Logic Synthesis for Disjunctions of Boolean Functions

Extended Abstract

Baruch Sterin Alan Mishchenko Niklas Een Robert Brayton

Department of EECS, University of California, Berkeley

sterin@berkeley.edu alanmi@eecs.berkeley.edu niklas@een.se brayton@eecs.berkeley.edu

Abstract

This paper develops theoretical foundations and presents a practical algorithm for optimizing multi-output Boolean function M(x) whose outputs are combined using Boolean OR operator into a single-output Boolean function S(x). The proposed algorithm simplifies the logic structure of function M(x) and may change or remove some of its outputs, while preserving the functionality of function S(x).

Applications of the proposed algorithm are (1) optimization of Boolean networks containing multi-input AND/OR gates, (2) minimization of sets of interpolants in interpolation-based model checking, (3) minimization of the structural representations of sets of states in circuit-based reachability analysis, to mention just a few.

Experiments will be conducted to show the practicality of the proposed approach.

1. Introduction

Many practical applications deal with multi-output Boolean functions whose outputs are combined using the disjunction operator (Boolean OR). The representation of such functions can be optimized because the disjunction produces a don't-care set for individual functions. Indeed, if one output has value 1 for some input combinations, other outputs can produce any value for these combinations and the output value of the disjunction will remain the same.

A well-known and well-studied example of this is a Sum of Products (SOP) expression. An SOP is a disjunction (sum) of Boolean functions, each of which is a conjunction (product) of literals of its input variables. Efficient methods for minimizing SOPs have been developed [7][14][8] and found numerous practical applications.

In a more general sense, disjunctions of Boolean functions, represented as multi-level circuits, often arise in the domain of logic synthesis and formal verification. For example, in interpolation-based model checking [9], the over-approximation of reachable states is computed as a disjunction of interpolants. Each interpolant is a highly redundant logic function in itself, but the disjunction of them brings even more redundancy due to the don't-cares projected by individual interpolants onto each other.

A similar situation arises in the exact reachability analysis when multi-level circuits rather than BDDs are used to represent sets (or subsets) of states reachable in a given number of transitions from the initial state [6][13]. In this case, the union of all states reached so far is a disjunction of multi-level circuits. Optimization of this representation is key for the scalability of these methods.

These practical applications motivate logic synthesis methods which can efficiently represent and optimize disjunctions of multi-output Boolean functions.

Boolean functions can be represented as truth tables, BDDs, SOPs, etc. Of practical interest is the handling of Boolean functions represented as multi-level circuits, in particular, And-Inverter-Graphs (AIGs). Many applications relying on AIGs to represent Boolean functions suffer from the lack of efficient methods to reduce intermediate AIGs. This is true about the two practical applications mentioned above (interpolation-based model checking and circuit-based reachability) as well as several others.

The contribution of this paper is a methodology for optimizing disjunctions of multi-level AIGs and a SAT-based algorithm for performing the optimization.

We note that the same methodology can be applied to conjunctions of Boolean functions, which can be, for example, sets of constraints imposed on a SAT instance in some applications. Minimization of a conjunction can be reduced to that of a disjunction using DeMorgan's rule.

The paper is organized as follows. Section 2 describes the background. Section 3 outlines the algorithms. Section 4 reports experimental results. Section 5 concludes and outlines future work.

2. Background

2.1 Boolean network

A *Boolean network* (or *netlist*, or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the gates. In this paper, we consider only combinational Boolean networks. Sequential networks are handled as combinational networks by cutting at the register boundary.

A combinational *And-Inverter Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. The *size* (*area*) of an AIG is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the primary inputs (PIs) to the primary outputs (POs).

2.2 Interpolation

Given Boolean functions, $(A(x,y),B(y,z))$, such that $A(x, y) \wedge B(y, z) = 0$, and (x,y,z) is a partition of the variables, an *interpolant* is a Boolean function, $I(y)$, such

that $A(x,y) \subseteq I(y) \subseteq \overline{B}(y,z)$. If $A(x,y)$ and $B(y,z)$ are sets of CNF clauses, then their conjunction is an unsatisfiable SAT instance, and an interpolant can be computed using the algorithm presented in [9] (Definition 2).

Function $A(x,y)$ can be interpreted as the onset of an incompletely-specified Boolean function, $B(y,z)$ as the offset, and $\overline{A(x,y)} \wedge \overline{B(y,z)}$ as the don't-care set. Thus, $I(y)$ can be seen as an optimized version of $A(x,y)$ under the don't-cares. The result of this optimization is a completely-specified function I, which depends on variables y.

2.3 Resubstitution

Resubstitution ([2], Section IVB) is a circuit restructuring technique, which expresses one logic function in terms of others. When the functions are represented as multi-level circuits (in particular, combinational AIGs), the goal is to reduce the circuit size by removing some circuit nodes and instead reusing some other nodes. Resubstitution for large circuits can be efficiently implemented using a SAT solver and an interpolation package.

Additional information can be found in the following publications: AIGs [3], AIG-based synthesis [10][11], SAT solving [4], SAT-based resubstitution [12].

3. Algorithm

A *Sum of AIGs* (SOAIG) is a disjunction of the outputs of a circuit represented as a multi-output combinational AIG.

A SOAIG A can be optimized as shown in Figure 1. During optimization, AIGs of the primary output functions are considered in some order. First, a given AIG is checked whether it is redundant, that is, fully contained in the disjunction of other AIGs. If it is indeed redundant, it is removed and the computation moves on to the next AIG. Otherwise, the AIG structure is optimized by replacing it with a new structure, which is functionally equivalent to the original one on the care-set imposed by other AIGs. This optimization flow is performed as long as there are improvements or until a resource limit is reached.

```
aig optimizeAllOutputs (
   aig A,          // A is a multi-output combinational AIG
   params P )   // P is a set of parameters
{
    do {
        for each primary output aig f in A in some order {
            if ( f implies the disjunction of other AIGs of A )
                remove f from A;
            else {
                f' = optimizeOneOutput( A, f, P );
                replace f by f' in A;
            }
        }
    }
    while ( A has changed and resource limits are not reached );
    return A;
}
```

Figure 1: Iterative SOAIG optimization.

The computation of a new logic structure for an AIG with a care set is performed by procedure optimizeOneOutput() shown in Figure 2 and illustrated in Figure 3.

```
aig optimizeOneOutput (
   aig A,          // A is a multi-output combinational AIG
   aig fi,         // f is a single-output AIG contained in A
   params P )   // P is a set of parameters
{
    create two copies of A dependent on variable sets, x1 and x2;
    compute the initial set D of divisors of fi in A;
    do {
        constrain outputs of all other AIGs fk(x), k ≠ i, in A
                to have value 0 in both copies;
        constrain the output of fi(x)
                to have different values (0 and 1) in the two copies;
        constrain the candidate divisors
                to have equal values in the two copies;
        solve the resulting SAT problem;
        if ( the problem is 'unsat' ) {
            derive interpolant fi' of fi in terms of divisor variables gj;
            return fi'(g);
        }
        else if ( the SAT problem is 'undecided' )
            return "ran out of resources";
        else if ( the SAT problem is 'sat' )
            add new divisors to D;
    }
    while ( problem is 'sat' and new divisors can be added );
    return "there is no solution";
}
```

Figure 2: AIG restructuring with a care set**.**

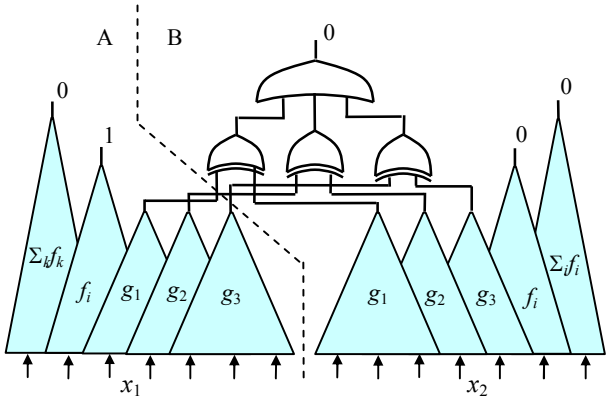

Figure 3: Illustration of AIG restructuring with a care set.

This procedure is applied to an SOAIG A and its individual AIG, $f_i(x)$, which is to be optimized. First, the SOAIG is duplicated in such a way that each copy depends on a separate set of primary input variables, x_1 and x_2. An initial set of divisors is computed. Next, the SAT problem is repeatedly constructed and solved, as shown in the loop.

If the problem is 'unsat', an interpolant derived from the proof gives the expression of f_i in terms of divisors $g_j(x)$. This expression becomes a new representation $f_i(g)$, which is returned to the caller. If the problem is 'undecided', the procedure reports that a resource limit is reached and quits. If the problem is 'sat', the given set of divisors, $g_j(x)$, lacks the expressive power to produce $f_i(x)$. New divisors are added to the set and the SAT problem is tried again, as long as new divisors can be added. If no new divisors can be added, the procedure turns "no solution".

The implementation of procedure optimizeOneOutput() uses a SAT solver and an interpolation package, which can derive an interpolant of the proof of unsatisfiability, as in the implementation of SAT-based resubstitution [12].

4. Experimental results

To be added.

5. Conclusions

Logic restructuring under don't-cares is an important practical problem. An efficient solution to this problem can improve scalability of several applications in logic synthesis and formal verification.

This paper addresses the above problem by proposing an algorithm for deriving a new circuit structure for a multi-output combinational circuit whose outputs are combined using Boolean OR. The proposed solution is based Boolean resubstitution with don't-cares and is implemented using SAT-based interpolation.

Future experiments will show whether the implemented solution is advantageous for practical applications, such as interpolation-based model checking.

A traditional SOP can be mapped into an SOAIG, by representing each product as a tree of two-input AND-gates whose leaves are literals of the product. It would be interesting to investigate whether SOP minimization can be done by optimizing the corresponding SOAIG.

6. Acknowledgements

This work was partly supported by SRC contract 1875.001 and NSA grant, "Enhanced equivalence checking in crypto-analytic applications". We are grateful to the industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Microsemi, Real Intent, Synopsys, Tabula, and Verific, for their continued support.

7. References

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] R. K. Brayton, R. L. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system", *IEEE TCAD'97*, vol. 6(6), pp. 1062-1081.

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, LNCS 6174, pp. 24-40.

[4] N. Een and N. Sörensson. "An extensible SAT-solver", *Proc. SAT'03*.

[5] N. Een and N. Sörensson. *MiniSAT*. http://minisat.se/MiniSat.html

[6] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based unbounded symbolic model checking using circuit cofactoring". Proc. ICCAD'04, pp. 510-517.

[7] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization". *IBM Journal of Research and Development*, 1974, vol. 18(5), pp. 443-458.

[8] A. A. Malik, R. K. Brayton, A. R. Newton, and A. L. Sangiovanni-Vincentelli, "Reduced offsets for two-level multi-valued logic minimization". *Proc. DAC'90*, pp. 290-296.

[9] K. L. McMillan. "Interpolation and SAT-Based model checking". *Proc. CAV'03*, pp. 1-13.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[11] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure", *Proc. IWLS '06*, pp. 15-22.

[12] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't care based logic optimization and resynthesis", *Proc. FPGA'09*, pp. 151-160.

[13] F. Pigorsch, Ch. Scholl, and S. Disch, "Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling". *Proc. FMCAD'06*, pp. 89-96.

[14] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization", IEEE Trans. CAD, Vol. 6(5), pp. 727-750, Sep. 1987.