

[20] B. Wurth and N. Wehn. Efficient calculation of Boolean relations for multi-level logic optimization. In *Proc. European Design and Test Conference*, pages 630-634, 1994.

[21] J. Yuan , K. Albin , A. Aziz , C. Pixley. Constraint synthesis for environment modeling in functional verification. In *Proc. Design Automation Conf.*, 2003.

[22] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using BDDs. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 584-589, 1999.