Memory Efficient State Storage in SPIN

Willem Visser

June 14, 1996

Abstract

The use of an Ordered Binary Decision Diagram (OBDD) to store all visited states during on-the-fly model checking (or reachability analysis) is investigated. To improve the time and space efficiency a state compression technique is introduced. This compression technique is safe, in the sense that no two unique states will have the same compressed representation. A number of examples are used to evaluate an experimental implementation of the OBDD state store within the SPIN validation tool. In all the examples a reduction in space is achieved when using the OBDD state store as opposed to the more traditional hash table state store. The memory and time usage when combining partial orders with the OBDD state store is also considered.

1 Introduction

Temporal logics can express changes over time without introducing time explicitly and is therefore suitable for specifying many correctness properties of concurrent systems. Since many interesting programs can be modelled as finite-state systems, it was a significant development when algorithmic methods were discovered to verify temporal properties of finite-state systems[6]. Since that time a variety of concurrent systems have been modelled and automatically verified in this fashion, these include, communication protocols, hardware designs and operating system kernels. A finite labelled state-transition graph (also called a *Kripke* structure) is used to represent the behaviour of concurrent system and propositional temporal logic formulae are used to express desired properties of the system. A so-called *model checker* is then used to check whether the Kripke structure satisfies (is a model of) the temporal formula specifying the required behaviour. Surveys of model checking techniques can be found in [23] and [5].

Early model checking algorithms[7] required the complete state graph be generated before-hand and kept in memory throughout the model checking process. Due to the so-called *state explosion* problem (the number of reachable states grow exponentially in the number of concurrent components) when analysing sufficiently large concurrent systems, the state graph is however often too large to fit into memory. This limits the size of the systems that can be model checked with these algorithms. One possible solution to this is only to generate the part of the state graph required to validate (or invalidate) the given temporal logic requirement. This technique is commonly referred to as *on-the-fly* model checking, since the state graph is generated during model checking[21, 1]. On-the-fly algorithms generate the state space in a depth first manner and keeps track of all reached states to avoid doing unnecessary work. The boundaries of on-the-fly techniques were advanced considerably when it was realised that much of the state explosion problem was due to the generation of all interleavings of independent transitions in different concurrent components. *Partial order reduction* techniques were therefore introduced to ensure that many of these unnecessary interleavings are not explored during state generation [20, 11, 17, 15].

Arguably the biggest advancement in reducing the limitations imposed by the state explosion problem was made with the advent of *symbolic model checking*[16]. In symbolic model checking the transition relation is no longer represented explicitly, but rather implicitly by encoding it with *ordered binary decision diagrams* (OBDDs) [4]. The temporal logic formula to be checked is translated into its fixed point representation. The model checking algorithm proceeds by calculating these fixed points by performing operations on the OBDD that represents the transition relation. After each iteration of the fixed point calculation the set of states obtained are again represented by an OBDD. The main advantage of this algorithm is therefore that the state graph is never explicitly constructed. If the state space of a system exhibits some form of regularity then the OBDD representation of a set of its states can be very compact. Symbolic model checkers are not goal directed, since the complete state space is generated during model checking.

Unfortunately, both on-the-fly techniques and those using OBDD encodings have potential drawbacks. The main problem with on-the-fly algorithms is the storage space required to record already visited states during state generation. For OBDDs, on the other hand, the size of an OBDD is heavily dependent on the ordering of the boolean variables within it. An even bigger disadvantage of OBDDs is that certain boolean functions will always have an exponentially sized OBDD regardless of the variable ordering used[3]. Integer multiplication is an example of such a function. The size of the OBDD representing the transition relation can therefore cause the symbolic model checking to become impractical.

In this paper we will investigate a combination of on-the-fly model checking and OBDD encodings. The on-the-fly ethos will be adopted: the algorithm will be goal directed and therefore the state space will be generated during the model checking process. The transition relation will therefore not be represented by an OBDD, but, one will be used to represent the set of states already visited during the state generation. If there are any regularity in the state space generated this would be reflected in the size of the OBDD, i.e. its size will increase polynomially, rather than exponentially, with the increase in states visited. OBDD based model checkers have until now been used mostly for hardware verification, since hardware systems usually exhibit regular state spaces. These systems also tended to be synchronous. Since asynchronous software (and hardware) systems have not received the same amount of attention from the symbolic model checking community, it was decided to apply our technique to these type of systems. The effect of using partial order techniques in combination with OBDDs will also be investigated.

The outline of the paper is as follows. Section 2 will introduce the basic on-the-fly algorithm used during state generation. The next two sections will be devoted to explaining our use of OBDDs (section 3) and how it is combined with the SPIN [13] on-the-fly validation system (section 4). In section 5 a novel way of state compression will be introduced to allow more efficient OBDD representations. In sections 5.2 and 6, the results achieved by the technique will be discussed as well as the effect of partial order reductions on the OBDD size. The last section will contain conclusions and some future extensions to the algorithm.

2 On-the-fly Model Checking

Although it has long been known that linear time temporal logic can be efficiently model checked by only generating relevant parts of a state graph[21], it is has only comparatively recently been shown that the same is true of the branching time temporal logics CTL and CTL* [1, 2]. Here we will only concentrate on efficient state storage during on-the-fly model checking and therefore not be concerned about the temporal logic being used.

In Figure 1 the standard depth-first search algorithm is shown that implements the state generation during model checking. When used during model checking the iterative form of the algorithm is used and after each state s is generated the model checker will determine the truth-value of the current temporal logic subformula from the value assignments within s. Since the focus here is on state generation, rather than the model checking algorithm, and the recursive algorithm in Figure 1 is easier to follow than the iterative version we will use the recursive one to explain the basic issues involved. Two data structures are used within the search: a set of all states visited during the search, *VisitedStates*, and a stack of states to keep track of the current execution path, *Stack*. During initialisation the initial state, s_0 is entered into *VisitedStates* as well as pushed onto *Stack*. The depth-first search is then invoked by calling *dfs()*. In *dfs* all transitions enabled in the current state, s, at the top of the stack are executed in a depth-first manner. If the first successor state s' of s has not been reached before, i.e. it is not in *VisitedStates*, it is entered into *VisitedStates* and pushed onto *Stack*. By calling *dfs* again s' now becomes the current state. When all the successor states from the current state s have been visited then s is removed from

```
PROCEDURE DFS()

    PROCEDURE dfs()
    BEGIN
        s = top(Stack);
        FOR all transitions t enabled in s DO
            s' = successor(s) after executing t;
            IF s' NOT IN VisitedStates THEN
                Enter s' into VisitedStates;
                Push s' onto Stack;
                dfs()
            END
        END;
        Pop s from Stack
    END dfs;

BEGIN
    Enter $s_0$ into VisitedStates;
    Push $s_0$ onto Stack;
    dfs()
END DFS;
```

Figure 1: Depth-first State Generation during Model Checking.

the stack and previous state on the stack becomes the current state.

The representation of the set *VisitedStates* has been the focus of much research[12, 10, 24]. This is not surprising since this is the part of on-the-fly model checking that will determine its tractability when checking large designs. The most commonly used method is to represent the set as a large hash table of states. When a new state is generated it is hashed to obtain the index into the table. Since the states must be stored in its entirety, to allow for comparisons during the resolving of possible hash conflicts, this method is not very efficient when a large number of states must be stored. An optimisation on this method, was to allow table entries to be over-written when the table became too full[11, 22]. The intuition was that the chance of a state being revisited after it was over-written would be small in practice. However, empirical results showed in many cases as soon as the table entries were over-written the validation time increased exponentially. The most novel approach to date is the so-called *bitstate hashing*[12] used in the SPIN validation system. This technique requires a large vector of bits (of fixed size) to be maintained in memory to keep track of previously generated states. A hashing technique is used to compute an index into this bit vector from the value of each state. However, since the validation results can be invalid when a hash conflict occurs this technique is not always desirable when validating safety critical systems. The coverage obtained with bitstate hashing can be improved by either increasing the number of hashing functions used or using more than one bit to hash into (*hashcompact* [24]). A summary of bitstate hashing as well as comparisons with the hashcompact method can be found in [14].

In the next section we will introduce OBDDs as a possible alternative to the above mentioned representations of *VisitedStates*. After implementing the OBDD to record visited states in the SPIN validation system a further state compression method was added to reduce both the time and memory requirements of this method. This compression method can also be used with the traditional hash table methods to reduce the memory required, at the price of adding a small constant factor to the run-time.

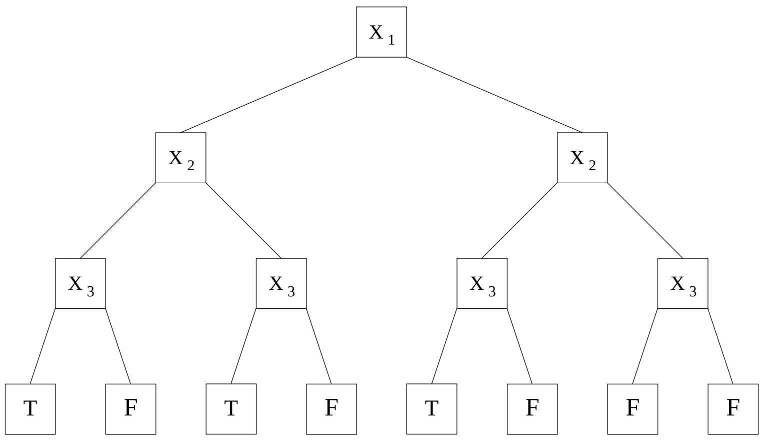

Figure 2: Decision tree for the boolean function $f(x_1, x_2, x_3) = \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 + x_1 \cdot \overline{x}_2 \cdot \overline{x}_3$

3 OBDD State Store

OBDDs are directed acyclic graphs that represent boolean functions in a canonical form. As an example, let us consider the OBDD that represents the function $f(x_1, x_2, x_3) = \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 + x_1 \cdot \overline{x}_2 \cdot \overline{x}_3$, where $\cdot$ denotes the AND operation and $+$ the OR operation. The decision tree for this function is given in figure 2. Left branches from a node indicates the variable is 0 and similarly right branches indicate value 1. Terminal nodes are labelled with T for true and F for false. To construct the OBDD representation of f a total ordering on the variables of f must be imposed. In figure 2 this ordering is $x_1 < x_2 < x_3$. There are three transformation rules on these graphs, that do not alter the function represented, but may reduce their size. An OBDD is the name given to the graph that cannot be reduced any further. The rules are:

Remove Duplicate Terminals There must remain only one terminal with a given label. All arcs to duplicate labelled terminals must be redirected to the remaining one. A further optimisation is not to show any arcs leading to F, although they are implicitly there. In figure 3 this optimisation is performed on the decision tree for f.

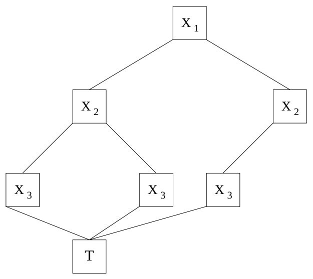

Figure 3: Removing duplicate T terminals and explicit F terminals from the decision tree of f

Remove Duplicate Nonterminals If two nonterminal nodes have the same label and their left and right branches are the same then one must be removed. All arcs must be redirected to the remaining one. This is shown in figure 4 where the three nodes labelled x_3 (with left arc to T and right to F) in figure 3 have been replaced by one x_3 node.

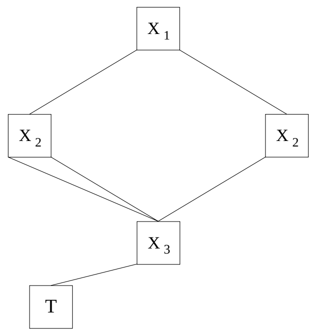

Figure 4: Removing duplicate nonterminals.

Remove Redundant Tests If both arcs from a nonterminal,say n,point to the same node,say n', then n must be removed and all its incoming arcs must be redirected to n'. In figure 4 the left-hand x_2 is such a redundant node since both its left and right arcs point to x_3. The OBDD for f is shown in figure 5 after this last transformation is performed.

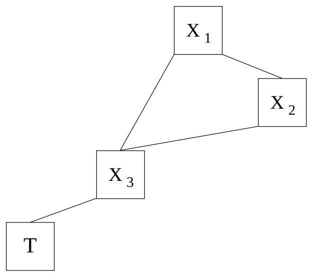

Figure 5: OBDD for function f after removing redundant tests.

These rules must be applied repeatedly, since the application of one rule can cause more transformations to be possible on the resulting graph. It will now be shown how an OBDD can be used to represent the set of states visited during on-the-fly model checking. First, recall that the state, s, of a system consists of a vector of bits, $s = x_0, x_1, \cdots, x_n$, where x_i is bit i in this so-called state vector. If a state, s, is visited during a traversal it can be represented by the boolean function $f(s) = 1$, or, in other words, $f(x_0, x_1, \ldots, x_n) = 1$. For example, consider a state vector with only three bits, x_1, x_2 and x_3 and a state is written as $< x_1x_2x_3 >$. Assume the following states are visited $< 000 >$, $< 010 >$ and $< 100 >$. Then the OBDD for the boolean function $f(x_1, x_2, x_3) = \overline{x}_1 \cdot \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_2 \cdot \overline{x}_3 + x_1 \cdot \overline{x}_2 \cdot \overline{x}_3$ will represent the states visited. Thus if the variable ordering $x_1 < x_2 < x_3$ is assumed then the OBDD in figure 5 represents the visited states for this example. To add a state, s, to the set of visited states it is therefore necessary to first convert s to its OBDD representation and then to perform an OR-operation on this OBDD and the one representing the set of states already visited. The resulting OBDD will represent the set of states with s added. To establish if a new state, s, is in the set of already visited states, the OBDD representing the visited states is traversed according to the values of the bits in s. For example, to check if the state $s =< 010 >$ is in the OBDD of figure 5 we check if node x_1 has a left arc–since $x_1 = 0$ in s–this arc is traversed to reach node x_3; now since $x_3 = 0$ in s the left arc of x_3 is traversed and the resulting terminal node T is found indicating s is in the OBDD. If the existence of state $s =< 011 >$ was checked then at node x_3 the right branch would have been taken leading to the terminal F indicating s

has not been visited yet. The following three operations are thus the only requirements for implementing an OBDD state storage mechanism during on-the-fly model checking:

EncodeOBDD(s) This function takes as input a state and returns the OBDD encoding of the state.

OR(OBDD1,OBDD2) Returns the OBDD that is obtained after applying the transformation rules to the result of adding OBDD1 to OBDD2.

Exists(OBDD1,s) Returns true if state s is in OBDD1 else false.

The advantage of using an OBDD to record visited states becomes apparent when large parts, or even, the complete reachable state space of a model is visited. Since this will in most cases cause the OBDD to become smaller, due to the applying of the transformation rules. For example consider the previous example where the states $< 000 >$, $< 010 >$ and $< 100 >$ were visited. If we now assume the rest of the reachable states of this example (the five states $< 001 >$,$< 011 >$, $< 101 >$,$< 110 >$ and $< 111 >$) are visited as well then the OBDD representing the visited states will only contain the terminal node T.

3.1 Example

Consider the mutual exclusion problem for two processes where each process ($i = 1, 2$) can be in one of three code regions: noncritical (N_i), trying (T_i) or critical (C_i). A binary semaphore S is used to protect the critical region. The value of the semaphore is indicated by S_i, where i can be 0 or 1. A process can only enter its *critical* region from its *trying* region if the value of the semaphore is 0. When a process enters its critical region the value of the semaphore becomes 1 and on leaving the critical region and entering the *noncritical* region the value of the semaphore becomes 0 again. The state of the system consists of the values of the control variable for $process_1$, $process_2$ and the *semaphore* variable and is described by $(process_1, process_2, semaphore)$. An asterisk ("*") in the guard of a transition indicates that the field can hold any value for the guard to be satisfied in that state. The action part of the transition will indicate how each field is changed, with "*" indicating unchanged fields. The start state is (N_1, N_2, S_0) and the transitions are the following:

$$(N_1, *, *) \rightarrow (T_1, *, *) \quad (1)$$
$$(T_1, *, S_0) \rightarrow (C_1, *, S_1) \quad (2)$$
$$(C_1, *, *) \rightarrow (N_1, *, S_0) \quad (3)$$
$$(*, N_2, *) \rightarrow (*, T_2, *) \quad (4)$$
$$(*, T_2, S_0) \rightarrow (*, C_2, S_1) \quad (5)$$
$$(*, C_2, *) \rightarrow (*, N_2, S_0) \quad (6)$$

The two control variables each have three possible values: N_i (0), T_i (1) or C_i (2). The semaphore has two values: S_0 (0) or S_1 (1). The two control variables will thus have two bits and the semaphore one bit allocated for it in the state vector (Figure 6).

State vector: 5 bits		
$Process_1$	$Process_2$	Semaphore
4..3	2..1	0

Figure 6: A state vector for the mutual exclusion system

During initialisation the initial state (N_1, N_2, S_0), which corresponds to the state vector $< 00000 >$, is inserted into the set of visited states. The next state to be visited is (T_1, N_2, S_0) ($< 01000 >$) which is generated after executing transition 1. The OBDD resulting from adding this state to the initial state is shown on the left-hand of figure 7. In the rest of figure 7 the resulting OBDDs are shown after adding the next three states visited in the depth-first search. The OBDD on the right therefore contains the states $< 00000 >$, $< 01000 >$, $< 10001 >$ and $< 00010 >$. In figure 8 the OBDD is shown after each of the remaining four states of this model is added to the set of visited states.

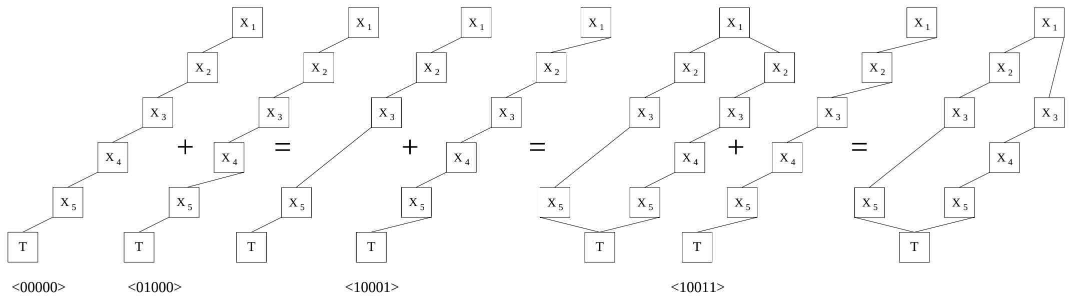

Figure 7: OBDD representations for adding the first four states of the mutual exclusion system to the state store.

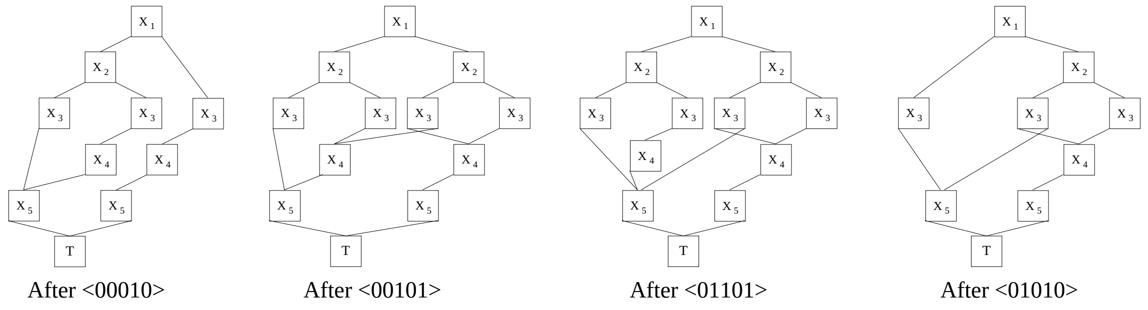

Figure 8: OBDD representations after the last four states of the example are added. The rightmost one represents all eight reachable states of the system.

The memory requirement for this method is that of the largest OBDD generated during the search. For this example it is the memory required to represent the second and third last OBDDs (see figure 8) which both have 12 nodes (excluding the implicit F-terminal). Considering that the largest OBDD for a boolean function with five variables is 14, this example perform close to the worst-case[1]. In general the OBDD representing the state store will have a best-case performance that is *linear* and a worst-case that is *exponential* in the number of bits in the state vector. As it is unclear what the heuristics should be for choosing a variable ordering, we will only consider one variable ordering in the sequel. The variable order will be the same as the order of the bits in the state vector.

4 SPIN with OBDD State Store

With the aid of the BDD package[2] from Carnegie Mellon the two functions EncodeOBDD and Exists were implemented and the `bdd_or` function provided were used to add two OBDDs together. The functions were added to the SPIN[3] tool in a straight-forward fashion to obtain an experimental implementation for an on-the-fly model checker that uses an OBDD to record visited states.

The first example on which the method is evaluated is a model consisting of n concurrent counters, where each counter loops through the values 0 to 9. This example was chosen because it exhibits a regular state space and would therefore allow the OBDD state store to perform close to optimally. In Table 1 the results for 4, 5 and 6 concurrent counters are shown when using the OBDD state store compared with the table storage. The rest of the examples in Table 1 are all real-world models: pftp is a protocol model and snoopy a cache coherence model that are both available with the SPIN system. Scheduler is a model of a process scheduler that form part of a commercially available micro-kernel[8] and Address

[1]If n is the number of variables in a boolean function then there are $3 * 2^{\frac{n}{2}} - 2$ nodes when n is even and $2 * (2^{\lceil \frac{n}{2} \rceil} - 1)$ nodes if n is odd in the largest possible OBDD for the function (excluding the implicit F-terminal).

[2]Available by ftp from emc.cs.cmu.edu (pub/bdd/bddlib.Tar.Z)

[3]Available by ftp from netlib.att.com (netlib/spin/spin2.8.Src.tar.Z)

Model	Table Store		OBDD Store			Bits	States
	Memory (Mb)	Time (secs)	Memory	Nodes (largest)	Time		
counter 4	0.1	11	0.3	158	358	72	10^4+1
counter 5	1	647	0.32	233	1580	88	10^5+1
counter 6	32	3012	0.33	283	18780	94	10^6+1
address interface	0.7	2.8	0.74	22470	580	608	9491
snoopy	7.7	100	9.97	421165	36824	704	91920
scheduler	14.7	202	0.95	53354	36259	472	256929
pftp	48	600	>36	>10^6	>60000	928	439895

Table 1: Comparison between an OBDD and a (hash) table for storing visited states in SPIN.

Interface is a model of part of an asynchronous microprocessor [9]. The comparison is done on the memory used in megabytes and the time[4] taken in seconds for the two methods. The number of nodes in the largest OBDD required, the number of bits in the state vector and the number of reachable states are also indicated. As expected the counter examples show the biggest improvement when using the OBDD state store. The memory for the table store increase exponentially (since the number of states increase exponentially) while the OBDD only require a linear increase, in the number of concurrent counters.

Unfortunately for the real-world examples the OBDD state store does not perform as well as in the contrived examples of concurrent counters. For example the generation of the state space for the pftp model was aborted after 16 hours, since only half the number of states were generated by that time. The reason for this is the performance of the OBDD store is critically dependent on the number of bits in the state vector. Since every state generated during the search must first be checked against the visited states in the OBDD and then encoded in an OBDD if it has not been visited, the Exists and EncodeOBDD functions are dependent on the state vector size. Therefore, since the size of the OBDD representing a new state is dependent on the number of bits in the state vector, so too will the `bdd_or` function be dependent on the state vector length. In the next section a compaction scheme will be given that will compress the length of the state vector, thus allowing the OBDD state store to be more effective.

5 State Compression

A model that has n bits in its state vector, seldom visits all 2^n states. In SPIN this is mostly due to the fact that only predefined types are available. Therefore when a variable will only be assigned values in the range $0 \ldots 7$, it would still be required to be of type *byte* which would mean 8 bits in the state vector instead of the required 3. In general, however, it is because variables in different concurrent processes do not "interleave" all their possible values. Finding ways of reducing the state vector size, without losing information, would therefore seem a worthwhile pursuit. Strangely, however, *probabilistic* methods [14, 24, 19] are attracting more attention than *safe* reduction methods[22] in the research community. Probabilistic methods allow the state vector to be compressed in such a fashion that different states might have the same compressed state. During on-the-fly model checking this might cause a search to be truncated prematurely in which case certain parts of the state space might be ignored. Here a safe reduction method will be introduced that will reduce the memory requirements for a large class of models at the cost of an acceptable run-time overhead.

The intuition behind the method is firstly described. In the same fashion that a model will seldom visit all its 2^n possible states, so too will a process seldom visit all of its potential states. Furthermore, a state explosion is seldom caused by one process visiting a large number of states, it is more commonly caused by the cartesian product of a number of "small" concurrent processes. Now if we consider a process that is allocated m bits in its state vector but it visits only $k = 2^n$ states where $k \ll 2^m$ then $m-n$ bits will be wasted every time a state is stored during the validation. This becomes more significant when this process is part of model that generates a large number of states, since $m-n$ bits will be wasted

[4]The tests were run on a SPARC10 with 64Mb of memory.

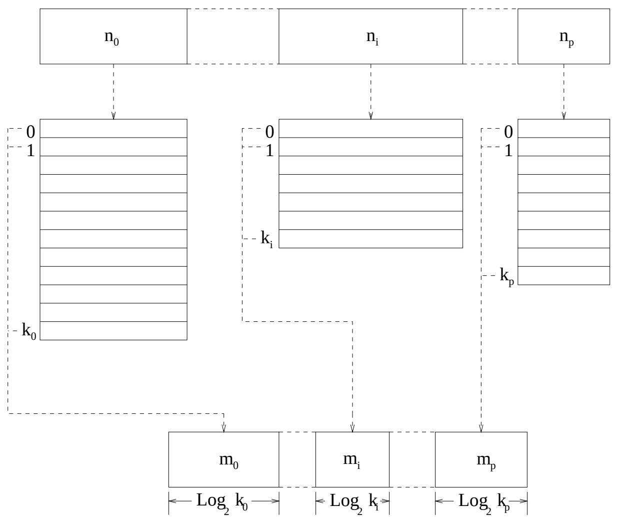

Figure 9: State vector compression via intermediate tables.

not only when the process generates a new state, but whenever any of the other processes in the model cause a new state to be generated. It would therefore seem sensible to only store each of the 2^m states for the process once and only refer to them indirectly when the states are stored. This is achieved by storing the k m-bit values in a table and using the index in the table as the state in the state vector. This would mean only n bits would be required in the state vector for this process. The table will take-up memory but this will be offset when a large enough number of states is visited. Thus the only penalty will be in the run-time since the lookup in the table must be performed for every state generated.

In general, consider the state vector of length n to be constructed of $p+1$ parts each of length n_i with $i = 0 \ldots p$. The parts n_i might be the bits allocated to a process, a variable or even a byte boundary. For every part n_i allocate a table t_i with k_i entries. Let the compressed state be of length m also consisting of $p+1$ parts each of length $m_i = log_2 k_i$. When a new state $s = n_0 n_1 \ldots n_{p-1}$ is generated take part n_i and check if these bits are in the table t_i, if so, take the table index and assign it to m_i; if n_i is not in the table find the next open slot (starting from slot 0) , insert n_i, and assign the index to m_i. The compression technique is illustrated in figure 9. When will this compression fail to save memory? If there are too many unique n_i parts then the table t_i will become too large, or if a fixed table size is assumed, the table will overflow. Thus this compression method is only suitable for models where each n_i part only has $k \ll 2^{n_i}$ unique states. There is also a break even point in the number of states that need to be generated before the memory used for the tables is offset. This value can be calculated as follows:

Let $|s|$ be the number of states generated. Memory will be saved when the memory used without compression is greater than the memory required if compression is used. Therefore,

$$|s| \sum_{i=0}^{p} n_i > |s| \sum_{i=0}^{p} log_2 k_i + \sum_{i=0}^{p} m_i n_i$$

and thus if,

$$|s| > \frac{\sum_{i=0}^{p} m_i n_i}{\sum_{i=0}^{p} n_i - \sum_{i=0}^{p} log_2 k_i}$$

This is read as, the number of states visited must be larger than the number of bits in the tables divided by the number of bits saved in each state. This fits in nicely with the intuition that the method will only work well if the tables are relatively empty, i.e. each n_i part only has a few unique states.

Model	Compress			Collapse			New Collapse		
	Memory	Time	Bytes	Memory	Time	Bytes	Memory	Time	Bytes
address interface	0.17	4.1	19	0.25	5.4	27	0.11	2.3	12
snoopy	1.92	29	22	4.0	33	46	1.57	23.3	18
scheduler	3.67	221	15	7.8	234	32	4.41	226	18
pftp	12	741	29	10.9	730	26	5.8	659	14

Table 2: Results with state compression added to SPIN.

The implementation used here is to divide the state vector into 4 byte parts, i.e. every n_i is 4 bytes long. The table sizes for all t_i is fixed at 256, i.e. k_i is 256. Thus every n_i will be compressed into one byte in the compressed state. This configuration was chosen due to its simplistic implementation in the SPIN tool. The memory required by one table will therefore be $256 * 4$ bytes (1 kilobyte). Thus if the state vector is n bytes (note in SPIN the state vector is always a multiple of 8 bits) long the tables will require $\lceil \frac{n}{4} \rceil$ kilobytes of memory.

5.1 Related Work

Independently of the work reported here the compression technique described above was implemented in the SPIN system by Gerard Holzmann. This implementation, referred to as collapsing the state vector, is more general than the compression of 4 bytes into one described above (which will be called compression in the sequel). It allows the collapsing of each process into either 1, 2, 3 or 4 bytes, depending on how many unique states the largest process generates. Similarly all global variables are grouped together and collapsed. Since PROMELA (the language used to describe the models to be checked by SPIN) supports asynchronous communication, every channel (queue) is also collapsed in a similar fashion to each process. The collapsing of each state therefore consists of three parts: globals (as a unit), processes (one or more) and channels (zero or more). It was found that each process (and globals) in snoopy, pftp and scheduler can be collapsed into 2 bytes, and, the processes (and globals) in the address interface were small enough to be collapsed into 1 byte.

In table 2 the comparative results between collapsing and compression are shown. For each method the memory (Mb), time (secs) and the number of bytes in the reduced state are shown. As expected the memory usage show a four fold reduction when using compression. Significantly, though, the time increase is only 1.46 in the worst case. Note that the concurrent counter models cannot be handled by this method, since the 4 byte parts generate more than 256 unique states and therefore cause the tables to overflow. They were therefore not considered in the comparisons. Unexpectedly, the collapse method only performed better in reducing the state size for the pftp model. On closer inspection it was found that redundant information was stored in the state vector when channels were globally defined (as is the case in all four examples). Specifically, the channel contents, was stored in the "globals" section of the reduced state as well as in the "channels" section. The collapse implementation was changed such that no global channel will be stored in the "channels" section of the reduced state. This caused a considerable reduction in the state size as can be seen from the last column in figure 2. Now for all but the scheduler example the collapse method saves memory. A further optimisation to the collapse method would be to allow the user to specify the number of bytes each process will require in the reduced state. Currently, the largest process, globals or channels determine how many bytes will be used during collapsing of the state vector. For example, in the scheduler only the globals (281 states) and one of the processes (274 states) require 2 bytes and all the other processes generate less than 256 states and can therefore be collapsed into 1 byte. Therefore, the reduced state of the scheduler could be 12 bytes instead of the current 18. As is the case with compression the time increase incurred is not significant.

	OBDD + Compress			OBDD + Collapse		
Model	Memory	Nodes	Time	Memory	Nodes	Time
address interface	0.29	11246	340	0.2	6950	212
snoopy	1.97	108193	3910	2.6	126141	4613
scheduler	0.49	19705	6891	0.48	18474	6185
pftp	6.3	349210	17403	3.43	167767	15870

Table 3: OBDD results with state compression/collapsing added to SPIN.

5.2 Combining Compression with the OBDD State Store

In the previous section it was shown how the state vector length can be reduced. The effect of this reduction on the performance of the OBDD state store will now be investigated. Note however that any reduction technique can be combined with the OBDD state store, even probabilistic ones. The results of this experiment are shown in table 3. The results indicate that the compression scheme now makes the OBDD state store a much more viable option. For instance, the pftp model that took more than 16 hours to generate only halve its reachable states when just the OBDD state store was used can now complete the search in a reasonable time when compression is added. When considering the memory required for the normal hash table state store (table 1) it is evident that the OBDD state store combined with state compression is more space efficient for all four of the examples. The best reduction is achieved for the scheduler where 30 times less memory is required.

Interestingly, in the snoopy and scheduler models the compression method that yielded the best reduction in the size of the state vector did not save the most memory when the OBDD state store was added. However, in both these cases the difference in state size is small and the results just highlights the unpredictability of the growth of OBDDs. In the pftp example, where the difference in state size is significant, the method which yielded the smaller state vector (collapse) also required less nodes in its largest OBDD.

Unfortunately, time requirements are now becoming an increasing problem. It would seem that the space savings are almost directly proportional in the time increase. This would seem to indicate that this method of combining an OBDD state store and state compression will only be suitable where memory is a bottle-neck. Until now, however, we have only considered models where the complete state space is traversed. With the use of partial order methods it might be unnecessary to traverse all interleavings of transitions in concurrent processes. Using partial orders in combination with the OBDD state store will be investigated in the next section.

6 Partial Orders

Partial orders will not be discussed in any great detail here, the interested reader is referred to [20, 11, 17, 15]. It is sufficient to know that when partial orders are exploited during state generation it might cause certain states never to be generated. This is due to the fact that certain interleavings of independent transitions might not be considered. In [15] the implementation of partial order rules in the SPIN system are described.

When combining the OBDD state store (with the collapse method of state compression) with the partial order rules the results were surprising (see table 4). The expected reduction in memory is less than might have been anticipated, especially considering so many fewer states are visited. In fact, when considering the second last column in table 4, showing the memory required if partial orders and compression were used with the traditional hash table (i.e. not using an OBDD state store), it can be seen that using the hash table is more memory efficient than the OBDD state store when used in combination with partial orders.

In figure 10 the number of nodes in the OBDD state store is plotted during state generation for

Model	OBDD + Partial Orders				Partial Orders + Collapse	
	States	Memory	Time	Nodes	Memory	Time
address interface	9491	0.21	242	7229	0.11	2.3
snoopy	15295	0.9	171	44543	0.26	3.1
scheduler	9251	0.29	259	12619	0.13	1.1
pftp	95241	1.72	1325	90057	1.27	19.8

Table 4: OBDD state store with Partial Orders

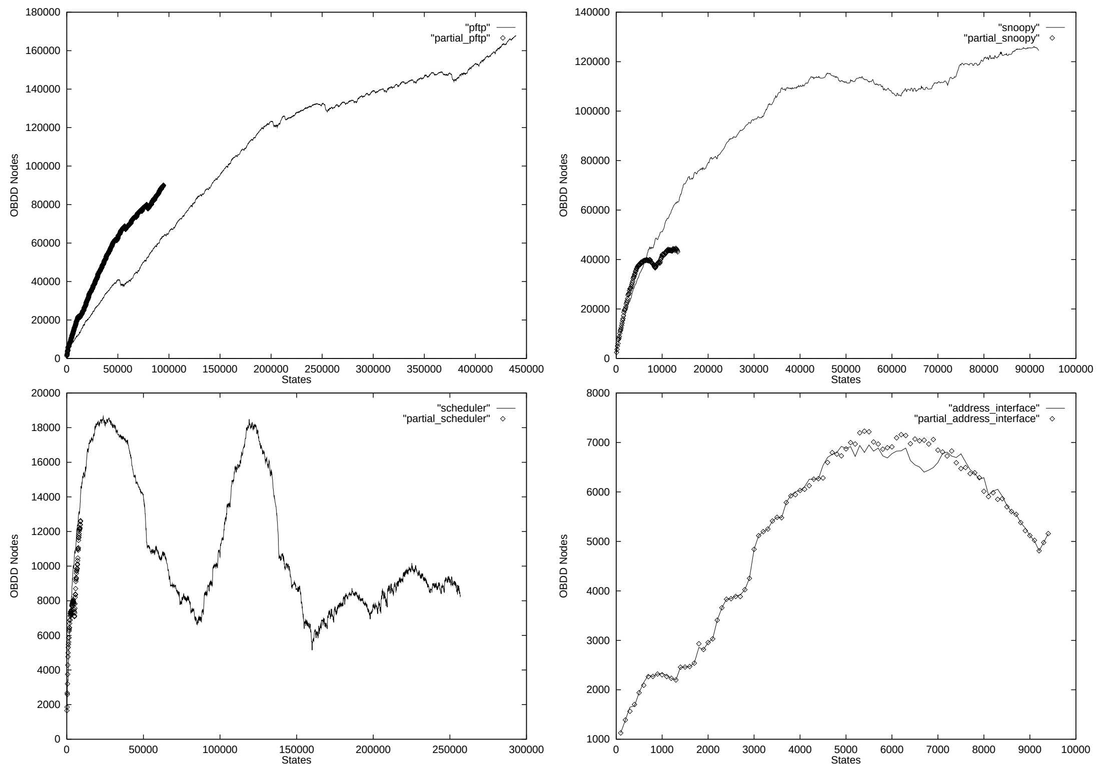

Figure 10: OBDD growth with and without partial order rules.

our four examples (with and without partial order rules). For pftp (top left) the OBDD grows faster when partial order rules are used. Even more surprising is the graph for the address interface: here the same number of states are visited with and without partial order rules, but the OBDD state store requires more nodes when partial orders are enforced! The reason for this behaviour can be explained by remembering that OBDDs perform well when there is regularity in the state space, when partial order rules are used some of this regularity may be removed. This explains the behaviour seen for pftp. In the case of the address interface, however, it is simply that the states were generated in a different order when partial orders were used and this resulted in a loss of regularity in the state space.

7 Conclusion

We have investigated a new approach to reduce the memory requirements during on-the-fly model checking. An OBDD encoding of the states visited during the depth-first exploration of the state space was used to achieve this goal. The initial results (table 1) were not encouraging. Therefore, a state compression technique was given that will reduce the length of the state vector and therefore the memory required during the generation of the state space. This compression method will work well for models that generate a large number of states $k \ll 2^n$ where n is the length of the state vector. Although the

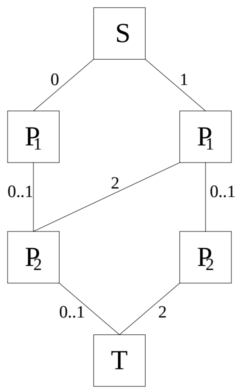

Figure 11: ONDD for the mutual exclusion system. Nodes are labelled with the three process variables.

method was introduced to improve the performance of the OBDD state store, it can be used with any type of state storage method. For instance, combining a specific implementation of the state compression with a hash table as a state store produced a four fold memory reduction while only increasing the time taken by less than a factor of two compared to the case where no compression was used (see table 2). The more general implementation of the compression technique that is currently used in SPIN, was also considered and was found to store unnecessary information in the compressed state in certain cases. The improved version of this so-called collapse method was found to achieve more than a four fold reduction in the majority of the examples used. Combining the state compression and the OBDD state store gave reductions in the memory requirements over both the cases where compression and a hash table is used and an OBDD state store without compression (table 3). Lastly, the OBDD state store (with state compression) was combined with a partial order reduction method. Although the partial order rules did reduce the number of reachable states, the expected memory reductions did not occur. Due to the removal of certain interleavings during partial order reductions, and therefore certain previously reachable states, the size of the OBDD state store were not reduced as much as before by the three OBDD transformation rules. Therefore, when reducing the state space according to partial order rules, it is in general more space and time efficient to use a hash table state store (see table 4).

In summary, an OBDD state store must in general be used with some form of compression, even probabilistic compression, if it is to be a worthwhile mechanism for space saving during on-the-fly model checking. The only type of model for which this prerequisite do not hold is models where the number of reachable states is close to the number of potentially reachable states 2^n where n is the length of the state vector. Furthermore, the state compression method introduced in section 5, is safe (i.e. all reachable states will be visited) and will in most cases reduce memory requirements no matter what state storage mechanism is used. Again, the only type of model for which the compression will not cause a memory saving, are those that reach a large portion of its potentially reachable states.

Before this experiment was started the view was expressed that combining an OBDD state store with the SPIN tool, will not be practical. A combinatorial explosion in the size of the OBDD was expected. It is our opinion that this view has been dispelled, since a number of real-life examples were chosen and for all of them a space saving was achieved over the traditional hash table storage method. It must be said that a substantial time increase is incurred, but this in part due to the way the operations on the OBDDs were implemented. The operations are by no means time efficient since this was meant to be an experiment in memory reduction. Furthermore, no mention is made of the performance of other variable orderings, but this is again due to the fact that we were interested in the general behaviour of an OBDD state store and were therefore not concerned about fine-tuning the method to obtain better performance. Improving the performance of the three OBDD functions used and experimenting with heuristics for finding better variable orderings will be the focus of future work.

Another future extension is to use ordered n-ary decision diagrams (ONDDs)[18] instead of OBDDs. The main difference between ONDDs and OBDDs is that an ONDD node is n-ary, where n is the cardinality of the domain of the variable it is labelled with. Every node in an ONDD can therefore have

n possible arcs to the nodes below it in the graph. Since it is possible that many of the arcs from a node can go to the same node, we will adopt the compaction described in [18] where every arc is labelled by a range of constants rather than with an individual constant. In figure 11 the ONDD encoding the reachable states in the mutual exclusion example is shown. Note that the state compression scheme of section 5 will fit in nicely with the use of an ONDD state store. For instance, every component m_i of the compressed state (see figure 9) will be one of the variables of the ONDD. Furthermore, if the indexes of the tables t_i is allocated consecutively (as is the case in all the examples in this paper), then the chances of compacting the arcs from a node by labelling it with ranges is increased.

References

[1] O. Bernholtz, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *CAV '94 : 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[2] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for ctl*. In *Symposium on Logic in Computer Science*, June 1995.

[3] Randall E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[4] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, 1993.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[8] W. Fouché and P. de Villiers. A Reusable Kernel for the Development of Control Software. In *6-th Southern African Computer Symposium*, pages 83–94, July 1991.

[9] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[10] P. Godefroid, G.J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Computer Aided Verification Workshop*, Montreal, Canada, June 1992. also in: Formal Methods in System Design, Kluwer, 1995.

[11] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[12] G.J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice & Experience*, 18(2):137–161, February 1988.

[13] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[14] G.J. Holzmann. An Analysis of Bitstate Hashing. In *15-th International Symposium on Protocol Specification, Testing, and Verification*, Warsaw,Poland, June 1995. North-Holland.

[15] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.

[16] K.L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, May 1992.

[17] Doron Peled. Combining partial order reductions with on-the-fly model checking. In *CAV '94 : 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[18] A. Rauzy. Toupie = μ-calculus + constraints. In *CAV '95 : 5th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, 1995.

[19] U. Stern and David Dill. Improved probilistic verification by hash compaction. In *CHARME '95 : Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, 1995.

[20] A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of the Second International Conference for Computer-Aided Verification.* Lecture Notes in Computer Science, 531, June 1990.

[21] M.Y. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *First Symposium on Logic in Computer Science*, pages 322–331, June 1986.

[22] W.C. Visser. A Run-Time Environment for a Validation Language. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1993.

[23] P. Wolper. On the relation of programs and computations to models of temoral logic. In *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, 1987.

[24] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *CAV '93 : 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.