Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking

Poul F. Williams
Technical University of Denmark
pfw@it.dtu.dk

Armin Biere
ETH Zentrum, Switzerland
biere@inf.ethz.ch

Edmund M. Clarke
Carnegie Mellon University
edmund.clarke@cs.cmu.edu

Anubhav Gupta
Carnegie Mellon University
anubhav@cs.cmu.edu

Abstract

In this paper we show how to do symbolic model checking using Boolean Expression Diagrams (BEDs), a non-canonical representation for Boolean formulas, instead of Binary Decision Diagrams (BDDs), the traditionally used canonical representation. The method is based on standard fixed point algorithms, combined with BDDs and SAT-solvers to perform satisfiability checking. As a result we are able to model check systems for which standard BDD-based methods fail. For example, we model check a liveness property of a 256 bit shift-and-add multiplier. As opposed to Bounded Model Checking (BMC) our method is complete in practice.

Our technique is based on a quantification procedure that allows us to eliminate quantifiers in Quantified Boolean Formulas (QBF). The basic step of this procedure is the up-one operation for BEDs. In addition we list a number of important optimizations to reduce the number of basic steps. In particular the optimization rule of quantification-by-substitution turned out to be very useful: $\exists x : g(x \leftrightarrow f) \Rightarrow g[f/x]$. The rule is used (1) during fixed point iterations, (2) for deciding whether an initial set of states is a subset of another set of states, and finally (3) for iterative squaring.

1 Introduction

Symbolic model checking has been performed using fixed point iterations for quite some time [8]. The key to the success is the canonical Binary Decision Diagram (BDD) [6] data structure for representing Boolean functions. However, such a representation explodes in size for certain functions. In this paper we show how to do symbolic model checking using Boolean Expression Diagrams (BEDs) [2, 3], a non-canonical representation of Boolean functions. The method is theoretically complete as we only change the representation and not the algorithms. Dropping the canonicity requirement has both advantages and disadvantages: Non-canonical data structures are more succinct than canonical ones – sometimes exponentially more. Determining satisfiability of Boolean functions is easy with canonical data structures, but with non-canonical data structures it is hard. We show how to overcome the disadvantages and exploit some of the advantages in symbolic model checking.

As a non-canonical representation, BEDs do not allow for constant time satisfiability checking. Instead we use two different methods for satisfiability checking: (1) SAT-solvers like GRASP and SATO, and (2) conversion of BEDs to BDDs. BDDs are canonical and thus satisfiability checking is a constant time operation. We perform symbolic model checking the classical way with fixed point iterations. One of the key elements of our method is the quantification-by-substitution rule: $\exists x : g(x \leftrightarrow f) \Rightarrow g[f/x]$. The rule is used (1) during fixed point iterations, (2) while deciding whether an initial set of states is a subset of another set of states, and finally (3) while doing iterative squaring.

While complete in the sense that it handles full CTL model checking, our method performs best if the system has few inputs and the transition relation can be written as a conjunction of next-state functions. The reason is that this allows us to fully exploit the quantification-by-substitution rule.

Using our method, we can model check a liveness property of a 256 bit shift-and-add multiplier, which requires 256 iterations to reach the fixed point. This should be compared with the 23 bit multipliers that standard BDD methods can handle. It was generally thought that iterative squaring was of no use in model checking. However, we show that iterative squaring enables us to calculate the reachable set of states for all 32 outputs of a 16 bit multiplier faster than without iterative squaring.

*This research is sponsored in part by the Semiconductor Research Corporation (SRC) under agreement through Contract No. 99-TJ-684 and the National Science Foundation (NSF) under Grant Nos. CCR-9505472 and CCR-9803774. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.
2 Boolean Expression Diagrams

A Boolean Expression Diagram [2, 3] is a data structure for representing and manipulating Boolean formulas. A Boolean Expression Diagram (BED) is a directed acyclic graph \( G = (V, E) \) with vertex set \( V \) and edge set \( E \). The vertex set \( V \) contains four types of vertices: terminal, variable, operator, and quantifier vertices. The relation between a BED and the Boolean function it represents is straightforward. Terminal vertices correspond to the constant functions 0 and 1. Variable vertices have the same semantics as vertices of BDDs and correspond to the \( \text{if-then-else} \) operator \( x \rightarrow f_1, f_0 \) defined as \( (x \land f_1) \lor \neg x \land f_0 \). Operator vertices correspond to their respective Boolean connectives. Quantifier vertices correspond to the quantification of their associated variable.

There exist algorithms for transforming a BED into a BDD. One such algorithm is \textit{up-one}. It sifts variables one at a time to the root of the BED. Using \textit{up-one} repeatedly to sift all the variables transforms the BED to a BDD. We refer the reader to [2, 3] for a more detailed description of \textit{up-one} and its applications.

3 Quantification

Image computation is the key step in model checking. The basic step in our quantification algorithm is to eliminate one quantified variable by the following rules. Note that this basic step can easily be computed by performing a \textit{up-one}(\( f, x \)) BED-operation and then replacing the top level variable vertex by an appropriate operator vertex.

\[
\exists x : f \equiv f[0/x] \lor f[1/x] \quad \forall x : f \equiv f[0/x] \land f[1/x]
\]

In the worst case, while removing a quantifier from a formula, we double the formula size. The most important transformation is the \textit{quantification-by-substitution} rule. It allows us to replace an existential quantification by a substitution:

\[
\exists x : g \land (x \leftrightarrow f) \equiv g[f/x]
\]

where \( x \) does not occur as a free variable in \( f \).

Our verification method performs best when we can exploit the \textit{quantification-by-substitution} rule. Such cases include systems with few inputs and systems with a transition relation that is mainly in functional form. After performing \textit{quantification-by-substitution}, we quantify the remaining state variables (including inputs) using the rules below.

By applying scope reduction rules to a formula, we can push quantifiers down and thus reduce the potential blowup. The scope reduction rules are the following (shown for negation, conjunction and disjunction):

\[
\exists x : \neg f \equiv \neg \forall x : f \quad \forall x : \neg f \equiv \neg \exists x : f
\]

\[
\exists x : f \land g \equiv (\exists x : f) \land (\exists x : g) \quad \forall x : f \land g \equiv (\forall x : f) \land (\forall x : g)
\]

\[
\exists x : f(y) \land g(x) \equiv (\exists x : g(x)) \quad (\forall x : f(y) \land g(x)) \equiv f(y) \land (\forall x : g(x))\]

Because BEDs are always reduced, for details see [2, 3], the quantifiers disappear if they are pushed all the way to the terminals.

4 Satisfiability Checking

There are two places where we need to determine whether a Boolean formula represented by a BED is satisfiable. First we need to detect that a fixed point has been reached in the computation of the set of states satisfying a CTL formula. Let \( R_i \) be the \( i \)th approximation to the fixed point. The fixed point has been reached if \( R_{i+1} = R_i \). Using characteristic functions, this translates to \( R_{i+1} \equiv R_i \). Until we reach the fixed point, these formulas will not be tautologies. In other words, the negation of the formulas will be satisfiable. SAT-solvers are good at finding a satisfying variable assignment so we use a SAT-solver here.

Second we need to determine whether the initial set of states \( I \) is a subset of the set of states \( R \) represented by the CTL specification. In particular we have to check \( I \Rightarrow R \) for tautology. If the specification holds, \( I \Rightarrow R \) is a tautology. It is our experience that most SAT-solvers are not very good at proving non-satisfiability. So, by using the \textit{up-one} algorithm, we convert the BED for \( I \Rightarrow R \) to a BDD. If the specification does not hold, a proof will be a variable assignment falsifying \( I \Rightarrow R \). SAT-solvers are good at finding such variable assignments. Of course, we do not know before hand whether the specification holds. A possibility is to run a SAT-solver and a BED to BDD conversion in parallel.

5 Applications of Quantification-by-Substitution

If the transition relation \( T \) is written in functional form,

\[
T(s, s') = \bigwedge s_i' \leftrightarrow f_i(s)
\]
then we can apply rule (1) directly for the functional part. This can be done in one traversal of the BED. The algorithm works in a bottom-up way replacing all variables from the functional part of \( T \) with their next-state function. The initial set of states \( I \) often has the form:

\[
I = \bigwedge_i s_i \iff init_i(s)
\]

Thus, we can apply this optimization to simplify the computation of \( I \Rightarrow R \), i.e., whether the initial set of states is a subset of the states characterized by the specification.

Iterative squaring is a technique for reducing the number of iterations needed to reach the fixed point [7]. During reachability analysis we repeatedly square the transition relation:

\[
T^2(s, s') = \exists s'' : T(s, s'') \land T(s'', s')
\]

If we restrict ourselves to transition relations purely in functional form, squaring can be done easily:

\[
T^2(s, s') = \exists s'' : T(s, s'') \land T(s'', s') = \exists s'' : \left( \bigwedge_i s''_i \iff f_i(s) \right) \land \left( \bigwedge_i s''_i \iff f_i(s'') \right) = \bigwedge_i s'_i \iff (f_i(s'')[f(s)/s''])
\]

6 Experimental Results

We have constructed a prototype implementation of our proposed model checking method. It performs CTL model checking on SMV programs. For the experiments presented here we use Sato as our SAT-solver. We compare our method with the NuSMV model checker and with Bwolen Yang’s modified version of SMV\(^1\), both of which are state-of-the-art in BDD-based model checking. Finally we compare reachability results with F i x I T from Abdulla, Bjesse, and Eén [1].

The FixIT results are taken directly from the paper by Abdulla and his group\(^2\). All other experiments are run on a Linux computer with a Pentium Pro 200 MHz processor and 1 gigabyte of main memory.

<table>
<thead>
<tr>
<th>Bit</th>
<th>BED</th>
<th>NuSMV</th>
<th>Bwolen</th>
<th>FixIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.2</td>
<td>11</td>
<td>9.4</td>
<td>2.9</td>
</tr>
<tr>
<td>1</td>
<td>2.3</td>
<td>23</td>
<td>17</td>
<td>3.1</td>
</tr>
<tr>
<td>2</td>
<td>2.9</td>
<td>50</td>
<td>33</td>
<td>3.7</td>
</tr>
<tr>
<td>3</td>
<td>3.8</td>
<td>130</td>
<td>71</td>
<td>4.8</td>
</tr>
<tr>
<td>4</td>
<td>5.2</td>
<td>290</td>
<td>159</td>
<td>6.6</td>
</tr>
<tr>
<td>5</td>
<td>7.0</td>
<td>702</td>
<td>383</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>9.2</td>
<td>-</td>
<td>1031</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>544</td>
</tr>
<tr>
<td>10</td>
<td>68</td>
<td>-</td>
<td>-</td>
<td>2078</td>
</tr>
<tr>
<td>11</td>
<td>352</td>
<td>-</td>
<td>-</td>
<td>8134</td>
</tr>
<tr>
<td>12</td>
<td>2201</td>
<td>-</td>
<td>-</td>
<td>30330</td>
</tr>
</tbody>
</table>

Table 1: Runtimes in seconds for verifying the correctness of a 16 bit multiplier. A dash “-” indicates that the verification could not be completed with 800 MB of memory.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Without I.S.</th>
<th>With I.S.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.1</td>
<td>0.9</td>
</tr>
<tr>
<td>5</td>
<td>6.8</td>
<td>1.6</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>3.7</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>8.3</td>
</tr>
<tr>
<td>20</td>
<td>37</td>
<td>12</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>8.8</td>
</tr>
<tr>
<td>30</td>
<td>&gt; 12 hours</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Table 2: Runtimes in seconds for the fixed point calculation in verifying the correctness of the 16 bit shift-and-add multiplier. Results are shown for computations with and without iterative squaring (I.S.). The space requirements are small, i.e., less than 16 MB.

7 Conclusion

We have presented a BED-based CTL model checking method based on the classical fixed point iterations. Quantification is often the Achilles heel in CTL fixed point iterations but by using quantification-by-substitution we are in some cases

\(^1\)http://www.cs.cmu.edu/~bwolen

\(^2\)From personal correspondence with the authors we have learned that they used a 296 MHz Sun UltraSPARC-II for the barrel shifter experiments and a 333 MHz Sun UltraSPARC-III for the multiplier experiments.
Table 3: Runtimes in seconds for verifying that shift-and-add multipliers of different sizes always terminate, i.e., we check “AF done”. The number of iterations to reach the fixed point is equal to the size of the multiplier.

<table>
<thead>
<tr>
<th>Size</th>
<th>BED</th>
<th>NoSMV</th>
<th>Bwolen</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.6</td>
<td>2.2</td>
<td>5.2</td>
</tr>
<tr>
<td>18</td>
<td>1.8</td>
<td>1.8</td>
<td>9.1</td>
</tr>
<tr>
<td>20</td>
<td>2.0</td>
<td>90</td>
<td>24</td>
</tr>
<tr>
<td>22</td>
<td>2.3</td>
<td>472</td>
<td>104</td>
</tr>
<tr>
<td>23</td>
<td>2.7</td>
<td>-253</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2.8</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>5.7</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>17</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>119</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1185</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Runtimes in seconds for invariant (left) and liveness (right) checking of the barrel shifter example. A question mark indicates that the runtime for FixIt was not reported in [1]. For the BED method we use SATO for checking satisifiability of $I \Rightarrow R$.

able to deal effectively with it. While our method is complete, it performs best on examples with a low number of inputs and where the transition relation is mainly in functional form. In these situations we can fully exploit the quantification-by-substitution rule.

We have shown how the quantification-by-substitution rule can also help simplify the final set inclusion problem of model checking and help perform efficient iterative squaring. Our proposed method combines SAT-solvers and BED to BDD conversions to perform satisfiability checking. We use a set of local rewriting rules which helps to keep the size of the BEDs down.

We have demonstrated our method by model checking large shift-and-add multipliers and barrel shifters, and we obtain results superior to standard BDD-based model checking methods. Furthermore, we were able to find a previously undetected bug in the specification of a 16 bit multiplier.

Future work includes investigating two variable ordering problems. One is the variable ordering when converting the BED for $I \Rightarrow R$ to a BDD. The variable ordering is known to be very important in BDD construction, and since we, in some cases, spend much time on converting $I \Rightarrow R$ to a BDD, our method will benefit from a good variable ordering heuristic. The other problem is the order in which we quantify the variables in the PreImage computation. This will be interesting especially in cases where we cannot use the quantification-by-substitution rule. Finally we are currently investigating how to extend our method to work well for systems with many inputs.

References


