Binary Decision Diagrams and automated verification

Kevin Brewer

January 21, 2004

Abstract

This report serves as an overview of the use of Ordered Binary/Boolean Decision Diagrams (OBDDs) in automated verification. The history, key developments and current state of the problems involved will be discussed.

1 BDDs: An overview

There are many ways of representing an arbitrary boolean function f that maps inputs $x_1 \ldots x_n$ to $\{0, 1\}$ including circuits, truth tables, boolean algebra. The problem with most of these representations is either they are exponentially large (such as truth tables) or do not help in finding a satisfiability in a guaranteed reasonable time (such as boolean functions).

With the introduction of Ordered Binary Decision Diagrams (OBDDs), a form of Binary (also called boolean) Decision Diagrams, a format that offers efficient satisfiability and the possibility of being of reasonable size, computer scientists began to use it in a field that depends heavily on solving and comparing boolean functions; circuit checking. Eventually this extended into model checking, thus creating an entire field of research. Below is an overview of how OBDDs came to be used in model checking.

The central form of a BDD is a directed acyclic graph. Each node represents a particular variable in the function/circuit; the two successors (sometimes called the 0 and 1 successors) represent the status of the function if the value of the variable is a 0 or 1. The DAG ends with 0 or 1-terminals, which indicate whether the functions returns 0 (false) or 1 (true).

1.1 Early forms of BDDs

Bryant [8] cites an article by Lee [24] as the first article to use notation for binary decision trees that resembles BDDs. Further work by Akers [2] in the area of binary decision programs popularized the concept as well. In principle, the concept was simple; to model programs as a series of binary decisions, a concept as old as branching programs. Prior to 1986, however, these structures were not of great practical value for verification.

1.2 Bryant's work

In 1986, however, Bryant [8] published a watershed paper on the use of OBDDs (as a graph-based data structure) with the specific purpose of automated verification. The key introduction in this paper was the notion of the *Ordered* Binary Diagram, a variant on the BDD but a very useful modification. In fact, most of the literature uses the term BDD to refer to an OBDD, since BDDs themselves have shown little promise in model-checking work (as discussed in Section 1.5.1).This paper included the following key proofs and algorithms for using OBDDs and how they would assist in automated verification:

1.2.1 Canonical form

Possibly the most significant result is that all boolean function mappings are represented by a unique up to isomorphism (canonical) OBDD for a given variable ordering, regardless of the formula/circuit used to represent the function. Therefore, two different formulae are equivalent iff their OBDDs are isomorphic. Though graph isomorphism has neither been proven to be in P or NP-complete, a comparison of OBDDs is possible in time linear to the number of nodes in the reduced OBDD given the acyclic and rooted nature of the graph, plus the existence of a clear labelling of the nodes and edges with their input label and values respectively. Therefore, an isomorphism check involves starting at the first input for the two OBDDs to be compared, checking for equality of nodes and edges, and recursing down the same paths and comparing the values.

At the time of this paper, this served mostly to verify that a circuit implemented in hardware matched the specification; temporal-based queries would come later (see Section 2.3). Still, this method for verifying a circuit provided an important step forward in verification.

1.2.2 Reducing BDDs to OBDDS

Given the canonical form of OBDDs, an efficient algorithm was needed to guarantee a minimal reduction for the given variable ordering. Bryant provides an $O(|G| \log |G|)$ algorithm where $|G|$ is the number of nodes in the BDD (in an update to Bryant's original paper, Bryant notes that in 1993 an $O(|G|)$ algorithm for reduction was shown by Wegener and Sieling [36]).

Bryant uses an algorithm for tree isomorphism (from Aho, Hopcroft and Ulman [1]) as the basis for the reduction algorithm; starting at the terminal nodes, if the node and successors have not been seen yet, it is given a new key value (otherwise, it is assigned the same key value as the other node).

After labelling each node with its key value (and the key values of the 0 and 1 valued successors) the algorithm performs one of two reductions from the bottom up; if the 0 and 1 values point to the same key, then this key can be removed. Secondly, if two different nodes point to the same keys for their 0 and 1 valued children, then these two nodes can be merged as well. Once the graph has been traversed from the terminal nodes up to the top, the remaining keys are sorted (using a bucket sort-type algorithm) and you have the reduces OBDD. This will result in a reduced BDD in $O(|G| \log |G|)$ time. Figure 1 shows the two reduction rules.

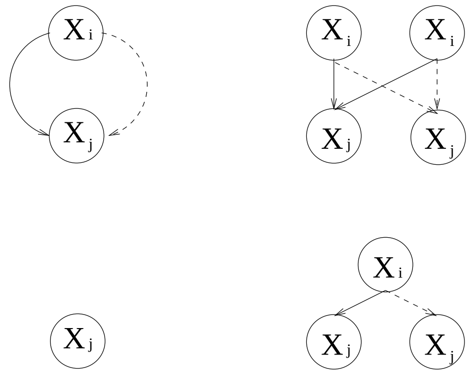

Figure 1: Left and right: two different rules for reduction of BDDs to OBDDs

1.2.3 Satisfiability of OBDDS

Once an OBDD has been reduced, simple satisfiability (i.e. whether there exists an assignment to inputs $x_1 \dots x_n$ that makes satisfies the boolean function in question) is a matter of seeing that there is more than the 0-terminal in the graph, so finding the existence of an assignment is $O(1)$. For finding a single satisfying assignment, the complexity is $O(n)$, where n is the number of inputs. The reason that only $O(n)$ nodes need be visited is due to the reduced nature of the graph; given the rules of reduction, any node that is not a 0-terminal is guaranteed to have beneath it some satisfying assignment (otherwise it would be reduced to a 0-terminal) so the algorithm need only backtrack at most once per input.

For deciding tautology and contradiction, one must only check whether the graph consists of a 1-terminal or 0-terminal respectively, thus making the test $O(1)$.

As for *satisfy-all* (that is, the set all inputs that satisfy the function) the complexity is $O(n|S_f|)$ where $|S_f|$ is the number of satisfying assignments, in the worst case. The n is due to having to do a similar traversal in the basic satisfiability situation, while the $|S_f|$ accounts for having to possibly do many traversals if there are a large number of satisfying solutions for the function in question.

Finally, a count of all satisfying solutions is an $O(|G|)$ operation, using simple bookkeeping to determine how many assignments correspond to a given 1-terminal.

1.2.4 The function *Apply*

The *Apply* function simply takes two reduced OBDDs and combines them with a two argument binary operator. The flexibility of this function allows for a single algorithm where other representations require several. The algorithm is generally implemented either recursively or using dynamic programming and was initially believed to be $O(|G_1| \bullet |G_2|)$ where $|G_1|$ and $|G_2|$ are the sizes of the respective OBDDS. An update to the initial paper by Bryant [8] indicated that a logarithmic overhead factor of $\log |G_1| + \log |G_2|$ was needed to reduce the combined OBDDs. This overhead factor was not needed once a linear time reduction algorithm was found by Wegener and Sieling [36].

Regardless of the complexity of such an operation, it should be clear that the complexity of this operation is directly related to the size of the two OBDDs. With reasonable (i.e. polynomial-sized) graphs, this is a very efficient operation for generic OBDD manipulation.

1.2.5 The function *Restrict*

When one applies the *Restrict* function to an OBDD, one is fixing one or more of the inputs to a given value, thus eliminating it from the OBDD and reducing the OBDD further. Notated, this would be $f|x_i = b$ for $b \in \{0, 1\}$. This algorithm is fairly simple in concept; traverse the graph until the algorithm that reaches a node that points to x_i. When it does so, it replaces that edge with the edge that x_i points to for the value b as described before. After traversing the graph, the OBDD is reduced again. When OBDD reduction was considered to take $O(|G| \bullet \log |G|)$ time, *Restrict* took $O(|G| \bullet \log |G|)$ time as well, but with the later result by Wegener and Sieling [36] it now only takes $O(|G|)$ time.

1.2.6 The function *Compose*

In composing, one is substituting a second function f_2 for each x_i in the inputs. Bryant notes that composition can be done by using *Apply* and *Restrict* using the Shannon expansion:

$$f_1|_{x_1=f_2} = f_2 \bullet f_1|_{x_1=1} + (\neg f_2) \bullet f_1|_{x_1=0}$$

This works, but Bryant notes it takes $O(|G_1|^2 \bullet |G_1|^2)$ time this way, where G_1 and G_2 are the OBDDs representing the two functions.

With some modifications (mostly using a three-argument version of *Apply*) this was reduced to an algorithm that ran in what was believed to be $O(|G_1|^2 \bullet |G_2|)$ time in the worst case; this bound was later proven by Jain et al. [22]. Though quadratically bounded by the first element, this is still an improvement on the previous bound and reasonable for polynomially-sized graphs.

1.3 Limitations of OBDDs

Given their ability to solve satisfiability in $O(n)$ time (with n being the number of nodes in the reduced OBDD), there must be something that prevents OBDDs from being able to solve SAT in polynomial time. Simply put, the number of nodes in an OBDD can be exponential in size to the number of inputs, depending on the type of function being modelled and the particular variable ordering. The size of the reduced graph can range from quadratic up to exponential in size, thus necessitating some work to try and ensure that the variable ordering does not produce an exponential number of nodes.

Bryant notes, however, that some functions (like multiplication) will produce an exponential number of nodes *regardless of the variable ordering.*In an appendix to the original paper [8], Bryant shows that no variable ordering on a OBDD for a multiplier circuit will result in less than $2^{\frac{n}{8}}$ vertices in the resultant graph.

1.4 SAT solvers versus BDDs

With the introduction of SAT solvers that work well in many cases (considered to be introduced by Biere et. al [5]) the question arose of when a BDD should be used and when SAT solvers should be used. As noted by Langberg, Pnuelli and Rodeh [23], Groote and Zantema [18] have shown cases where SAT solvers can work in polynomial time while OBDDs are exponential in size, and also show that there exists polynomial-sized OBDDs with exponential size CNF formulas. Therefore, there will always be situations where one is preferable to the other.

In the same paper by Langberg, Pnueli and Rodeh [23], random CNF formulas were tested for their OBDD size. With random monotone 2CNF formulas (which are solvable in polynomial time), they proved that as 2CNF formulas grow to non-trivial size, the OBDD representing them grows exponentially. Though a somewhat degenerate case (because of the selection of 2CNF formulas which are guaranteed to be solvable in polytime by SAT solvers), it still demonstrates a clear advantage of SAT solvers in some circumstances.

As a counterpoint, Cabodi, Camurati and Quer [10] present some experimental work comparing one OBDD-based verifier based on algorithms in [15] versus SAT solvers in NuSMV [11] and one called Chaff [28]. By employing both forward and backward checking combined with both exact and approximate checking (among other techniques) they showed in trials that a computer could solve problems of increasing size with BDDs while eventually growing too large for SAT solvers. Like the previous paper, this is not exactly a fair contest - the amount of extra work on the BDD-based system over the SAT-based systems leaves some question as to whether the same amount of work could aid the SAT-solvers, but it still shows areas where BDDs are still useful.

In practice, it generally makes sense to attempt to use both at the same time, since it's better than waiting to find out that one is exponential in length and the other polynomial. That being said, competition between BDD and SAT-based fields results in a collective improvement in both methods due to both sides attempting to best the other, making the debate a useful one.

1.5 Variants of OBDDs

Besides OBDDs, there have been several variations in the literature designed to solve various problems with OBDDs; these models, however, have varying degrees of usefulness in regards to model-checking, because some of these models do not maintain the key property of canonicity. Bryant [9] provides a list of some of the major variants, some of which are summarized below:

1.5.1 General BDDs

Satisfiability using the most generalized form, namely BDDs, has been examined by Ashar Devadas and Ghosh [3]. In such a case, guaranteed canonicity is lost, and some operations that were polynomial in time now become exponential, so the complete removal of ordering requirements loses the key properties of OBDDs that make them valuable for model checking, and are therefore not generally considered suitable for use without some restrictions.

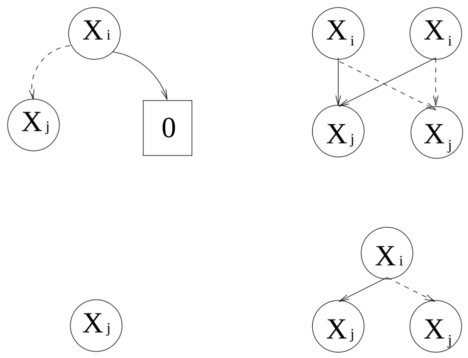

Figure 2: Left and right: two different rules for reduction

1.5.2 Zero-suppressed BDDs

Though intended for set representation, *Zero-Suppressed BDDs* (introduced by Minato [25]) could also serve in verification-centric areas in certain circumstances. Instead of representing functions with BDDs, one can also represent sets as follows: instead of serving as inputs, $x_1 \ldots x_n$ would represent inclusion (1) and exclusion (0) in a set; all sequences whose terminals end in a 1 are considered valid members of a set.

For zero-suppressed BDDs (*0-Sup-BDDs*), the reduction from a BDD follows a different set of rules. Instead of removing nodes whose 0 and 1 edges point to the same node, nodes whose 1 edge point to the 0 terminal are removed. The resulting reduction allows for the removal of nodes for which the given x_i is not present within the set. Thus, 0-Sup-BDDs are meant to represent sparse sets, but can also represent functions where very few inputs need to be true in order to satisfy it. Figure 2 shows the two reduction possible for 0-Sup-BDDs.

Most importantly, however, is that this variant on OBDDs still possesses efficient algorithms and maintains canonicity, two properties that allow it to be useful for automated verification. Due to being designed to handle sparse data, however, the domains where such a representation allows for a much smaller graph is limited, but experimental data within this paper (and within a follow-up paper by Minato [26] that focussed more on verification) demonstrates the usefulness of 0-Sup-BDDs. The total reduction in nodes possible by 0-Sup-BDDs has been proven by Schrer and Wegener [34] to be a factor of n for a n bit vector, thus offering a significant reduction to memory usage that often runs short in automated verification.

1.5.3 Shared BDDs

So far, the OBDDs discussed can only handle one function at once. However, most applications involving OBDDs are ones that ultimately involve more than one function being turned into an OBDD. When being used on real systems, all the extra space needed for multiple OBDDs can make a problem too complex for systems.

In order to counter this problem, Minato, Ishiura and Yajima [27] proposed a solution where multiple functions share the same nodes on one OBDD. This

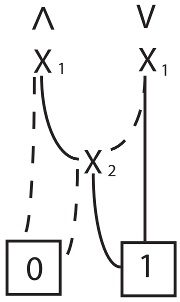

Figure 3: A Shared Binary Decision Diagram: the left x_1 marked with a $\wedge$ is the beginning node for $x_1 \wedge x_2$ and the right node marked with a $\vee$ is for the formula $x_1 \vee x_2$

new type of OBDD is called a *Shared Binary Decision Diagram* (SBDD). In terms of creating SBDDs, the process is not much more different than creating an OBDD; the BDD construction algorithm creates nodes where needed (reusing nodes with identical pointers to the same keys) and then performs the reduction as normal. What results is a single OBDD with multiple input sets, and depending on which inputs are used, different functions are represented.

There are several advantages to this approach; for instance, instead of checking for equivalence of functions using isomorphism, one must only check to see whether the inputs to both functions are pointing to the same nodes. If the pointers are the same, then the two functions are equivalent. More importantly, this allows for a compact way to represent multiple functions by letting them share the parts of the OBDD that are isomorphic to one another.

In order to perform comparisons and ensure the minimal size BDD, the construction algorithm must guarantee that no isomorphic subgraphs exist within the SBDD. Fortunately, by maintaining a hash table of all the nodes created so far, it can be guaranteed that no isomorphic subgraphs exist.

Finally, by putting all of the OBDDs into the same graph, one can not immediately just delete an OBDD when it isn't needed anymore. Instead, one must differentiate between the nodes that are still being used and nodes that are no longer useful. This can be done by keeping track of how many functions use a particular node, decrementing the value when a function is removed until it reaches zero and then removing it. In terms of implementation, this final part is done by keeping the nodes (since they may become useful later and removing is computationally expensive) and waiting until the system is running low on memory, only then performing garbage collection.

Also introduced in this paper is *attributed edges*, that is assigning extra functions to certain edges in the SBDD (like inversion of the value) in order to represent the SBDD more compactly. Provided some restrictions are met, this often works to shrink a graph further.

There is one drawback to the SBDD - namely, all of the functions in the same SBDD must have the same variable ordering. This restriction can be problematic as the number of functions increases, as the likelihood of finding an ordering that makes all OBDDs non-exponential in size decreases. However, when performing model checking, the OBDDs should be mostly related to one another in some way, hopefully allowing for a relatively efficient ordering.

For model checking purposes, SBDDs can be an excellent way to manage several functions using the same memory structure. In fact, a well-known implementation by Brace, Rudell and Bryant [7] uses this global OBDD structure to manage all the separate functions.

1.5.4 BDDs for non-boolean functions

At some point, the idea surfaced to try out OBDDs for non-boolean functions; that is, whether or not desirable properties can be obtained for functions that return numeric values.

Multi-Terminal BDDs (MTBDDs) [13], similar to Algebraic Decision Diagrams (ADDs) by Bahar et al. [4], attempt to solve this problem by allowing for values besides 0 and 1 as terminals. The main problem comes from the possibly exponential number of terminal nodes in order to represent a range of numbers; therefore, except for limited ranges of values, these approaches suffer from state explosion in most cases and as of right now are only useful in limited situations.

2 BDDs and automated verification

Given the above description of BDDs, it should be evident that people within the automated verification community held a great deal of interest in these structures. In this section, some of the main goals of automated verification will be discussed and how BDDs in particular aid in this work.

2.1 Equivalence of boolean expressions

A simple question in verification may be "does circuit x produce the same results as formula y". Given the exponential (in terms of the number of terms/inputs) number of equalities to test by manually checking, BDDs allow for a potentially simpler solution.

Recall the canonical nature of BDDs; that is, the reduction of a boolean formula to a OBDD is unique for a given variable ordering. Now, equivalence can be tested by turning both formulas/circuits into OBDDS and checking if the resultant OBDDS are identical or not. If they are, the two circuits/functions/formulas are equivalent. This simple property results in an elegant way to verify circuits, and leads to further notions in automated verification as detailed below.

2.2 Simulating finite-state machines

Though equivalence in circuits is a start, an extension is needed in order to verify equivalence in finite-state machines; that is, does the circuit implementation match the specification. The answer came from Touati et al. [38]

In order to compare two sequential circuits they need to have their outputs compared through all reachable states. In order to do this, BDDs must be used in a different way; instead of representing the circuit itself, the BDD will be used to represent a set of truth assignments that make the circuit true; every 1-terminal node means that the given input/latch values make the assignment true.

Next, an *image* is needed. An image represents the set of states that a BDD can be in given a previous state. This can be done using BDDs by encoding the *trasition relation* for each input; that is, defining each variable in terms of all the previous states that make it true. In theory, an OBDD allows for a compact representation of all the inputs that make the input true in the next state.

Finally, all the images need to be combined and run in an iterative fashion until no more states can be found. Let S_0 be the base (reset) state; for the first state S_1 we use $S_1 = S_0 \vee Image(S_0)$, that is the set of all reachable states includes both S_0 and the $Image$ of S_0. This continues up to state $S_i = S_{I-1} \vee Image(S_{I-1})$ indicating all possible states up to state I. Eventually, this system will converge; testing for this is easy (provided the BDDs are not exponential in size) by doing a simple comparison of S_I and S_{I-1}. If they are the same, then no more states are reachable.

As an implementation note, this all is relatively easy using reasonably-bounded BDDs; each iteration of this algorithm takes $O(|G|)$ time (using the *Apply* function), turning all the transition relations into OBDDs takes linear time as well, and finding the set of satisfiable inputs for each iteration takes $O(n \bullet |SAT_f|)$ where $|SAT_f|$ is the number of satisfying assignments. Therefore the verification remains in polynomial time provided that the OBDDs remain polynomial in size.

2.3 Model checking

The next step to automated verification came with formal model checking (originating from Clarke et al. [12]) where instead of verifying sequential circuits, properties expressed in a temporal logic (such as CTL) are verified. Rather than just checking equivalence of circuits, now more detailed properties (like checking for whether a given state can always be reached) can also be tested.

The major addition over finite state machines is the need to handle the next state as well as more complicated aspects such as the until command in CTL. For the Boolean operators $\vee, \wedge, \neg$ the *Apply* function works well enough. EX and AX require the image of the current state to work (depending on the method, the image or the pre-image). Finally, EU and EG require combine the previously discussed elements and employ a tactic similar to finite state machine verification; namely, they continue to check until state S_I is equal to state S_{I+1}, thus ensuring all reachable states have been checked.

Through the same basic operations that Bryant discovered, one can use OBDDs in model checking to verify complex properties. The power of OBDDs goes a long way in explaining why so much work is put into a problem that is known to be worst-case exponential in size. As a result, much work goes into writing heuristic algorithms for better input orderings. They will be discussed in the next section.

3 Approximating optimal OBDDS

Unlike most notions of approximation for NP-hard problems, it is important to note that the notion of approximating an OBDD does not imply that the OBDD that is produced is in any way incorrect. Instead, this approximation is based on the size of the OBDD (i.e. the number of nodes) and whether or not the variable ordering provided results in the minimal size OBDD. In this section, work relating to the minimizing (and approximation thereof) will be examined.

3.1 Initial observations

What should be immediately obvious is that polynomially-sized BDDs allow for satisfiability to be determined in polynomial time; since a OBDD can be constructed in polynomial time [8], and satisfiability has long been established as an NP-hard problem (introduced by Cook [14]), if such an OBDD can be found in polynomial time for all boolean functions, then $P = NP$.

In terms of specific NP-hard or NP-complete problems involving BDDs, NP-completeness for the decision variant of BDD minimization (that is, whether $\exists k$ such that an ordering of the variables produces an OBDD with size$\leq k$) called MinOBDD was first proven for SBDDs (discussed in Section 1.5.3) by Tani, Hamaguchi and Yajima [37] in 1993. In 1996, Bollig and Wegener [6] proved that MinOBDD is in fact NP-complete.

3.2 Heuristic methods

Initially, variable ordering were chosen by hand; trial-and-error would eventually produce a reasonable ordering in most cases. Later, heuristic methods were used to automatically order according to past research and experience.

There are various styles of heuristic ordering methods, probably enough to write a separate paper on. Among logic-driven work (that is when using OBDDs to model already existing circuits) a depth-first traversal through the circuit itself from the outputs to the inputs is the most common technique; putting the inputs that affect the lowest levels of the circuit are put at the top of the order, since these often have the greatest effect on the other variables and are best evaluated at the beginning. Work by Minato et al. [27] and by Fujita et al. [16] is typical of this approach. These heuristics and others can offer performance better than random assignment in most cases.

The fundamental problem with using a heuristic method at the beginning is that most OBDD packages use SBDDs; therefore, adding in more functions later can significantly change the number of nodes from manageable to exponential. Plus, having to input all functions in at the beginning means that one cannot take advantage of garbage collection midway through to possibly make smaller SBDDs. Therefore, it is beneficial to have methods that allow for variables to be re-ordered in mid computation, as described in the next section.

3.3 Dynamic variable re-ordering techniques

Instead of deciding on the order and being forced to stay with it, the concept of *dynamic variable re-ordering* emerged.

As an aside, however, it is incorrect to assume that these dynamic variable reordering techniques are not heuristic algorithms in their own right; none of these re-ordering algorithms can be exact without taking non-deterministic polynomial time (as discussed in Section 3.1) but the literature has left the term heuristic algorithms to the initial ordering of variables.

The term itself was introduced by Rudell [33], but he cites similar work by Fujita et al. [17] and Ishiura et al. [21] as alternative forms of dynamic variable re-ordering. In these papers, work was done to look at a group of variables in the ordering and to determine which pair needed to be exchanged in order to make the OBDD smaller. Rudell [33] referred to this as a *window algorithm* because the algorithm selected a "window" to choose among it the pair that should be exchanged.

In contrast Rudell [33] introduced the concept of a *sifting algorithm* to continually reduce the size of OBDDs produced by an implementation provided by Brace, Rudell and Bryant [7]. In this implementation, Rudell left the re-ordering as a process do be decided entirely by the computer, making it transparent to the user. In essence, the reduction operation was performed when the system was getting too large; i.e. at the same time as garbage collection was being performed on the SBDD for nodes that were no longer being used.

This algorithm works on the observation of Fujita et al. [17] that swaps of adjacent variables only affects the number of nodes on those two layers, thus allowing for a relatively efficient swap procedure for adjacent variables to be performed. For the algorithm, a brute-force approach was taken; each variable was taken and exchanged with its lower neighbours down to the bottom level. Then, it was moved back up to the top. Finally it was moved back to the level that resulted in the smallest number of nodes. Though brute force in nature (taking $O(n^2)$ swaps for the n variables) Rudell showed that it could often help in managing the size of SBDDs.

It should be noted, however, that not all re-ordering techniques seem to help according to Harlow and Brglez [19]; for the ALU4r (a series of 100 4-bit isomorphic ALU circuits), statistical tests showed that improper uses of the sifting algorithm performed equivalently to a random variable order at a 95 percent confidence interval. Thus, it is key to the operation of these re-ordering techniques that they be applied sufficiently and appropriately.

In conclusion, none of the above methods claim to offer optimality, but all are considered to aid in reaching a smaller OBDD. The next part, however, discusses a proof that no heuristic or re-ordering algorithm can possibly guarantee closeness to an optimal solution within any constant factor.

3.4 Proof of non-approximability

In 1998, Sieling [35] constructed a proof of non-approximability for MinOBDD, the decision variant problem for variable ordering in a OBDD. The theorem states that there does not exist a polynomial time approximation algorithm with a constant performance ratio (that is, the optimal to approximate graph sizes for a given ordering) unless $P = NP$.

In order to prove non-approximability, Sieling used an *L-reduction* from another known NP-complete problem. Like reductions to prove NP-completeness, an L-reduction (introduced by Papadimitriou and Yannakakis [30]) is a polynomial time reduction where one problem can be translated to another such that the approximation ratio of the translated problem is the same within a given constant factor β.

In this case Sieling used a L-reduction from known NP hard problem Max-Cut (proved to be non-approximable by Hastad [20]), which attempts to find a division of a graph into two subsets such that the number of edge crossings is maximized.

For the transformation, Hastad created a general class of graphs and then showed that he could encode this graph in a SBDD in such a way that the size is directly proportional to the MaxCut of the graph. Thus, if a polynomial time algorithm exists to approximate this size within a constant factor, it could also approximate MaxCut in polynomial time, which has been proven to be impossible unless $P = NP$. Therefore, Hastad was able to prove that MinSBDD is an NP-complete problem, followed by a similar work proving the same for OBDDs, and a restricted class of OBDDs called OBDD* (which basically had a promise that the given OBDD with N vertices has an optimal solution with at least $\frac{N}{2}$ nodes in it). Sieling also showed that the use of edge-attributed output inverters do not allow for an approximation either, since such edge-attributed items had been useful in shrinking the size of a SBDD.

In a way, the result itself is not surprising, considering how closely OBDDs are tied to satisfiability. That, and heuristic methods still offer some hope, but this result has placed a significant obstacle in front of researchers who wish to use OBDDs on larger problems and were hoping for provable approximations to achieve this.

4 Possible future work

Given the non-approximability result for variable ordering (not to mention the existence of functions known to produce exponential-sized OBDDs regardless of ordering) it might seem that approximation algorithms have little or no place in OBDD work, and that heuristic methods (combined with more powerful computers) is the best approach.

This is not the only way one can approximate a BDD, however. A different way to approach approximation for OBDDs would involve changing the focus on what is being approximated. So far, the main thrust of research has been centred around an approximation of the variable ordering; that is, approximating the NP-hard problem of finding the best variable ordering.

Instead of approximating the variable ordering, an alternate area of more theoretical research would involve using BDDs to create an approximation of the boolean function itself; that is, try and find approximation bounds for the satisfiability of a given system. For instance, instead of satisfiability one would aim for MAX SAT, where the object is to satisfy as many clauses in a formula as possible with a given variable assignment. Though this may not be of significant use to model checkers, because the results will not be definite, there may be some interest in probabilistic methods for problems that BDD-driven model checkers cannot solve.

In some ways, this can already be done by approximating the system being modelled; thus, some of the inputs being modelled are no longer present in the reduced system, thus making for a smaller BDD. Ravi and Somenzi [32] (followed up later by Ravi et al. [31]) gave some algorithms for under-approximating systems to improve reachability for difficult classes of problems. The approach hypothesized above, however, deals with the larger model, and tries to extract as many truthful clauses out of it as possible. Such a result may be useful later to decompose the formula further, or to decide where to abstract in order to get the most interesting results.

Consider a formula with c clauses; instead of making one large OBDD, an approximation algorithm would construct a BDD for each clause. Using the concept of priority algorithms (introduced by Borodin, Nielsen and Rackoff [29]) an ordering can be performed on the clauses. With such an ordering, the highest priority clause would be analysed and the satisfying assignments would be stored; the algorithm would continue by using the *Restrict* function with subsequent clauses in order to try and find a polytime approximation that uses BDDs to find MAX-SAT.

With proper bookkeeping, such a method might be a powerful way of solving MAX SAT, since each of the clauses themselves will not grow sufficiently large (even if exponential in size) to be unmanageable in size by a modern computer.

5 Conclusion

This paper represents a small survey across the field of OBDDs as they apply to model-checking. Granted, one could write a summary paper on many of these topics alone (given the volume of alternate forms of BDDs and heuristic algorithms) but these provide an interesting sampling. It is interesting to note how well this field has mixed various streams of computer science; BDDs started off with engineers looking to validate circuits, became a focus for those in data structures and algorithms, migrated over to empirical testing and then ended up in the hands of theoreticians with proofs of NP-completeness and non-approximability.

Even though the NP completeness and non-approximibility results are "bad news" for researchers, there still exists a strong field willing to work on improving the bounds, if only because of the usefulness of OBDDs in model-checking; one can only assume the benefits of using OBDDs will continue to outweigh the worst-case running times.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[3] P. Ashar, S. Devadas, and A. Ghosh. Boolean satisfiability and equivalence checking using general binary decision diagrams. In *Proceedings of the International Conference on Computer Design*, pages 259–264. IEEE, 1991.

[4] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD*, pages 188–191, 1993.

[5] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[6] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45:993–1002, 1996.

[7] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE, 1990.

[8] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] R. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose/CA*, pages 236–243. IEEE CS Press, Los Alamitos, 1995.

[10] G. Cabodi, P. Camurati, and S. Quer. Can BDDs compete with SAT solvers on bounded model checking? In *Proceedings of the Design Automation Conference (DAC)*, 2002.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.

[12] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Lang. Syst.*, 8:244–263, 1986.

[13] E. Clarke, K. McMillan, X. Zhao, and M. Fujita. Spectral transforms for large Boolean functions with application to technology mapping. In *Proceedings of the 30th Design Automation Conference*, pages 54–60. IEEE, 1993.

[14] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158. ACM, New York, 1971.

[15] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking in an industrial setting. In *Computer-Aided Verification (CAV)*, 2001.

[16] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 2–5, 1988.

[17] M. Fujita, Y. Fujisawa, and Y.Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the Design Automation Conference*, pages 50–54, 1991.

[18] J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. In *Ershov Memorial Conference*, 2001.

[19] J. Harlow and F. Brglez. Design of experiments in BDD variable ordering: lessons learned. In *Proceedings of the International Conference on Computer-Aided Design*, pages 646–652, 1998.

[20] J. Hastad. Some optimal inapproximability results. In *Proc. of the 29th Symposium on Theory of Computing*, pages 1–10. STOC, 1997.

[21] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the International Conference on Computer-Aided Design*, pages 472–475, 1991.

[22] J. Jain, K. Mohanram, D. Moundanos, I. Wegener, and Y. Lu. Analysis of composition complexity and how to obtain smaller canonical graphs. In *Proceedings of the Design Automation Conference*, pages 681–686, 2000.

[23] M. Langberg, A. Pnueli, and Y. Rodeh. The ROBDD size of simple CNF formulas. In *CHARME*, 2003.

[24] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[25] S. Minato. Zero-supressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEEDesign Automation Conference*, pages 272–277, 1993.

[26] S. Minato. *Binary decision diagrams and applications for VLSI CAD*. Kluwer, 1995.

[27] S. Minato, N. Ishiura, and S. .Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th ACM/IEEEDesign Automation Conference*, pages 52–57, 1990.

[28] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[29] A. Borodin M. Nielsen and C. Rackoff. Incremental priority algorithms. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 752–761. ACM, 2002.

[30] C. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[31] K. Ravi, K. McMillan, T. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Design Automation Conference*, pages 445–450, 1998.

[32] K. Ravi and F. Somenzi. High density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, 1995.

[33] R. Rudell. Dynamic variable reordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47, 1993.

[34] O. Schrer and I. Wegener. The theory of zero-suppressed bdds and the number of knight's tours.

[35] D. Sieling. The nonapproximability of OBDD minimization, 1998.

[36] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48:139–144, 1993.

[37] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problem of shared binary decision diagrams. In *Proceedings of the 4th International Symposium on Algorithms and Computation ISAAC*, pages 389–398, 1993.

[38] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *International Conference on Computer-Aided Design*, pages 130–133, 1990.