Building Circuits from Relations

James H. Kukula[1] and Thomas R. Shiple[2]

[1] Synopsys, Inc., Beaverton, OR. `kukula@synopsys.com`
[2] Synopsys, Inc., Mountain View, CA. `shiple@synopsys.com`

Abstract. Given a Free BDD for the characteristic function of an input-output relation $T(\boldsymbol{x}, \boldsymbol{y})$, we show how to construct a combinational logic circuit satisfying that relation. Such relations occur as environmental constraints for module specifications, as parts of a proof strategies, or can be computed from existing circuits, e.g., by formal analysis of combinational cycles. The resulting circuit C can be used for further analysis, e.g. symbolic simulation, or to reformat a circuit as a logic optimization tactic.
The constructed circuit includes supplementary parametric inputs to allow all legal outputs to be generated in the case that T is non-deterministic. The structure of the circuit is isomorphic to that of the BDD for T, and hence is as compact as the BDD. In particular, when T represents a relation between bit vector integer values definable in Presburger arithmetic, the constructed circuit will have a regular bit slice form.

1 Introduction

A general Boolean relation $T(\boldsymbol{x}, \boldsymbol{y})$ admits multiple interpretations and representations arising in various contexts. We consider the case when the $\boldsymbol{x}$ variables are considered to be inputs presented to some system component, and $\boldsymbol{y}$ are output variables that the system component generates, subject to the constraint that the given $\boldsymbol{x}$ and the generated $\boldsymbol{y}$ must satisfy the T relation. The problem we address is to construct a combinational circuit satisfying the input-output relation represented by a Free Binary Decision Diagram (FBDD). Since an FBDD is a generalization of the more common Ordered BDD (OBDD), such a construction will also work for OBDDs.

The primary context we consider is that of a verification constraint or precondition. In this case, the input variables $\boldsymbol{x}$ encode the state of some module under verification, and the output variables $\boldsymbol{y}$ provide the input stimulus to that module. The set of stimuli to be presented to the module, in general or for a particular verification task, may depend on the current state of the module. For example, Yuan et al. [19] describe a verification methodology where a module's environment is specified as a constraint which can depend on the state of the module. Jain and Gopalakrishnan's methodology [12] also includes "action" constraints which depend on the system state. These constraints are used as a verification tactic rather than a specification of the module's environment. Aagaard et al. [1] also use tactical constraints, but their constraints do not depend on the state of the module and thus are a special case of the more general one we consider.

Input-output relations may arise in other contexts. For example, such a relation may be derived from a combinational logic circuit, summarizing the behavior of the circuit.

In such a case the input-output relation will be complete (for each $\boldsymbol{x}$ at least one $\boldsymbol{y}$ exists that satisfies T) and deterministic (for each $\boldsymbol{x}$ at most one $\boldsymbol{y}$ exists that satisfies T). One can also analyze the context of a subcircuit to construct a nondeterministic input-output relation for the allowable behaviors of the subcircuit which will preserve the overall behavior of the containing complete circuit [18]. Another use of input-output relations is in the analysis of cyclic circuits built of combinational gates. Such circuits may reliably settle their output values to a deterministic function of their inputs despite their cyclic topology. The constructivity analyses of Shiple [17] and Namjoshi et al. [15] generate Boolean relations (output bit by output bit in Shiple's analysis) that represent the combinational function of cyclic circuits, along with checking whether those cyclic circuits are indeed not state holding.

Input-output relations are also used as a means to express the intended function of a machine being designed in high level languages such as SMV [14]. One advantage of using relations for design is the natural representation of non-determinism.

A Boolean relation $T(\boldsymbol{x}, \boldsymbol{y})$ may be represented in various ways. A OBDD [8] can be used to represent the characteristic function of the relation. A generalization of the OBDD representation is the FBDD [9], where variables may occur in different orders on different paths from root to terminal. Both FBDDs and OBDDs are constrained so that variables occur at most once on any path. The added flexibility of FBDDs permits a much more compact representation for some relations.

Another possible representation is as a multiple output combinational logic circuit with inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. If the relation T is complete and deterministic then the required values of the outputs are well defined, and can be expressed as the positive cofactors of the bitwise characteristic functions:

$$(\exists_{j \neq i} y_j . T(\boldsymbol{x}, \boldsymbol{y})) \mid_{y_i}$$

In the general case, a given value of $\boldsymbol{x}$ might be related to multiple $\boldsymbol{y}$ values or to none. One can supplement a circuit with two additional features to allow it to accurately represent a general input-output relation $T(\boldsymbol{x}, \boldsymbol{y})$. To handle incompleteness, one can add an extra output $v(\boldsymbol{x}) = \exists \boldsymbol{y}.T(\boldsymbol{x}, \boldsymbol{y})$ to the circuit, indicating whether any $\boldsymbol{y}$ exists that is related to a given input $\boldsymbol{x}$. To handle non-determinism one can add extra parametric inputs $\boldsymbol{p}$ to the circuit, so that for every output value $\boldsymbol{y}$ that satisfies $T(\boldsymbol{x}, \boldsymbol{y})$ for a given input $\boldsymbol{x}$, there is some value of $\boldsymbol{p}$ such that the circuit will generate $\boldsymbol{y}$ when applied to the inputs $(\boldsymbol{x}, \boldsymbol{p})$.

A multiple output combinational logic circuit provides a broadly applicable represention for T. As Jain and Gopalakrishnan [12], Aagaard et al. [1], and Bertacco et al. [3] point out, symbolic simulation is a powerful technique for exploring the behavior of a circuit. Symbolic simulation can be directly applied to the combinational circuit representation of T. Other state exploration engines such as those based on SAT [4] or ATPG [5] generally accept combinational logic circuits as a problem representation. Logic emulation hardware is another state exploration mechanism for which a combinational logic circuit is an ideal problem representation.

When T represents a constraint on the inputs to some module under verification, the outputs of the circuit we construct for T would be connnected to the inputs of the module, and the composite circuit submitted to state exploration. While the circuit we construct does not reduce the number of input variables compared to the unconstrained circuit, the constraints on the inputs can prevent false error reports that could have occurred if improper input stimuli were allowed to propagate into the module [19]. The constraints may also improve the efficiency of state exploration techniques such as symbolic simulation by reducing BDD sizes [1].

A combinational logic circuit also provides a structure for implementation in digital hardware. In this case, the relation to be implemented should be complete, so the v output should be constant 1 and can be ignored. An implementation will also generally be deterministic. If the input-output relation to be implemented is non-deterministic, the supplementary inputs $\boldsymbol{p}$ in the non-deterministic circuit representation can be connected to arbitrary constants or variable signals to form a deterministic circuit.

Our contribution is an elegant translation procedure that constructs a combinational logic circuit, with inputs $\boldsymbol{x}$ and $\boldsymbol{p}$ and outputs $\boldsymbol{y}$ and $\boldsymbol{v}$, from a general input-output relation $T(\boldsymbol{x}, \boldsymbol{y})$ represented as the Free BDD of its characteristic function. The size of the circuit is proportional to the number of nodes in the Free BDD. When the input-output relation is non-deterministic, the supplementary parametric inputs $\boldsymbol{p}$ are used to index all the output $\boldsymbol{y}$ values related to a given input $\boldsymbol{x}$.

In the remainder of this paper, we will first discuss the details of the construction procedure. We will then demonstrate that the circuit constructed does effectively represent the input-output relation. Next we address the compactness of the circuit. Finally we review related work and conclude.

2 Circuit Construction Procedure

Given an FBDD for an input-output relation $T(\boldsymbol{x}, \boldsymbol{y})$, we construct a circuit implementing T that has the same top level topology as the FBDD. First we describe the high level structure and signal flow of the circuit. We will then discuss the internal details of each of the modules that compose the circuit.

2.1 High Level Signal Flow

For every node in the BDD for T there is an instantiation of a basic module. The basic modules come in two types, corresponding to the two classes of variables that occur in the BDD. There is an input module that is used in place of BDD nodes labeled by input variables, and an output module used in place of nodes labeled by output variables. The connections between modules are created to match the edges between the corresponding BDD nodes.

We describe the construction process in terms of an example. Suppose we are given the BDD shown in Figure 1. The circuit shown in Figure 2 provides a combinational logic representation for that relation. The inputs to the circuit are at the bottom of the figure, with two main input variables x_1 and x_2, and two supplementary parametric inputs p_1 and p_2. There are many possible ways to parameterize or encode the $\boldsymbol{y}$ in terms of a set of parameters $\boldsymbol{p}$. In our encoding we use one parameter input bit for each output bit. The outputs are at the top of the figure, with the two main outputs y_1 and y_2 and the supplementary output v.

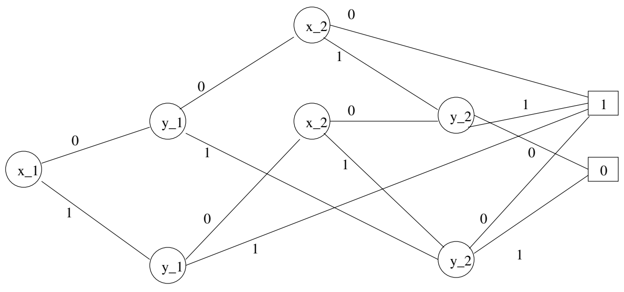

Fig. 1. Binary Decision Diagram Representing Input-Output Relation

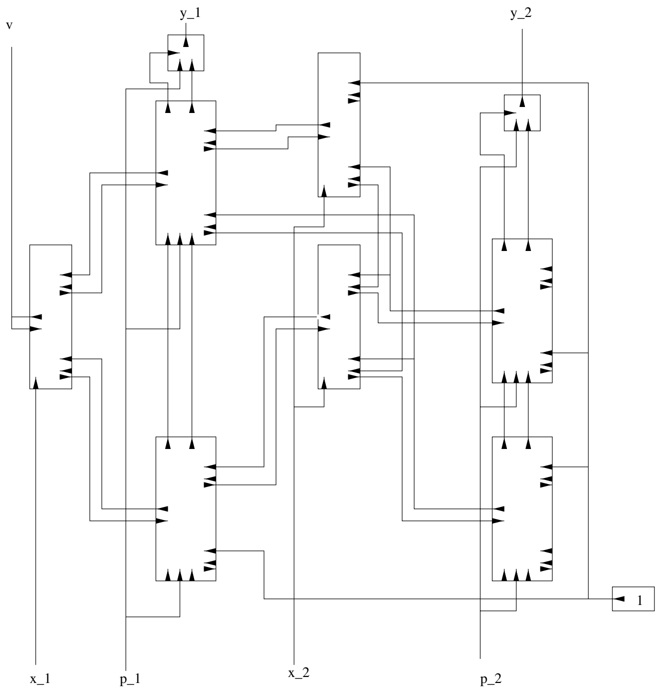

Fig. 2. Circuit Implementing Relation. Disconnected inputs are to be driven by constant 0.

Just as there are seven nodes in the BDD, there are seven major modules in the circuit. In this top level diagram, each connection point on the modules is marked with an arrow to indicate the direction of signal flow. Disconnected inputs are driven by logic value 0. Each edge in the BDD corresponds to two wires in the circuit, one that flows from terminal to root and one that flows from root to terminal.

The overall logic flow in the circuit can be broken into three phases. The first phase flows from the terminal nodes to the root node, the second from the root node back to the terminal nodes, and the third phase across all the nodes labelled by the same variable.

In the first phase, constant 0 and 1 values corresponding to the terminal nodes start to flow toward the root, combining with the circuit inputs $\boldsymbol{x}$ along the way. This terminal-to-root flow results in the v signal which is the auxiliary circuit output, indicating whether any valid circuit outputs are possible for the particular values being presented at the inputs $\boldsymbol{x}$. In this first phase, each node will receive signals from the modules corresponding to the destination nodes of its two outgoing edges, indicating whether those two nodes have any path to the 1 terminal consistent with the presented values of the primary circuit inputs.

In the second phase, signals propagate from the root to the terminals to activate a single path from root to terminal. This path is steered by the primary circuit inputs and also by the auxiliary inputs. Each node receives a signal that indicates whether any of its incoming edges are active. If an incoming edge is active, then the current node is active and must choose which outgoing edge to activate. If the node is labeled by an input variable, then the node is constrained to choose as directed by the value of that variable. If the node is labeled by an output variable, then the node will choose as directed by the corresponding auxiliary input variable if possible. During the first phase of signal propagation the node received signals from the two destination nodes which indicated which of them had possible paths to the 1 terminal. The node can then use these signal values to be sure to choose a valid value for the primary circuit output signal, one that can form part of an unbroken path from the root of the BDD to the 1 terminal.

In the third phase, the value of the circuit outputs $\boldsymbol{y}$ are computed, based on the activated path. If the activated path includes a node labelled by a particular output bit y_i, then the edge of that node followed by the path will fix the value of the output bit. If the activated path does not include y_i, then the value of the corresponding parametric input p_i is used. The modules substituted for the nodes labelled by a single output variable y_i are connected together in a serial chain. There are two logic signals propagated along this chain. The order of the nodes in the chain is arbitrary. This chain gathers information to compute the value of y_i. For each output variable we also include a single multiplexor to handle the case where the activated path does not include a node labelled by that variable. So in this circuit there are two multiplexors, shown near the top of Figure 2, corresponding to the two output variables y_1 and y_2. If the BDD for an input-output relation doesn't include any nodes at all for a particular output variable, then the value of that output variable is not constrained by the inputs, and can simply be copied directly from the corresponding supplementary parameter variable. In this case no multiplexor is needed.

Another implementation detail arises in handling multiple edges leading to the same destination node. In the circuit we build, this is translated, in part, to a collection of signals whose disjunction drives the module corresponding to the destination node. We implement this here with a chain of OR gates, one in each of the modules corresponding to the sources of the edges which all have the same destination node.

2.2 Input Module

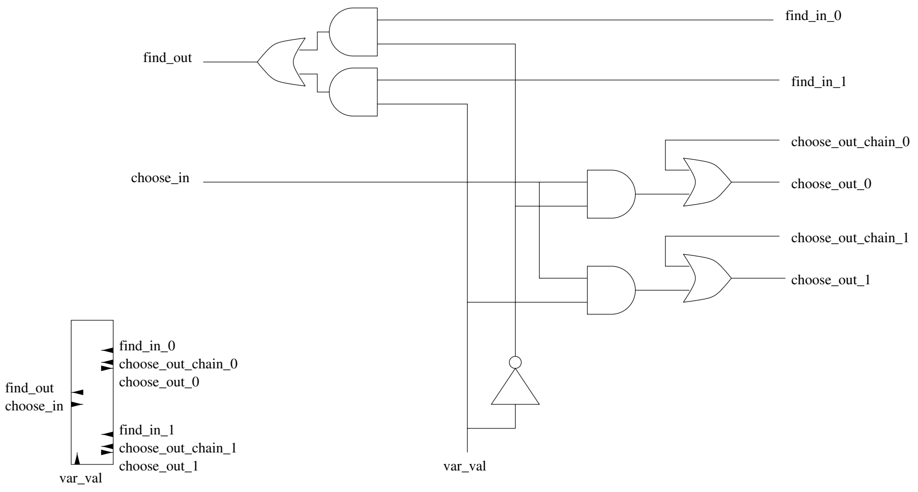

Fig. 3. Module for Input Node

Figure 3 shows the internal details of the module to be substituted for each BDD node labelled with a circuit primary input x_i. The upper part of the circuit is a multiplexor whose data inputs are the signals from the two outgoing edge destination modules. The multiplexor control is the signal x_i. If there is a compatible path to the 1 terminal along the edge labeled by the present value of x_i, then there is a compatible path from this node to the 1 terminal.

The lower part of the circuit steers the active path in the second phase. If this node is marked as active by one of its incoming edges, then it activates one of its destination nodes as chosen by the value of x_i. The OR gates at the outputs work with the other nodes that have edges to the same destination, so that if any of these source nodes activate their edges to that destination, then the activation will reach that node.

2.3 Output Module

Figure 4 shows the module to substitute for a BDD node labelled by a primary output variable y_i. Again the upper part is for the first phase of propagation and the lower part for the second phase. In the first phase the circuit computes the paths through the BDD allowed by the current values of the inputs $\boldsymbol{x}$ before the output values are picked. There is a path to the 1 terminal through this output node compatible with the $\boldsymbol{x}$ inputs if there

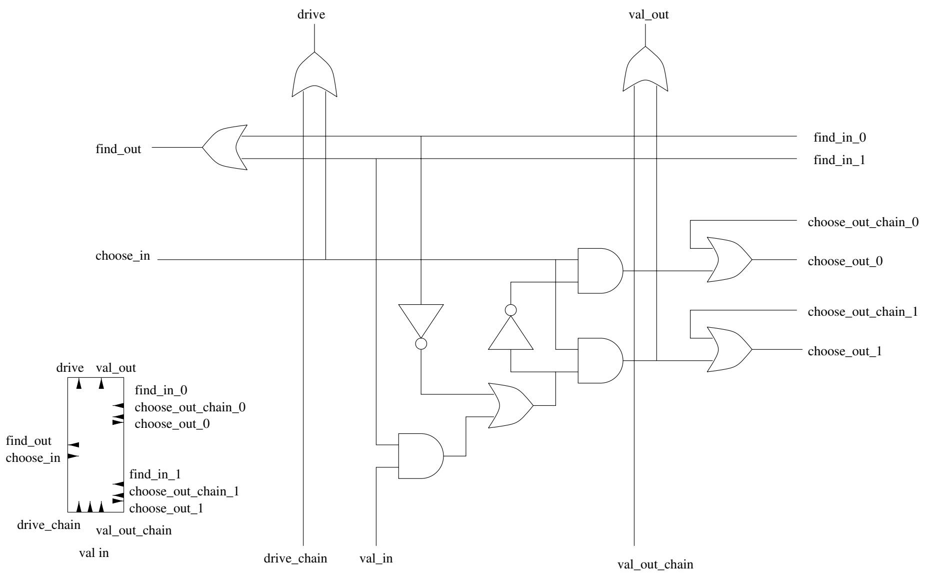

Fig. 4. Module for Output Node

is a path from either of its destination nodes. So a simple OR gate is enough for the first phase.

In the second phase, if an incoming edge to the module is activated then it must choose an outgoing edge to activate. The val_in signal to the module is driven in the top level circuit by the parametric input p_i. If both outgoing edges indicate the existence of paths to the 1 terminal, then the edge suggested by p_i will be activated. If only one edge has a path to the 1 terminal, then that edge will be activated and the p_i value is ignored.

The value for y_i will be computed in the third phase of computation in the circuit. The OR gate at the top right of the diagram works together with all the other nodes labeled by y_i. If any of these nodes has been activated and has chosen the 1 value, then the output should be driven to 1. Otherwise the 0 value should be chosen.

The OR gate at the top left accumulates a value for y_i indicating whether any of the BDD nodes labelled by y_i were activated. This value is then fed to the multiplexor which chooses the final value for y_i. If a node was activated, then the value determined by that node and passed along through the value chain should be chosen. If no node was chosen, this indicates that an edge was activated that skipped over the y_i. In this case its value is unconstrained by the present values of the $\boldsymbol{x}$ inputs, and the value of p_i should be chosen.

3 Circuit Correctness

Let $T_C(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{y}, v)$ denote the input-output relation of the circuit constructed from the Boolean relation $T(\boldsymbol{x}, \boldsymbol{y})$. In this section we prove that T_C correctly implements T. We start with two lemmas, whose proofs use induction based on a topological ordering of the nodes of the FBDD. Each node n in an FBDD represents some Boolean function $f_n(\boldsymbol{x}, \boldsymbol{y})$.

Lemma 1. *For a given $\boldsymbol{x}$, the "find_out" signal from a module n has the value 1 iff, for some $\boldsymbol{y}$, $f_n(\boldsymbol{x}, \boldsymbol{y}) = 1$.*

Proof (sketch): This is clearly true for a node whose edges both lead to terminal nodes. If the find_in_0 and find_in_1 edges reflect the existence of a path to the 1 terminal, then each module will in turn determine if a path exists flowing through the corresponding node. Thus by induction, for all modules find_out reflects the existence of a path. ■

Since the v output is given by the find_out signal of the module corresponding to the root node, this shows the value of v for a given $\boldsymbol{x}$ is $\exists \boldsymbol{y}.T(\boldsymbol{x}, \boldsymbol{y})$.

Lemma 2. *The "choose_in" signals will activate a single path through the BDD from the root node to the 1 terminal, if any such path exists for the given input $\boldsymbol{x}$.*

Proof (sketch): A topological order of the BDD nodes determines a series of cuts which partition the nodes into a root set and a terminal set, where any BDD edge that crosses the cut will be directed from a node in the root set to a node in the terminal set. We can prove inductively that the number of activated edges crossing any cut is exactly 1 if $\exists \boldsymbol{y}.T(\boldsymbol{x}, \boldsymbol{y})$, and 0 otherwise. The induction starts with the base case of the cut with all the BDD nodes on the terminal side, where the root edge coming into the root node is activated just in case $\exists \boldsymbol{y}.T(\boldsymbol{x}, \boldsymbol{y})$ is true. Now we assume that the i'th cut has a single activated edge crossing it, and show that the $i + 1$'th cut will also be crossed by a single activated edge. Each node will activate a single outgoing edge if the incoming edge is activated, or neither outgoing edge if the incoming edge is not activated. Thus each node n preserves the number of activated edges crossing the cuts just before and after n. ■

With these basic properties of the circuit established, we can now prove the correctness of T_C.

Theorem 1. *The input-output relation $T_C(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{y}, v)$ for the circuit built from the relation $T(\boldsymbol{x}, \boldsymbol{y})$ satifies:*

$$\begin{aligned} T(\boldsymbol{x}, \boldsymbol{y}) &\Rightarrow T_C(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{y}, 1) \\ T_C(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{y}, 1) &\Rightarrow T(\boldsymbol{x}, \boldsymbol{y}) \end{aligned}$$

Proof (sketch): Since any path from root to terminal includes at most one node labelled by any given output variable y_i, then the third phase will propagate to the output y_i the value computed corresponding to the outgoing edge from that node (or the corresponding p_i if no such node is included). Thus

$$T_C(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{y}, 1) \Rightarrow T(\boldsymbol{x}, \boldsymbol{y})$$

Since each output module attempts to steer the path to follow the choices suggested by the parametric input $\boldsymbol{p}$, the path activated will drive the outputs to $\boldsymbol{p}$ if $T(\boldsymbol{x}, \boldsymbol{y})$ holds. This together with the value of v shows that

$$T(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow T_C(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{y}, 1)$$

■

4 Circuit Compactness

Since the circuit constructed by this technique has the same high level topology as the BDD of the input-output relation, its size is proportional to the BDD. The circuit generated is not at all locally optimal. There are many constant inputs to modules, such as the constants corresponding to the BDD terminal nodes and the starting points of the various node chains. There are also disconnected outputs of modules, principally at the last edges of the second pass root-to-terminal propagation, since the activation signal does not have to be propagated to the terminal nodes. These constants and disconnected outputs provide straightforward opportunities for simple local logic optimization, but other more sophisticated techniques could also be applied. As long as the input-output relation's BDD is reasonably compact, the circuit we construct should provide an efficient high level structure and a good starting point for such low level optimization, which could be followed by mapping to a specific technology if the circuit is to be manufactured as digital hardware.

In order to generate a more efficient circuit, before converting the BDD to a circuit one could apply exact [11] or approximate [16] variable reordering techniques to attempt to reduce the size of the BDD. Since the circuit construction procedure we provide here also applies to the Free BDD representation of an input-output relation, one could further reduce the size of the circuit by exploiting the freedom to use different variable orderings on different branches of the diagram [10].

Kukula et al. [13] observe that a relation between radix-encoded integers definable in Presburger arithmetic will have a compact, regularly-structured OBDD so long as the variable ordering interleaves the bits in the order of the encoding weights. In this case the circuit constructed will have a bit slice form, a linear array of repeated instances of a single module. Such a circuit is not only efficient in terms of gate count but also lends itself to an efficient physical layout.

5 Related Work

Brown [7] discusses parametric general solutions for Boolean equations. His method of successive elimination will give the same parametric functions as implemented by the circuit we construct in the special case of an OBDD. Brown's methods deal with general Boolean functions rather than specific circuit implementations or BDD representations, and in particular he does not address the issue of circuit size.

Our construction technique is most closely related to the stimulus generation algorithm of Yuan et al. [19] and the parametric constraint representation of Aagaard et al. [1]. Yuan et al. present an algorithm which generates random stimuli satisfying a constraint, represented as a BDD, which may depend on state variables of the design. Our circuit has a flow very similar to their algorithm. The main differences between their work and ours are that our technique constructs a compact circuit rather than generating a single stimulus instance, and the output value selection of our circuit is controlled by parametric inputs, rather than weighted random numbers as in Yuan et al. The circuit constructed by our technique can be used by a wide variety of downstream tools such as SAT or symbolic simulation.

Aagaard et al. present an algorithm which generates a vector of OBDDs over a set of parametric input variables; the combinations of values of these BDDs span the space of stimulus values which satisfy some constraint. The constraints they deal with do not involve any dependence on state variables in the design, and hence their technique is limited to unary relations. In contrast, we work with the more general problem of binary relations. Another difference is that their algorithm generates a parametric result in the form of OBDDs. Some relations will not admit a tractable parametric representation as a vector of OBDDs. Since we map directly from a Free BDD representation of the input-output relation to a circuit, our technique can be used with a broader range of relations.

Other related work includes synthesis of multiplexor circuits from BDDs [2]. In that work, a multi-rooted BDD defines the vector of output functions to be implemented. Our technique differs, working instead with the input-output relation and also in working with incomplete and/or non-deterministic functions. Synthesis from the input-output relation can result in circuits considerably more compact than those built from the multi-rooted functional BDD used in multiplexor synthesis. Consider the arithmetic function $\max(\boldsymbol{a}, \boldsymbol{b}+\boldsymbol{c})$, where $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are n-bit vectors representing integers in the usual radix-2 encoding. Each output bit of this function can be represented by a BDD with size bound $O(n)$, by using an interleaved variable ordering. With n output bits to be represented, the total shared size of the multi-rooted BDD will be quadratic in n unless there is significant node sharing across the multiple outputs. But there is a conflict between the variable orders required by the max and addition functions if nodes are to be shared. With the low-order bits ordered at the top, the max function will give a compact multi-rooted BDD representation, since the high-order bit nodes at the bottom of the BDD can be shared by all the low order nodes. However, efficient multi-rooted representation of addition requires the low order bits at the bottom to be shared by all the high order nodes. Whichever order is chosen, one function or the other will fail to share the nodes at the bottom of the BDD. Thus the entire multi-rooted BDD will end up being quadratic in the bit-width. In contrast to this, the circuit constructed by our technique will grow only linearly with the bit-width since the input-output relation is definable in Presburger arithmetic.

6 Conclusion

We have presented a simple and direct mapping from a Free BDD representing an input-output relation $T(\boldsymbol{x}, \boldsymbol{y})$ to a compact combinational circuit. This mapping supports both incomplete and non-deterministic relations by means of a supplementary output signal and a set of supplementary parametric inputs. The combinational circuit provides a flexible representation which can be used for verification or synthesis.

The usefulness of the OBDD representation has led to a variety of extensions. We have presented our circuit construction technique in terms of the more general Free BDD representation. Another common extension is to add various attributes to the BDD edges, for example complementation [6]. Edge complementation can reduce BDD size by up to a factor of two. It is quite possible to construct a circuit directly from a BDD with complemented edges, but the modules required grow by about a factor of two, so there doesn't appear to be any advantage. Our next steps in this research will be to investigate other extensions to OBDDs to see which of them can support effective circuit construction techniques.

References

[1] M. Aagaard, R. Jones, and C.-J. Seger. Formal Verification Using Parametric Representation of Boolean Constraints. In *Proc. of the Design Automation Conf.*, pages 402–407, June 1999.

[2] P. Ashar, S. Devadas, and K. Keutzer. Gate-delay-fault Testability Properties of Multiplexor-Based Networks. *Formal Methods in System Design*, 1:93–112, Feb. 1993.

[3] V. Bertacco, M. Damiani, and S. Quer. Cycle-based Symbolic Simulation of Gate-level Synchronous Circuits. In *Proc. of the Design Automation Conf.*, pages 391–396, June 1999.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems*, 1999.

[5] V. Boppana, S. Rajan, K. Takayama, and M. Fujita. Model Checking Based on Sequential ATPG. In *Proc. of the Computer Aided Verification Conf.*, pages 418–430, July 1999.

[6] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proc. of the Design Automation Conf.*, pages 40–45, June 1990.

[7] F. M. Brown. *Boolean Reasoning*. Kluwer, 1990.

[8] R. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, Aug. 1986.

[9] J. Gergov and C. Meinel. Efficient Boolean Manipulation with OBDD's Can Be Extended to FBDD's. *IEEE Transactions on Computers*, 43:1197–1209, Oct. 1994.

[10] W. Günther and R. Dreschler. Minimization of Free BDDs. In *Proc. Asia and South Pacific Design Automation Conference*, Jan. 1999.

[11] N. Ishiura, H. Sawada, and S. Yajima. Minimization of Binary Decision Diagrams Based on Exchanges of Variables. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 472–475, Nov. 1991.

[12] P. Jain and G. Gopalakrishnan. Efficient Symbolic Simulation-Based Verification Using the Parametric Form of Boolean Expressions. *IEEE Transactions on Computer-Aided Design*, 13:1005–1015, Aug. 1994.

[13] J. Kukula, T. Shiple, and A. Aziz. Techniques for Implicit State Enumeration of EFSMs. In *Proc. of the Formal Methods in CAD Conf.*, pages 469–482, Nov. 1998.

[14] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[15] K. Namjoshi and R. Kurshan. Efficient Analysis of Cyclic Definitions. In *Proc. of the Computer Aided Verification Conf.*, pages 394–405, July 1999.

[16] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 42–47, Nov. 1993.

[17] T. Shiple. *Formal Analysis of Synchronous Circuits*. PhD Diss., Univ. Calif. Berkeley, 1996.

[18] Y. Watanabe, L. Guerra, and R. Brayton. Permissible Functions for Multioutput Components in Combinational Logic Optimization. *IEEE Transactions on Computer-Aided Design*, 15:732–744, July 1996.

[19] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 584–589, Nov. 1999.