Using Decision Diagrams to Compactly Represent the State Space for Explicit Model Checking

Hao Zheng, Andrew Price, and Chris Myers

Abstract—The enormous number of states reachable during explicit model checking is the main bottleneck for scalability. This paper presents approaches of using decision diagrams to represent very large state space compactly and efficiently. This is possible for asynchronous systems as two system states connected by a transition often share many same local portions. Using decision diagrams can significantly reduce memory demand by not using memory to store the redundant information among different states. This paper considers multi-value decision diagrams for this purpose. Additionally, a technique to reduce the runtime overhead of using these diagrams is also described. Experimental results and comparison with the state compression method as implemented in the model checker SPIN show that the approaches presented in this paper are memory efficient for storing large state space with acceptable runtime overhead.

Index Terms—formal verification, model checking, decision diagrams, state compression

I. INTRODUCTION

Model checking [7] is an automated formal analysis method for verifying hardware and software systems. It systematically checks whether a model of a given system satisfies a desired property such as deadlock freedom and request-response properties [5]. However, model checking is computationally very expensive as it searches the entire reachable state space of a design for errors. Typically, all reachable states found during reachability analysis need to be stored to avoid searching the same part of state space multiple times unnecessarily. As the number of states grows exponentially in the size of designs under verification, physical memory installed on typical computers can be exhausted quickly, therefore limiting model checking to the designs of small sizes. This problem is well known as state explosion.

In asynchronous designs, multiple components execute concurrently. When verifying asynchronous designs, generally interleaving semantics are used to represent the behavior of such designs. More specifically, when multiple components are ready to execute in a state, only one component is selected. This interleaved execution makes sure all possible orderings of concurrently enabled executions are considered, thus capturing all possible behavior for verification. The need to consider all possible interleavings of concurrent executions is the main cause of state explosion in asynchronous design verification as the number of interleavings grows exponentially if a design has a high degree of concurrency, and this leads to an excessively large state space for even a relatively small system.

In the interleaved execution as described above, only one component executes and updates the relative portion of a state, and the remaining portions of the state are unchanged. Therefore, storing the entire states results in unnecessary overhead. In model checker SPIN [10], a state compression method is described where the independent parts of the design states are considered when representing states. Independent parts refer to variables including global and local variables belonging to different processes. An unique copy is created for each different independent part, and the references to the copies of independent parts are used to construct the states.

While the above compression method reduces memory significantly in representing states, there are still a lot of redundancies in the state representations. As pointed out in the previous paragraphs, a global state of a design is in fact a tuple of pointers to the local states. Two states connected by a state transition may still share a large number of same pointers in their representations as a state transition typically causes only a few local changes. Therefore, reducing the redundancy in the state representations further can improve the scalability even more. This paper proposes to use the multi-value decision diagrams to represent the reachable states for explicit model checking of asynchronous systems. The multi-value decision diagrams, similar to the well-known binary decision diagrams (BDD) [3], uses directed acyclic graphs to sets of objects encoded by typed variables. The compactness of the decision diagram representations is due to sharing of common portions of a large number of different objects.

BDDs are widely used in symbolic model checking [4] where the model checking algorithms and the state representations are both based on the Boolean operations. However, in the explicit model checking, individual states are created and manipulated. Adding individual states into a decision diagram representation can incur high runtime overhead, and this would cancel largely the benefit of compact memory footprint of the decision diagrams. To address this problem, this paper also proposes to use multi-value decision trees along with the decision diagrams. Adding states into a decision tree is much faster, but it requires more memory than a decision diagram. In this method, a decision tree is used as a buffer where it stores the reachable states until some threshold is exceeded. Then, the decision tree is compressed, and merged with the decision diagram representing the whole reachable state space. This approach can significantly reduce the runtime overhead.
associated with using the decision diagrams, even though it may increase the average memory required compared to using the decision diagrams solely.

In this paper, the multi-value decision diagrams are only used to store the states found during the depth-first search that are generally implemented for the explicit model checking of asynchronous systems. The algorithms themselves remain the same. Therefore, the previous results to improve scalability such as partial order reduction [8], [9] can still be used.

II. BACKGROUND

A. Labeled Petri-Nets

This paper uses Labeled Petri-Nets to model asynchronous systems. Petri-Nets are a common modeling formalism for asynchronous designs [12], [1]. A Petri-net is a directed graph with a set of transitions and a set of places. A labeled Petri-Net is a Petri-net where transitions are labeled with various information representing a system’s properties and behavior [15]. Its definition is given as follows.

Definition 2.1: A labeled petri-net (LPN) is a tuple $N = \langle V, P, T, F, M_0, \text{init}, L \rangle$, where
1) $V$ is a set of state variables of the integer type,
2) $P$ is a finite set of places,
3) $T$ is a finite set of transitions,
4) $F \subseteq (P \times T) \cup (T \times P)$ is a finite set of the flow relations,
5) $\mu_0 \subseteq P$ is a finite set of initially marked places,
6) $\alpha_0 : V \rightarrow \mathbb{Z}$ is a labeling function that assigns each variable an initial value,
7) $L = \langle \text{Guard}, \text{Assign} \rangle$ is a pair of labeling functions for transitions in $T$, which is defined below.

A simple LPN example is shown in Fig 1. Fig 1(a) shows a simple asynchronous circuit consisting of three components, and Fig. 1(b) shows the LPNs for each component in the circuit. For each component, its LPN has 2 places and 4 transitions. The places are represented as circles, and the transitions are represented as boxes. Each place is preceded and followed by one or more transitions, and each transition is preceded and followed by one or more places. The flow relations are represented by the edges connecting the transitions and places [1]. The bullets found in some places are called tokens. Each place can have at most one token. A place is marked if it has a token. A marking of LPN, $\mu \subseteq P$ is a set of marked places.

The dynamic behavior of a concurrent system is captured by LPN transitions with labelings. Each transition $t \in T$ has a preset denoted by $\bullet t = \{ p \in P | (p, t) \in F \}$, which is the set of places connected to $t$, and a postset denoted by $t \bullet = \{ p \in P | (t, p) \in F \}$, which is the set of places to which $t$ is connected. The preset and postset for places are defined similarly.

Before defining the transition labels formally, the grammar used by these labels is introduced first below [15]. The numerical portion of the grammar is defined as follows:

$$\chi ::= c_i \mid v_i \mid (\chi) \mid \chi^+ \mid \chi^- \mid \chi \times \chi \mid \chi / \chi \mid \chi \% \chi \mid \text{NOT}(\chi) \mid \text{OR}(\chi, \chi) \mid \text{AND}(\chi, \chi) \mid \text{XOR}(\chi, \chi) \mid \text{INT}(\phi)$$

where $c_i$ is an integer constant from $\mathbb{Z}$, and $v_i$ is an integer variable. The functions NOT, OR, AND, and XOR are bit-wise logical operations assuming a 2's complement format with arbitrary precision. INT($\phi$) returns 1 if the Boolean expression $\phi$, which is defined below, evaluates to true, or 0 otherwise. The set $P_{\chi}$ is defined to be all formulas that can be constructed from the $\chi$ grammar.

The Boolean portion of the grammar is as follows:

$$\phi ::= \text{true} \mid \text{false} \mid v_i \mid \neg \phi \mid \phi \land \phi \mid \phi \lor \phi \mid \chi \equiv \chi \mid \chi \geq \chi \mid \chi > \chi \mid \chi \leq \chi \mid \chi < \chi$$

where the integer $v_i$ is regarded as true if its value is nonzero, and false otherwise. In this sense, it is similar to the semantics of the C language. The set $P_{\phi}$ is defined to be all formulas that can be constructed from the $\phi$ grammar.

As in Definition 2.1, each LPN transition is labeled with an enabling condition and a set of variable assignments. LPN transition labeling is defined by $L = \langle \text{Guard}, \text{Assign} \rangle$ where

- Guard : $T \rightarrow P_{\phi}$ labels each LPN transition with a Boolean expression that defines its enabling condition.
- Assign : $T \times V \rightarrow P_{\chi}$ labels each LPN transition $t \in T$ and each variable $v \in V$ with an integer assignment made to $v$ when $t$ fires.

For a large complex design, it usually consists of multiple components. Then, each component is modeled in a LPN module, and the LPN for the whole design is the parallel composition of the LPN modules. Let $N_1, \ldots, N_n$ be the LPN modules for the components consisting of a design. The LPN for the design is $N = N_1 \parallel \ldots \parallel N_n$ where $\parallel$ is the parallel composition operator for LPNs. In our method, communications among components are represented by the shared variables. Shared variables between components $N_i$ and $N_j$ are variables in $V_i \cap V_j$. Moreover, no two LPN modules share any common places. In other words, $\forall i,j \neq j \Rightarrow P_i \cap P_j = \emptyset$. In Fig. 1, the design is partitioned into three components for illustration purpose. These components are represented by LPN modules as shown in $M_1$, $M_2$, and $M_3$.

B. Reachability Analysis

A basic approach for analyzing the dynamic behavior of a concurrent system modeled with LPNs is reachability analysis, which finds all possible state transitions and thus reachable states for such a system. The reachable state space is typically represented by a state graph. A state graph is a directed graph where vertices represent states and edges represent state transitions.

A state of a LPN module $N_i$ is a pair $(\mu_i, \sigma_i)$ where $\mu_i$ denotes a marking of $N_i$ and $\sigma_i$ denotes a vector of values over variables $V_i$ of $N_i$. The initial state init of $N_i$ is $(\mu_0, \alpha_0)$. Given a state $s_i$ of module $N_i$, $M(s_i)$ denotes the marking of $s_i$ and $\sigma(s_i)$ denotes the state vector of $s_i$. Also, for any expression $e \in P_{\chi} \cup P_{\phi}$, value($e, s_i$) denotes a function that returns the value of expression $e$ in state $s_i$. As described in the previous section, the LPN of a design is the parallel composition of a set of LPN modules, each of which represents a component in the design. Let $N = N_1 \parallel \ldots \parallel N_n$ be the LPN
of a design where $N_i$ ($1 \leq i \leq n$) is the LPN module for the $i$th component of the design. A global state $\vec{s}$ of $N$ is a $n$-tuple $(s_1, \ldots, s_n)$ where each $s_i$ ($1 \leq i \leq n$) is a local state of LPN module $N_i$.

Before describing the reachability analysis, the enabling condition of the LPN transitions is defined below.

**Definition 2.2:** Let $N_i$ be a LPN module. A LPN transition $t$ is enabled in a state $\vec{s}$ if the following two conditions are met:

1) $t \subseteq \mu(s)$,
2) $\text{value}(e, s)$ is true or not zero for $e = \text{Guard}(t)$.

In Fig 1, every transition has its preset included in the initial marking. In the initial state, the values of variable $u$ and $z$ are 0, $\text{Guard}(t_{11})$, which is $u = 0 \land z = 0$, is evaluated to be true, therefore transition $t_{11}$ is enabled in the initial state.

Given a LPN module, the set of transitions enabled in a state $\vec{s}$ is denoted by $\text{enabled}(\vec{s})$. Naturally, the enabled transitions of the whole design LPN are denoted by $\text{enabled}(\vec{s})$.

The reachable state space of a LPN model can be found by exhaustively firing every enabled transition starting at the initial state. Firing a transition may lead to a new state by exhaustively firing every enabled transition starting at the initial state. The reachable state space of a LPN model is given in Algorithm 1. Reachable states are stored in stateTable. During the reachability analysis, after a LPN transition $t_i$ is fired in global state $\vec{s}$, the local state of $Mi$ is changed to a new one. In other words, if transition $t_i$ is fired in state $\vec{s} = (s_1, \ldots, s_i, \ldots, s_n)$, a new global state $\vec{s}'$ is generated such that $\vec{s}' = (s'_1, \ldots, s'_i, \ldots, s'_n)$. Observe that in $\vec{s}$ and $\vec{s}'$, many local states might be exactly the same. To avoid creating fresh copies of the same local states for $\vec{s}'$ in memory, the actual definitions of the distinct local states are stored in a hash table for each LPN module, and the pointers to the local states are used to construct the global states. From now on, $\vec{s} = (s_1, \ldots, s_n)$ is viewed as a tuple of pointers to the local states $s_1, \ldots, s_n$. In this way, a local state is created only once, and its pointer may be referenced many times in different global states. The similar state representation is also implemented in the model checker SPIN [10] and Java PathFinder [2].

In many existing model checkers, the obvious choice of the data structure for stateTable is hash tables. Even though inserting and accessing hash tables are usually very efficient, the memory usage increases exponentially as the number of states found during the reachability analysis increases exponentially in the size of a design. Therefore, representing and storing reachable states compactly is extremely important. To address this problem, the following sections describe how the reachable states are represented using graphs, specifically decision trees and diagrams which may allow very large number of states to be stored in a relatively small memory footprint.

---

**Algorithm 1: search($((T, P, F, M_0))$)**

1. stateStack.push($\vec{s}_0$);
2. enabledStack.push($\text{enabled}(\vec{s}_0)$);
3. while stack is not empty do
   4. $\vec{s} = \text{stateStack.top}()$;
   5. $E_\vec{s} = \text{enabledStack.top}()$;
   6. if $E_\vec{s} = \emptyset$ then
      7. stateStack.pop());
      8. enabledStack.pop());
      9. continue;
   10. Select $t \in E_\vec{s}$ to fire;
   11. $E_\vec{s} = E_\vec{s} \setminus t$;
   12. $\vec{s}' = t(\vec{s})$;
   13. if stateTable.contains($\vec{s}$) == true then
      14. stateStack.push($\vec{s}'$);
      15. enabledStack.push($\text{enabled}(\vec{s}')$);
      16. stateTable.add($\vec{s}'$);

---
III. STATE SPACE REPRESENTATIONS

As described above, different global states might have a large number of pointers to the same local states. To save memory even further by avoiding storing the same local state pointers in different global states, a data structure called multi-value decision diagram (MDD) is implemented in our method to store global states found during the reachability analysis. The concept of MDD [17] is very close to that of binary decision diagrams (BDDs) [3] in that both use graphs to represent a set of objects encoded by some variables. The memory efficiency comes from the sharing of the same variables used to encode different objects. In BDDs, the variables used for the encoding are binary variables, while integer variables are used for the encoding in the case of MDDs. In this sense, BDDs can be regarded as a special case of MDDs. The structural representation of the global states in our method is natural for MDDs. To facilitate the presentation, an unique integer index is assigned to each local state of a single LPN module. Therefore, a global state $s = (s_1,\ldots,s_n)$ can be regarded as a tuple of integers where $s_i$ is the index to a local state of module $i$.

A. Multi-Value Decision Trees

This section describes multi-value decision trees to introduce multi-value decision diagrams that are described in the next section. A multi-value decision tree is a directed acyclic graph with a single root node, terminal nodes and non-terminal nodes. The root node does not have incoming edges, and the terminal nodes do not have any outgoing edges. Each non-terminal node has a single incoming edge and one or more outgoing edges. Each edge is labeled with an integer number. A path is a sequence of edges from the root to one of the terminal nodes, therefore an integer tuple can be formed by collecting the integers labeled on the edges on the path. In other words, a path represents or encodes an integer tuple. An example of multi-value decision tree is shown in Fig. 2. Note that a terminal node is drawn for each path in the figure to make presentation clear. In the actual implementation, only a single terminal node is used, and all paths converge to this unique terminal node.

In our implementation, two operations are supported for multi-value decision tree, \texttt{add} and \texttt{contains}. Function \texttt{add} takes as input an integer tuple, and creates necessary edges in the decision tree to build a path corresponding to the input tuple. For example shown in Fig. 2, if a new tuple $(1,1,3)$ is given, it can be added to the decision tree by creating an edge from node $n_{21}$ to the terminal node, and this edge is labeled with 3. If another tuple $(1,1,1)$ is given, then no new edge is created as there is a path existing in the decision tree corresponding to the given tuple. Function \texttt{contains} also takes as input an integer tuple, and checks if there is a path in the decision tree that corresponds to the given tuple. It returns true if there is, false otherwise. The time complexity of both functions is linear in the size of the tuples.

The memory efficiency of the decision trees depends on the similarity of the prefixes of the tuples in the given set. If there are a large number of tuples with long same prefixes, a lot of node sharing can reduce the memory usage significantly. On the other hand, the worst case happens when a large number of tuples differ on their first elements. In this case, more memory may be required compared to using hash tables as an edge is needed for each element in each tuple.

B. Multi-Value Decision Diagrams

A multi-value decision diagram, similar to the decision trees, is a rooted directed acyclic graph where there is an unique root node, an unique terminal node, and non-terminal nodes. In a multi-value decision diagram, the nodes are partitioned into levels. The number of levels of a decision diagram is equal to the size of the integer tuples representing the global states. The root node is at level 0, which is at the top of the decision diagram. The node at the bottom is the terminal. Each non-terminal node has a number of outgoing edges connecting nodes at one level higher or the terminal, and a number of incoming edges from nodes including the root node from one level lower. Non-terminal nodes in the decision diagrams, unlike those in the decision trees, can have multiple incoming and outgoing edges. A path from the root node to the terminal node in a decision diagram corresponds an integer tuple. In the reachability analysis, the decision diagrams are used to replace the hash tables, therefore our implementation only supports two operations, \texttt{union} and \texttt{contains}. Since \texttt{contains} for the decision diagrams is the same as function \texttt{contains} for the decision trees, \texttt{union} is described in this section.

Function \texttt{union} takes as inputs two decision diagrams, and returns a decision diagram that includes all the paths from either of the two input decision diagrams. During the \texttt{union} operation, an unique table is maintained to make sure that no equivalent nodes are created. Before the node equivalence is defined, let $(n,i,n')$ be an outgoing edge of node $n$, and the set of all outgoing edges of node $n$ is denoted as \texttt{outgoing}(n). Two nodes $n_1$ and $n_2$ are \texttt{equivalent}, denoted as $n_1 \equiv n_2$, if $n_1$ and $n_2$ are the same, or the following conditions are satisfied.

- $\forall(n_{1},i,n'_{1}) \exists(n_{2},i,n'_{2}) \ n'_{1} \equiv n'_{2}$,
- $\forall(n_{2},j,n'_{2}) \exists(n_{1},j,n'_{1}) \ n'_{1} \equiv n'_{2}$.

Function \texttt{union} creates a new decision diagram for two input diagrams as follows. Starting from the root nodes $n_1$ and $n_2$ of the two input decision diagrams, it creates a new node $n$ by following the rules shown below.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{(a) A set of integer triples, (b) The corresponding multi-value decision tree representation for the set shown in (a).}
\end{figure}
1) For each edge \((n_1, i, n_1') \in \text{outgoing}(n_1)\), if there is no \((n_2, i, n_2') \in \text{outgoing}(n_2)\), add \((n, i, n_1')\) to \(\text{outgoing}(n)\).
2) For each edge \((n_2, i, n_2') \in \text{outgoing}(n_2)\), if there is no \((n_1, i, n_1') \in \text{outgoing}(n_1)\), add \((n, i, n_2')\) to \(\text{outgoing}(n)\).
3) For each edge \((n_1, i, n_1') \in \text{outgoing}(n_1)\), if there is \((n_2, i, n_2') \in \text{outgoing}(n_2)\), add \((n, i, n')\) to \(\text{outgoing}(n)\) where \(n' = \text{union}(n_1', n_2')\).

During the union operation, whenever a node is created, the unique table is checked to see if there is an existing node that is equivalent to the created one. If there exists an equivalent node, such existing node is returned. Otherwise, the created node is returned. Multiple decision diagrams can exist at the same time, and they share the same unique node table. This allows the node sharing among different decision diagrams.

The union operation may need to make a large number of calls to function \(\text{union}\) on different pairs of nodes. To avoid redundant work and improve efficiency, a cache is maintained to store pairs of nodes where function \(\text{union}\) has been applied and their corresponding results from the union operations. If later a call to \(\text{union}\) on the same pair of nodes is made, then the cached result is returned instead of performing the expensive union operations again.

After the union operation, if the input decision diagrams are no longer needed, their nodes can be removed by calling function \(\text{remove}\) on the root nodes \(n\) as follows.

1) Decrement the reference count of \(n\) by 1.
2) If the reference count of \(n\) is 0, remove \(n\) from the unique node table, then for each edge \((n, i, n')\) \(\in \text{outgoing}(n)\), remove \(n'\).

More detailed description on multi-value decision diagrams and the commonly supported operations can be found in [17]. In our method, multi-value decision diagrams are used as a mechanism for storing the reachable states compactly, and the deletion operation for individual paths is not supported as it is not needed for the above purpose.

The example in Fig. 3 shows the same set of integer triples in Fig 2(a) and the corresponding decision diagram. Each edge is labeled with an integer denoting the index of some local state. In the figure, an edge may be labeled with more than one integer. In that case, it corresponds to multiple edges, each of which is labeled with an integer. Another multi-value decision diagram is shown in Fig. 4(a), and the result from merging it with the diagram shown in Fig. 3(b) by function \(\text{union}\) is shown in Fig. 4(b). It is straightforward to verify that all the paths in diagrams in Fig. 3(b) and 4(a) are included in the diagram shown in Fig. 4(b). The resulting diagram represents a set of nine integer tuples. Note that in the shown diagrams, the labels of the nodes are not used to distinguish nodes from different diagrams, therefore nodes from different diagrams with the same labels should not be viewed as the same nodes.

C. Using Decision Diagrams in Reachability Analysis

A direct way to use the multi-value decision diagrams in the reachability analysis shown in Algorithm 1 is to replace the hash table for \(\text{stateTable}\) with a decision diagram. First, let \(\text{mdd\_create}(\vec{s}')\) be a function that takes an integer tuple and returns a multi-value decision diagram with a single path corresponding to \(\vec{s}'\). Then, the code in line 16 in Algorithm 1 can simply be replaced with the following line.

\[
\text{stateTable} = \text{union}(\text{stateTable}, \text{mdd\_create}(\vec{s}'));
\]

Directly using the decision diagrams as shown above is simple, and can be very efficient in memory usage. However, it is not very efficient in runtime. This is due to a large number of \(\text{union}\) calls as one is made for every new state \(\vec{s}'\). Since each \(\text{union}\) call can incur several operations on node creations and checks against the unique node table, a large number of \(\text{union}\) calls can lead to very high overhead when the number of reachable states is large.

To address this problem, multi-value decision trees are used as buffers for a set of states before adding them together into the decision diagram \(\text{stateTable}\). The basic reason is that it is much more efficient to add a state into a decision tree. On the other hand, a decision tree generally requires more memory to store the same number of states than a decision diagram. In this method, a decision tree is used as a buffer. The search algorithm adds states into the decision tree first. When a sufficiently large number of states are added such that a preset memory threshold is exceeded, this decision tree is compressed to become a decision diagram, and consequently
to be handled, or improves verification coverage by exploring stored with some extra runtime. This allows larger examples the whole state space or a much larger portion of it to be for large examples, the decision diagrams would allow either state space, hash tables are a better option. On the other hand, memory also very quickly. Therefore, for examples with small states to be explored in a reasonable amount of time. Hash storing reachable states is to allow much larger number of states. The main purpose of using the decision diagrams for representing reachable sets is also implemented in Platu, and used as the base of state

function compress takes a decision tree traverses a decision tree from the root node, and merges all equivalent nodes.

IV. EXPERIMENTS

The multi-value decision trees and diagrams described in this paper are implemented in a package, and integrated into an asynchronous system verification tool Platu, an explicit model checker implemented in Java. Experiments have been performed on a number of examples. These examples include asynchronous circuit designs from previously published papers [11], [6], [18], [19], [13]. Other examples are selected from the BEEM benchmark suite for explicit model checking [14]. This benchmark includes a large number of models of communication protocols, mutual exclusion algorithms, etc. All the examples used in the experiments have medium or high complexity to show the difference in using the hash tables and the decision diagrams.

In the experiments, all examples are ran by using the same depth-first search algorithm but with three different ways for storing reachable states: hash table, multi-value decision diagram, and multi-value decision diagram with a decision tree as buffer to reduce runtime overhead. To have somewhat fair comparison, the model checker SPIN [10] is not used in the experiments since SPIN is implemented in C while Platu is implemented in Java. It is well known that the Java applications generally have serious memory overhead. Instead, the state compression technique implemented in SPIN is also implemented in Platu, and used as the base of state representation for all three methods representing reachable sets of states.

Among all the examples, some have very large state space, and it can take enormous amount of time to find all reachable states. The main purpose of using the decision diagrams for storing reachable states is to allow much larger number of states to be explored in a reasonable amount of time. Hash tables can be accessed very efficiently, but usually exhaust memory also very quickly. Therefore, for examples with small state space, hash tables are a better option. On the other hand, for large examples, the decision diagrams would allow either the whole state space or a much larger portion of it to be stored with some extra runtime. This allows larger examples to be handled, or improves verification coverage by exploring significantly more states. In all experiments, upper bounds on time and memory are set to 900 seconds and 2 GB. The results collected include the actual runtime, memory usage, and the total number of reachable states found at termination of the search algorithm. In cases of time-out or memory-out, the number of reachable states is recorded. All experiments are performed on a iMac desktop with a Intel Quad-core processor. Only a single thread is used for all experiments.

The results are shown in Table I. The first column Column shows the different models. The numbers enclosed in parenthesis for the first few examples indicate the number of state variables. Since these examples are asynchronous circuits, the type of their state variables is boolean. The following examples are the models of mutual exclusion algorithms from the BEEM benchmarks [14]. The remaining columns are divided into three groups, one for results from using each different state representation. Columns under “Base” show results with the hash table is used. Columns “MDD” show the results with the MDD alone used. Finally, the columns under “MDD-Hybrid” show the results with MDD used where a decision tree is used to speed up the search. Among the five columns in each group, “Time” and “Mem” show the total runtime and the peak memory used during the search. Note that the memory numbers include those used for storing reachable states and other necessary data structures for the search such as stacks. Column “|S|” shows the total number of states found at the termination of the search. Note that this number shows all reachable states of an example if the run does not time out (< 900 second) and does not exhaust the allocated memory (< 2000 MB).

The first conclusion that can be drawn from the table is that the search using a hash table is the fastest for almost all examples. For examples with low complexity such as at.4 and fischer.3. However, using a hash table also causes the search to exhaust 2 GB for large number of examples as shown by the cells in the table under “Mem” filled with 2000. On the other hand, from the columns under “MDD”, the search with MDD used is significantly slower, and it times out for most examples. At the same time, using MDD allows more states to be searched with less memory. Take dmeN5 as an example. using MDD causes the search to time out, but over two times more states are found with slightly less memory than 2 GB used. The same conclusion is also observed for anderson.8, lamport.7, and so on. For some other examples, the search with the MDD finds less number of states due to the overhead from converting states into the MDDs as explained before. Finally, from the columns under “MDD-Hybrid”, the search in this mode finds much more states for almost every example within both limits of time and memory. This shows the effectiveness of using decision trees as buffer for the MDD representation.

To better compare these three representations, two more values are computed from the results for each example: search speed (SS) and state space density (SSD). The numbers in the column under SS show the numbers of states found per second. This is a rough measure of how fast the search runs. The numbers in the column under SSD show the numbers of states per MB of memory. This is a rough measure of the efficiency of different representations for storing states. For both measures, it is better and more efficient if the numbers
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>arbN5 (44)</td>
<td>4.1</td>
<td>53</td>
<td>227472</td>
<td>55481</td>
<td>4929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arbN7 (62)</td>
<td>292</td>
<td>2000</td>
<td>12582972</td>
<td>43092</td>
<td>6294</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arbN9 (80)</td>
<td>113</td>
<td>2000</td>
<td>9052417</td>
<td>80110</td>
<td>4526</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmcN3 (33)</td>
<td>2.77</td>
<td>49</td>
<td>267999</td>
<td>96751</td>
<td>5464</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmcN4 (44)</td>
<td>199</td>
<td>1725</td>
<td>15692028</td>
<td>78854</td>
<td>9097</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dmcN5 (55)</td>
<td>149</td>
<td>2000</td>
<td>12590803</td>
<td>84502</td>
<td>6295</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infoN8 (34)</td>
<td>119</td>
<td>653</td>
<td>3572036</td>
<td>30017</td>
<td>5470</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infoN10 (42)</td>
<td>292</td>
<td>2000</td>
<td>9481673</td>
<td>32471</td>
<td>4741</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mmm (55)</td>
<td>900</td>
<td>1528</td>
<td>8284885</td>
<td>92082</td>
<td>5422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pipect (50)</td>
<td>168</td>
<td>2000</td>
<td>10485516</td>
<td>62414</td>
<td>5243</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tagunit (48)</td>
<td>269</td>
<td>2000</td>
<td>3302262</td>
<td>12278</td>
<td>1651</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anderson.6</td>
<td>141</td>
<td>2000</td>
<td>1201306</td>
<td>85201</td>
<td>6007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>anderson.8</td>
<td>41</td>
<td>2000</td>
<td>9022879</td>
<td>220070</td>
<td>4511</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>at.4</td>
<td>102</td>
<td>1018</td>
<td>6597246</td>
<td>64679</td>
<td>6481</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>at.5</td>
<td>154</td>
<td>2000</td>
<td>10489013</td>
<td>68110</td>
<td>5245</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>at.6</td>
<td>167</td>
<td>2000</td>
<td>10100451</td>
<td>60482</td>
<td>5050</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>at.7</td>
<td>361</td>
<td>2000</td>
<td>12583081</td>
<td>34856</td>
<td>4292</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bakery.8</td>
<td>214</td>
<td>2000</td>
<td>17042811</td>
<td>79639</td>
<td>8521</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>driving_phils.3</td>
<td>1</td>
<td>13</td>
<td>436</td>
<td>436</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>driving_phils.4</td>
<td>204</td>
<td>2000</td>
<td>14039859</td>
<td>68823</td>
<td>7020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>driving_phils.5</td>
<td>78</td>
<td>2000</td>
<td>3599664</td>
<td>46150</td>
<td>1800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fischer.3</td>
<td>52</td>
<td>409</td>
<td>289650</td>
<td>55706</td>
<td>7082</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fischer.4</td>
<td>25</td>
<td>342</td>
<td>1272254</td>
<td>50890</td>
<td>3720</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fischer.5</td>
<td>340</td>
<td>2000</td>
<td>10780321</td>
<td>31707</td>
<td>5390</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fischer.6</td>
<td>296</td>
<td>2000</td>
<td>8237316</td>
<td>27550</td>
<td>4119</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fischer.7</td>
<td>365</td>
<td>2000</td>
<td>10563261</td>
<td>29132</td>
<td>5317</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lamport.5</td>
<td>10</td>
<td>135</td>
<td>1066799</td>
<td>106680</td>
<td>7992</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lamport.6</td>
<td>1</td>
<td>11</td>
<td>519</td>
<td>519</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lamport.7</td>
<td>252</td>
<td>2000</td>
<td>18075514</td>
<td>71728</td>
<td>9038</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lamport.8</td>
<td>1</td>
<td>11</td>
<td>50</td>
<td>50</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcsc.5</td>
<td>127</td>
<td>2000</td>
<td>12401513</td>
<td>94815</td>
<td>6021</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peterson.4</td>
<td>11</td>
<td>147</td>
<td>1119599</td>
<td>101778</td>
<td>7616</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peterson.5</td>
<td>109</td>
<td>2000</td>
<td>12583035</td>
<td>115441</td>
<td>6292</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peterson.6</td>
<td>88</td>
<td>2000</td>
<td>11101039</td>
<td>126257</td>
<td>5555</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peterson.7</td>
<td>222</td>
<td>2000</td>
<td>15975060</td>
<td>71960</td>
<td>7988</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>phil.6&quot;</td>
<td>2.5</td>
<td>159</td>
<td>241660</td>
<td>96664</td>
<td>1520</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>phil.7</td>
<td>44</td>
<td>2000</td>
<td>6927036</td>
<td>157433</td>
<td>3464</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>phil.8</td>
<td>72</td>
<td>437</td>
<td>914351</td>
<td>126993</td>
<td>2092</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>szymanski.4</td>
<td>204</td>
<td>282</td>
<td>2313863</td>
<td>113425</td>
<td>8205</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>szymanski.5</td>
<td>155</td>
<td>2000</td>
<td>15908655</td>
<td>102636</td>
<td>7954</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I**

Results from using different state representations on a set of asynchronous circuit designs and models of mutual exclusion examples. Entries filled with 900 under Time and 2000 under Mem indicate time-out and memory. Columns under "Base" show results using hash tables. Columns under "MDD" show results using MDDs, while those under "MDD-Hybrid" show results using MDDs combined with decision trees as buffers.
are larger. The averages of these two measurements are shown in the last row for every group. From the average numbers, it can be seen alternatively that using the hash tables leads to the highest search speed but the least memory efficiency. Using MDD alone leads to the highest memory efficiency, but it is the slowest to use. The MDD with the decision tree buffering achieves good balance between runtime and memory as it is slightly slower than using hash tables, and slightly less memory efficient than using the MDDs alone. This good balance allows the search to find much more states for many examples. Take szymanski.5 as an example, the search exceeds the memory limits by finding a little over 15 million states when the hash table is used, while the search runs out of time after finding about 2 times more states with only 527 MB memory used when the MDDs alone are used. However, the MDDs with buffering allow all reachable states to be found under both time and memory limits.

From the experiments, for examples that take long time to run for the search using hash tables, using the MDDs would lead to less number of states to be found as it is more expensive to use the MDDs even with faster buffering techniques. On the other hand, for examples that have large state space, the MDDs are much more efficient. Furthermore, buffering seems a very promising approach for handling larger designs. Since the MDDs implemented in Platu are not intensively optimized, there might be more potential for making them even more efficient. Meanwhile, it would be interesting to investigate if using the existing BDD packages such as CUDD [16] could bring more efficiency as they commonly support variable recording, which is well known critical for efficiency and compactness of BDD representations.

One last point to note is that in the work described in this paper, these different approaches are experimented to find out how states can be stored efficiently. The depth-first search algorithm itself remains the same. This indicates that partial order reduction can be naturally integrated with the MDDs to allow much larger designs to be handled. However, partial order reduction is not used in this paper.

V. CONCLUSION

This paper shows how the multi-value decision diagrams are used to store reachable states efficiently and compactly, which allows larger designs to be verified, or more states to be explored thus improving the verification coverage. By using the decision trees as a buffering technique, the overhead of using the multi-value decision diagrams is significantly reduced. In the future, it is interesting to find out how the binary decision diagrams compare against the multi-value decision diagrams in the same context. Moreover, combining the decision diagrams and partial order reduction will also be investigated.

REFERENCES