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 \( \{(x_1, x_2) \in \mathbb{B}^2 \} \rightarrow \{(y_1, y_2) \in \mathbb{B}^2 \} \) of Figure 1 as an example, we can express the (one-to-one) mapping of (a) using Boolean functions \( f_1 = x_1 x_2 \) 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) \( R = \neg x_1 \neg x_2 \neg y_1 y_2 \vee \neg x_1 x_2 \neg y_1 y_2 \vee \neg x_1 x_2 y_1 y_2 \vee x_1 \neg x_2 \neg y_1 y_2 \vee x_1 x_2 y_1 \neg y_2 \vee y_1 y_2 \).

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 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 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, determining a relation and extracting functional implementation in the Boolean domain.

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, 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], or in terms 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

\[ \begin{array}{cccccc}
00 & 00 & 00 & 00 & 00 & 00 \\
01 & 01 & 01 & 01 & 01 & 01 \\
10 & 10 & 10 & 10 & 10 & 10 \\
11 & 11 & 11 & 11 & 11 & 11 \\
\end{array} \]

Figure 1: Boolean mappings.

1In 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 determination 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 determination and function extraction. Boolean relations with thousands of variables can be determinized effectively and the extracted functions are typically of reasonable quality. Similar attempts were pursued recently in other efforts of scalable logic synthesis, e.g., [11, 12, 13, 16].

Therefore seeking a functional implementation of a total relation can be considered as determining 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 $\phi_A$ and $\phi_B$, with $\phi_A \land \phi_B$ unsatisfiable, then there exists a Boolean formula $\psi_A$ referring only to the common variables of $\phi_A$ and $\phi_B$ such that $\phi_A \Rightarrow \psi_A$ and $\psi_A \land \phi_B$ is unsatisfiable.

The Boolean formula $\psi_A$ is referred to as the interpolant of $\phi_A$ with respect to $\phi_B$. 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 $\psi_A$ of the refutation of

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

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 $\{a \in X \mid R(a, 0) = 0 \land R(a, 1) = 0\}$ is empty. Hence $\phi_A$ (respectively $\phi_B$) characterizes the set $S_A = \{a \in X \mid R(a, 0) = 1 \land R(a, 0) = 0\}$ (respectively $S_B = \{a \in X \mid R(a, 0) = 1 \land R(a, 0) = 0\}$). As $\phi_A \Rightarrow \psi_A$ and $\psi_A \Rightarrow \neg \phi_B$, the interpolant $\psi_A$ 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 implementations 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

\[
T(\vec{x}, y) = R(\vec{x}, y) \lor \forall \vec{y}. \neg R(\vec{x}, y).
\]

Note that \( T = R \) if and only if \( R \) is total.

Accordingly Theorem 2 is applicable to a totalized relation \( T \) with

\[
\phi_A = \neg T(\vec{x}, 0) \quad \text{and} \quad \phi_B = \neg T(\vec{x}, 1),
\]

which can be further simplified to

\[
\phi_A = \neg R(\vec{x}, 0) \land R(\vec{x}, 1) \quad \text{and} \quad \phi_B = \neg R(\vec{x}, 1) \land R(\vec{x}, 0).
\]

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 determination of single-output relations.

A determination procedure contains two phases: The first phase reduces the number of output variables; the second phase extracts functional implementation. We study two determination 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 \( \exists y_{i+1}, \ldots, y_m.R \) for \( 1 \leq i \leq m \). Through standard existential quantification by formula expansion, i.e., \( \exists x. \varphi = \varphi[x/0] \lor \varphi[x/1] \) for some formula \( \varphi \) 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.

\[
R^{(1)} = \exists y_1, R
\]

\[
R^{(2)} = \exists y_2, R^{(1+1)}
\]

for \( i = m - 1, \ldots, 2 \).

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

\[
f_1 = FI(y_1, R^{(2)})
\]

\[
f_i = FI(y_i, R^{(i+1)}[y_i/f_{i-1}, \ldots, y_{i-1}/f_{i-1}])
\]

\[
f_m = FI(y_m, R[y_1/f_1, \ldots, y_{m-1}/f_{m-1}])
\]

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 determination 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 \( y_1, \ldots, y_{m-1} \) as 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 \( \hat{f}_m = FI(y_m, R) \), obtained through interpolation mentioned in Section 3.2.1. Note that since \( \hat{f}_m \) 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.

\[
f_m = FI(y_m, R)
\]

\[
R^{(m)} = R[y_m/f_m]
\]

\[
f_i = FI(y_i, R^{(i+1)}[y_i/f_i', \ldots, y_{i-1}/f_{i-1}'])
\]

\[
R^{(i)} = R^{(i+1)}[y_i/f_i']
\]

\[
f_2 = FI(y_2, R^{(2)})
\]

\[
R^{(2)} = R^{(3)}[y_2/f_2']
\]

for \( i = m - 1, \ldots, 2 \).

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

\[
f_1 = FI(y_1, R^{(2)})
\]

\[
f_i = FI(y_i, R^{(i+1)}[y_i/f_i', \ldots, y_{i-1}/f_{i-1}'])
\]

\[
f_m = FI(y_m, R[y_1/f_1', \ldots, y_{m-1}/f_{m-1}'])
\]

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 \( R^{i} = R^{2} \).

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 \( R^{i+1} \).

Comparing \( R^{i} \) of Section 3.2.1 and \( R^{i} \) 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 \( R^{i} \) narrower but deeper.

As an implementation technicality, relations \( R^{i} \) (similarly \( R^{1} \)) can be stored in the same AIG manager. So structurally equivalent nodes are hashed together, and logic sharing is possible among relations \( R^{i} \) (similarly \( R^{1} \)).

3.3 Deterministic Relation

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

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

\[
D = \bigwedge_{i=1}^{m} (y_{i} \equiv f_{i}),
\]

let

\[
\phi_{A} = D(\vec{x}, y_{1}, \ldots, y_{i-1}, 1, y_{i+1}, \ldots, y_{m}) \quad \text{and} \quad \phi_{B} = D(\vec{x}, y_{1}, \ldots, y_{i-1}, 0, y_{i+1}, \ldots, y_{m}).
\]

where \( y \) and \( y' \) are independent variables. Then the interpolant of \( \phi_{A} \) with respect to \( \phi_{B} \) 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

\[
\exists y_{1}, \ldots, y_{i-1}, 1, y_{i+1}, \ldots, y_{m}, D[y_{i}/0]
\]

and

\[
\exists y_{1}, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{m}, D[y_{i}/1]
\]

must induce a partition on the input space \( X \), and thus the interpolant of \( \phi_{A} \) with respect to \( \phi_{B} \) must logically equivalent to Formula (11), which is unique.

Back to the computation of Section 3.2.2, let

\[
D = R \land \bigwedge_{i}(y_{i} \equiv f'_{i}).
\]

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 \( \phi_{A} \) and \( \phi_{B} \) 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 \( \phi_{A} \) and \( \phi_{B} \), this approach is particularly desirable. Second, only functions \( f'_{i} \), but not relations \( R^{i} \), are needed in the computation. Since the formula sizes of \( R^{i} \) are typically much larger than those of \( f'_{i} \), this approach saves memory by discharging \( R^{i} \).

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.**

<table>
<thead>
<tr>
<th>circuit</th>
<th>(( n, m ))</th>
<th>#n</th>
<th>#p</th>
<th>#v</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3384</td>
<td>(193, 102)</td>
<td>271</td>
<td>47</td>
<td>70</td>
</tr>
<tr>
<td>s3640</td>
<td>(132, 101)</td>
<td>168</td>
<td>36</td>
<td>64</td>
</tr>
<tr>
<td>s3935</td>
<td>(113, 102)</td>
<td>96</td>
<td>18</td>
<td>27</td>
</tr>
<tr>
<td>s4065</td>
<td>(105, 102)</td>
<td>84</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>s4128</td>
<td>(102, 102)</td>
<td>72</td>
<td>12</td>
<td>18</td>
</tr>
</tbody>
</table>

**Lemma 3.** For two Boolean formulas \( \phi_{A} \) and \( \phi_{B} \) 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 \( \phi_{B} \) (equivalently the conjunction of \( \phi_{A} \) and \( \exists x_{i}, \phi_{B} \)) is unsatisfiable.

**Proof.** Assume the conjunction of \( \exists x_{i}, \phi_{A} \) and \( \phi_{B} \) (similarly \( \phi_{A} \) and \( \exists x_{i}, \phi_{B} \)) is unsatisfiable. Since \( \phi_{A} \Rightarrow \exists x_{i}, \phi_{B} \) (similarly \( \exists x_{i}, \phi_{B} \Rightarrow \phi_{B} \)), the conjunction of \( \phi_{A} \) and \( \phi_{B} \) is as well. Also by the common-variable property of Theorem 1, the existence condition holds. 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 \( \phi_{A} \) or \( \phi_{B} \) 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 benchmarks. 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 were minimized with the ABC command dc2, and so were the AIGs produced during deterministic 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, \( \#p \) \#1 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 determined 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.

<table>
<thead>
<tr>
<th>circuit</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0318</td>
<td>10</td>
<td>156</td>
<td>286.4</td>
<td>112</td>
<td>29</td>
<td>156</td>
<td>49.8</td>
<td>132</td>
<td>24</td>
<td>156</td>
</tr>
<tr>
<td>s0241</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7837</td>
<td>59</td>
<td>2764</td>
<td>158.6</td>
<td>8015</td>
<td>45</td>
<td>2765</td>
</tr>
<tr>
<td>s13207</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>5772</td>
<td>140</td>
<td>3554</td>
<td>769.3</td>
<td>6625</td>
<td>223</td>
<td>3554</td>
</tr>
<tr>
<td>s15850</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>42622</td>
<td>188</td>
<td>13348</td>
<td>2700.0</td>
<td>42902</td>
<td>153</td>
<td>13318</td>
</tr>
<tr>
<td>s35302</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7280</td>
<td>10</td>
<td>6843</td>
<td>4178.5</td>
<td>7310</td>
<td>10</td>
<td>6843</td>
</tr>
<tr>
<td>s38584</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>22589</td>
<td>277</td>
<td>17768</td>
<td>5772.8</td>
<td>2269</td>
<td>387</td>
<td>17768</td>
</tr>
<tr>
<td>b10</td>
<td>200</td>
<td>10</td>
<td>152</td>
<td>0.9</td>
<td>197</td>
<td>8</td>
<td>152</td>
<td>0.9</td>
<td>231</td>
<td>14</td>
</tr>
<tr>
<td>b11</td>
<td>1301</td>
<td>18</td>
<td>394</td>
<td>0.9</td>
<td>1504</td>
<td>57</td>
<td>394</td>
<td>5.1</td>
<td>1759</td>
<td>55</td>
</tr>
<tr>
<td>b12</td>
<td>1663</td>
<td>14</td>
<td>1574</td>
<td>56.7</td>
<td>2166</td>
<td>25</td>
<td>1574</td>
<td>24.0</td>
<td>2368</td>
<td>35</td>
</tr>
<tr>
<td>b13</td>
<td>240</td>
<td>10</td>
<td>349</td>
<td>3.1</td>
<td>224</td>
<td>10</td>
<td>349</td>
<td>2.2</td>
<td>222</td>
<td>11</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>circuit</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0378</td>
<td>769</td>
<td>11</td>
<td>1561</td>
<td>200.2</td>
<td>1332</td>
<td>25</td>
<td>1561</td>
<td>49.05</td>
<td>1196</td>
<td>27</td>
</tr>
<tr>
<td>s0231</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7696</td>
<td>55</td>
<td>2765</td>
<td>166.74</td>
<td>878</td>
<td>65</td>
<td>2764</td>
</tr>
<tr>
<td>s13207</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>5818</td>
<td>202</td>
<td>3554</td>
<td>897.86</td>
<td>6828</td>
<td>228</td>
<td>3554</td>
</tr>
<tr>
<td>s15850</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>4078</td>
<td>136</td>
<td>13309</td>
<td>2596.94</td>
<td>42097</td>
<td>164</td>
<td>13318</td>
</tr>
<tr>
<td>s35302</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7360</td>
<td>25</td>
<td>6843</td>
<td>4811.1</td>
<td>7300</td>
<td>10</td>
<td>6843</td>
</tr>
<tr>
<td>s38584</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>23729</td>
<td>331</td>
<td>17676</td>
<td>5476.07</td>
<td>21595</td>
<td>285</td>
<td>17676</td>
</tr>
<tr>
<td>b10</td>
<td>199</td>
<td>9</td>
<td>152</td>
<td>0.1</td>
<td>193</td>
<td>8</td>
<td>152</td>
<td>0.99</td>
<td>217</td>
<td>9</td>
</tr>
<tr>
<td>b11</td>
<td>1221</td>
<td>20</td>
<td>394</td>
<td>0.9</td>
<td>1562</td>
<td>52</td>
<td>394</td>
<td>5.5</td>
<td>1638</td>
<td>46</td>
</tr>
<tr>
<td>b12</td>
<td>1619</td>
<td>15</td>
<td>1574</td>
<td>542.5</td>
<td>2261</td>
<td>23</td>
<td>1574</td>
<td>26.88</td>
<td>2081</td>
<td>24</td>
</tr>
<tr>
<td>b13</td>
<td>243</td>
<td>11</td>
<td>349</td>
<td>1.6</td>
<td>229</td>
<td>12</td>
<td>349</td>
<td>2.45</td>
<td>236</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4: Function extraction from relations — with don't care insertion.

<table>
<thead>
<tr>
<th>circuit</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>#n</th>
<th>#v</th>
<th>time</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0378</td>
<td>769</td>
<td>11</td>
<td>1561</td>
<td>200.2</td>
<td>1332</td>
<td>25</td>
<td>1561</td>
<td>49.05</td>
<td>1196</td>
<td>27</td>
</tr>
<tr>
<td>s0231</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7696</td>
<td>55</td>
<td>2765</td>
<td>166.74</td>
<td>878</td>
<td>65</td>
<td>2764</td>
</tr>
<tr>
<td>s13207</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>5818</td>
<td>202</td>
<td>3554</td>
<td>897.86</td>
<td>6828</td>
<td>228</td>
<td>3554</td>
</tr>
<tr>
<td>s15850</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>4078</td>
<td>136</td>
<td>13309</td>
<td>2596.94</td>
<td>42097</td>
<td>164</td>
<td>13318</td>
</tr>
<tr>
<td>s35302</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7360</td>
<td>25</td>
<td>6843</td>
<td>4811.1</td>
<td>7300</td>
<td>10</td>
<td>6843</td>
</tr>
<tr>
<td>s38584</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>23729</td>
<td>331</td>
<td>17676</td>
<td>5476.07</td>
<td>21595</td>
<td>285</td>
<td>17676</td>
</tr>
<tr>
<td>b10</td>
<td>199</td>
<td>9</td>
<td>152</td>
<td>0.1</td>
<td>193</td>
<td>8</td>
<td>152</td>
<td>0.99</td>
<td>217</td>
<td>9</td>
</tr>
<tr>
<td>b11</td>
<td>1221</td>
<td>20</td>
<td>394</td>
<td>0.9</td>
<td>1562</td>
<td>52</td>
<td>394</td>
<td>5.5</td>
<td>1638</td>
<td>46</td>
</tr>
<tr>
<td>b12</td>
<td>1619</td>
<td>15</td>
<td>1574</td>
<td>542.5</td>
<td>2261</td>
<td>23</td>
<td>1574</td>
<td>26.88</td>
<td>2081</td>
<td>24</td>
</tr>
<tr>
<td>b13</td>
<td>243</td>
<td>11</td>
<td>349</td>
<td>1.6</td>
<td>229</td>
<td>12</td>
<td>349</td>
<td>2.45</td>
<td>236</td>
<td>10</td>
</tr>
</tbody>
</table>

**Figure 2:** Circuit b11 determined 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, $R_i^{(4)}$ with $y_i = 1$, function $f_i$ has almost the same size as $R_i^{(4)}$ unless command dof can effectively minimize $f_i$. However, dof is unlikely to be helpful for large $f_i$ as justified by experiments.

Below we compare different determination methods, including BDD-based computation, that via expansion reduction (Section 3.2.1), denoted Xr, that via substitution reduction (Section 3.2.2), denoted St, and that via constructing deterministic relation (Section 3.3.1), 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, $\lfloor r \cdot 10\% \rfloor$ 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 is not guaranteed (even in BDD-based computation). The reasons can be twofold: Firstly, the simplification3 is not guaranteed (even in BDD-based computation).
cation achieves only local optiumums. Secondly, relations with don’t cares inserted become more sophisticated and affect interolation problem. 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 a15850, and the two ratios shown excludes the data of a15850. 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 determined inexpensively using interolation. 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, 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 interolation prevents relation determination from being robustly scalable. Moreover, we may need good determination scheduling and powerful interoant/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 NSF grants 95-2218-E-002-064-MY3 and 96-2221-E-002-278-MY3.

REFERENCES


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

| circuit | orig | X | Y | #n | #t | #v | #n, | #t, | #v, | #n,<sup>1</sup> | #t,<sup>1</sup> | #v,<sup>1</sup> | #n,<sup>2</sup> | #t,<sup>2</sup> | #v,<sup>2</sup> | time,<sup>2</sup> |
|---------|------|---|---|-----|----|----|-----|-----|----|-----|----|-----|-----|----|-----|-----|-----|
| s5378   | 622  | 12 | 1570 | 172 | 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 |
| s1207   | 1979 | 23 | 3836 | 2700 | 20  | 3554 | 5772 | 140 | 3554 | 2099 | 20  | 3554 | 5.2 |
| s15850* | 2648 | 16 | 35788 | 77857 | 152 | 7857 | 42622 | 188 | 7857 | 13348 | 1574 | 17676 | 0.2 |
| s35932  | 8820 | 12 | 7099 | 7825 | 9   | 6843 | 7280 | 10  | 6843 | 7857 | 9   | 6843 | 39.9 |
| s3854   | 9664 | 20 | 19259 | 12080 | 21  | 17676 | 22549 | 277 | 12080 | 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 | 1.00 | 0.93 | 0.93 | 2.02 | 1.00 | 1.00 | 0.93 | 1.00 | 0.93 |
| ratio 2  | 1.00 | 1.00 | 1.00 | 1.00 | 0.93 | 1.00 | 2.02 | 1.00 | 1.00 | 0.93 | 1.00 | 1.00 |