Modelling Message Buffers with Binary Decision Diagrams

Bernd–Holger Schlingloff
Universität Bremen, TZI-BISS

hs@tzi.de

0 Introduction

Binary decision diagrams (BDDs, [Bry92]) have been recognized as an extremely efficient data structure for the representation of transition relations in the verification of finite-state reactive systems. With BDDs, it is possible to represent relations over domains with more than 2^{100} elements ([BCDM91]), provided the represented relation is well-structured. Asynchronous parallel systems such as communication protocols often use implicit or explicit buffering of messages which are sent between the processes. In these notes, we analyze the complexity of various possibilities to model the transition relation of a bounded buffer with BDDs, and discuss alternative approaches to this problem.

1 Binary Decision Diagrams

To make these notes self-contained, we quickly describe the symbolic representation of sets and relations with BDDs. For a detailed survey, the reader is referred to [Bry92]. Consider a sequence of variables $V \triangleq (v_1, ..., v_k)$ over domains $(D_1, ..., D_k)$, where each D_i is finite. An *ordered decision diagram* (ODD) or *deterministic branching program* for V is a tuple $(N, \mathcal{L}, E, n_0)$, where

- N is a finite set of nodes,
- $\mathcal{L} : N \to V \cup \{\top, \bot\}$ is a labelling of nodes,
- $E \subset N \times D \times N$ is a set of edges ($D = \bigcup_i D_i$), and
- n_0 is the initial node.

The following conditions are imposed:

- E is functional on D_i: If $\mathcal{L}(n) = v_i$, then for each $(n, d, n') \in E$ it holds that $d \in D_i$, and for each $d \in D_i$ there is exactly one n_d such that $(n, d, n_d) \in E$, and
- E is acyclic: If $(n, d, n') \in E$ with $\mathcal{L}(n) = v_i$ and $\mathcal{L}(n') = v_j$, then $i < j$.

It is easy to see that this definition is equivalent to the one given, e.g., in [Bry92]. Any ODD accepts (defines) a subset of $(D_1 \times ... \times D_k)$ via the following definition:

$$(N, \mathcal{L}, E, n_0) \models (d_1, ..., d_k) \quad \text{if} \quad (N, \mathcal{L}, E, n_0) \models_1 (d_1, ..., d_k).$$

In this definition, the notion $\models_m$ is declared by:

$(N, \mathcal{L}, E, n) \models_m (d_1, ..., d_k)$ if

- $\mathcal{L}(n) = \top$, or
- $\mathcal{L}(n) = v_i$ and $m < i$ and $(N, \mathcal{L}, E, n) \models_{m+1} (d_1, ..., d_k)$, or
- $\mathcal{L}(n) = v_i$ and $m = i$ and $(n, d_m, n') \in E$ and $(N, \mathcal{L}, E, n') \models_{m+1} (d_1, ..., d_k)$.

In other words, given a specific tuple, it can be determined whether it belongs to the set represented by an ODD by traversing its edges according to the components of the tuple.

When drawing ODDs, we usually omit the node labelled $\bot$ and all edges leading to it. For example, the ODD with two variables v, v' over $D_1 = D_2 = \{a, b, c, d\}$ given in Figure 1 below represents the set of tuples $\{(a, a), (a, b), (a, c), (a, d), (b, b), (b, d), (c, c), (c, d), (d, a), (d, d)\}$. *Binary* decision diagrams (BDDs) are ODDs where all domains are $\{0, 1\}$. Given any ODD, there exists a BDD of the same order of size which represents the same set: Choose any binary encoding of the domains, and replace each m-ary branch by a $\log m$-depth binary decision tree. Thus, in practice only BDDs are used; ODDs can be understood as abbreviations of the respective binary encoded BDDs. For example, choosing the encoding $a \mapsto 00$, $b \mapsto 10$, $c \mapsto 01$, and $d \mapsto 11$, the BDD given in the right half of Figure 1 represents the same set as the respective ODD on its left.

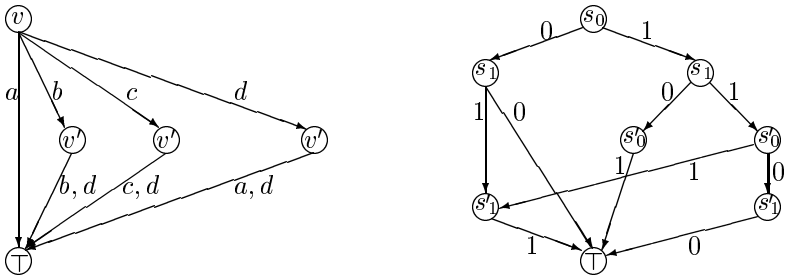

Figure 1: An ordered decision diagram and its binary encoding

The size of an ODD is the number of nodes it consists of. For a given ordering of the domains, and any set of values, there is a unique minimal ODD representing this set of values. The size of this minimal representation is not dependent on the size, but only on the structure of the represented set of values. E.g., the empty set and the set of all tuples both have an ODD representation of size one.

As another example, consider the elementship relation between a set $S = \{a, b, c\}$ and its powerset 2^S. The table and BDD are given in Figure 2. As can be seen, the table has no "regular" structure, thus both table and BDD are of order $S \cdot 2^S$. If we choose a different encoding as shown in Figure 3, the BDD representation exploits the fact that the matrix can be decomposed into isomorphic and constant submatrices.

Given a process P with state space D. Then the *transition relation* of P is a subset of $D \times D$. If P consists of k parallel processes $P_1, ..., P_k$ with state spaces $D_1, ..., D_k$, then the global state space of P is $D_1 \times \cdots \times D_k$. Therefore the transition relation can be described by $2k$ variables $s_1, ..., s_k, s'_1, ..., s'_k$, where s_i and s'_i are over domain D_i and describe the current and next state of process P_i. Again, if each D_i has up to m states, the global transition relation has up

$s_2s_3s_4$	000	001	010	011	100	101	110	111
s_0s_1	$\{\}$	$\{a\}$	$\{b\}$	$\{c\}$	$\{ab\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
$00(a)$		x			x	x		x
$01(b)$			x		x		x	x
$10(c)$				x		x	x	x

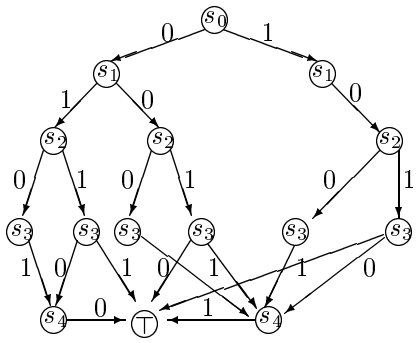

Figure 2: Power set relation and corresponding BDD

$s_2s_3s_4$	000	001	010	011	100	101	110	111
s_0s_1	$\{\}$	$\{a\}$	$\{b\}$	$\{a,b\}$	$\{c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
$00(a)$		x		x		x		x
$01(b)$			x	x			x	x
$10(c)$					x	x	x	x

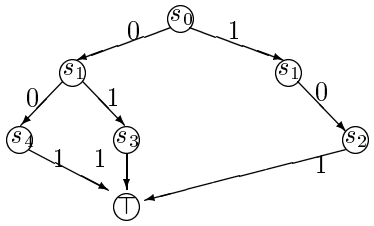

Figure 3: Power set relation and BDD with different encoding

$s'_0 s'_1$	a	b	c	d
	s_{00}	s_{01}	s_{00}	s_{01}
$s_0 s_1$	s_{10}	s_{10}	s_{11}	s_{11}
$a = s_{00}s_{10}$	x	x	x	x
$b = s_{01}s_{10}$		x		x
$c = s_{00}s_{11}$			x	x
$d = s_{01}s_{11}$	x			x

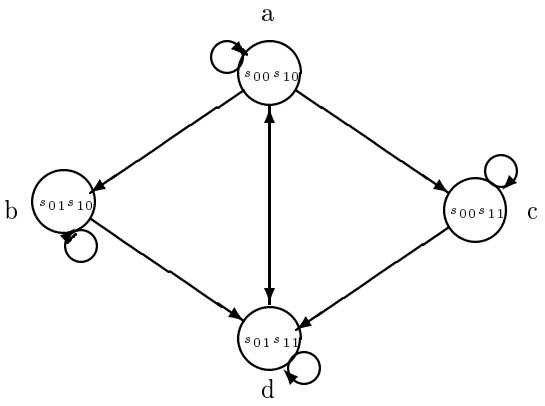

Figure 4: Matrix and graph of the encoded relation

to m^{2k} elements and can be described by a BDD over $2k \cdot \lceil \log |m| \rceil$ boolean variables. For example, consider the elementary net of Figure 5; it models two processes synchronizing on a common transition. The states of the first process are $D_0 = \{s_{00}, s_{01}\}$, the states of the second are $D_1 = \{s_{10}, s_{11}\}$. Since these domains are binary, we can use boolean variables s_0, s_1, s'_0, s'_1 to describe the current and next state of the processes. The global states are $a \triangleq (s_{00}, s_{10})$, $b \triangleq (s_{01}, s_{10})$, $c \triangleq (s_{00}, s_{11})$, and $d \triangleq (s_{01}, s_{11})$. In state d, either both processes idle or both processes synchronize and go to state a; in each other state, process P_i can either idle or make a step from s_{i0} to s_{i1}, independently of the other process. The transition relation of this system is the one represented by our example.

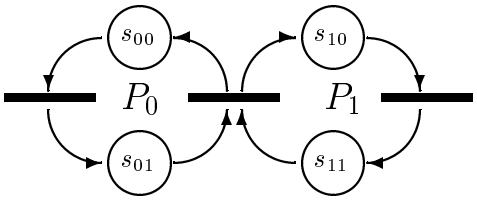

Figure 5: An elementary net model of synchronization

The set of *reachable states* of a system is the image set of the initial state(s) under the reflexive transitive closure of the transition relation. With BDDs, the transitive closure of a relation usually is calculated as the smallest fixed point of the recursive equation $R^* = \mathrm{I} \cup R \circ R^*$. Relational composition is calculated by the definition $xR \circ Sy$ iff $\exists z(xRz \wedge zRy)$, and existential quantification over

finite (binary) domains is replaced by a disjunction of the possible values of the domain.

Therefore, to calculate the set of reachable states with BDDs it is necessary to represent the complete transition relation. Since BDDs are graphs with a nonlocal connection structure, usually it is not possible to use virtual storage for BDD nodes; present technology limits the number of BDD nodes representing a transition function to approx. 10^6. The size of the BDD representation of the reachable states or reflexive transitive closure of a relation is often totally unrelated to the size of the representation of the relation itself; in our example, the transitive closure is the universal relation, and thus all states are reachable, with a BDD representation of size 1.

However, the size of a BDD crucially depends on the number and ordering of variables. In our example, consider the two processes as producer and consumer of messages which are passed at the synchronization step via handshake. That is, each process has an additional variable, m_0 and m_1, which are both over a domain $\mathcal{M}$ of, e.g., 4 messages $\{nil, x_1, x_2, x_3\}$. Process P_0 produces a message, i.e. sets variable m_0 to an arbitrary non-nil value, in the transition from s_{00} to s_{01}. On transition from (s_{01}, s_{11}) to (s_{00}, s_{10}) the value of m_0 is transferred to m_1, and m_0 is reset to nil. Process P_1 consumes (resets) variable m_1 in the transition from s_{10} to s_{11}. On idling transitions, the value of the message-variables is stable. The SMV-code (for SMV, see [McM93]) for this system is given in Figure 6, and the resulting BDD for variable ordering $(s_0, s'_0, s_1, s'_1, m_0, m'_0, m_1, m'_1)$ is shown in Figure 7.

```
MODULE main
VAR s0 : boolean; s1 : boolean; m0 : {nil,x1,x2,x3}; m1 : {nil,x1,x2,x3};
INIT    (s0 = 0 & s1 = 0)
TRANS   (s0 = 0 & s1 = 1 -> next(s1) = 1)
&       (s0 = 1 & s1 = 0 -> next(s0) = 1)
&       (s0 = 1 & s1 = 1 -> next(s0) = 0 & next(s1) = 0 |
                                next(s0) = 1 & next(s1) = 1)
& (s0 = 0 & next(s0) = 1 -> next(m0) in {x1,x2,x3})          -- produce
& (s0 = 1 & next(s0) = 0 -> next(m0) = nil)                  -- reset
& (s0 = next(s0)         -> next(m0) = m0)                   -- stable
& (s1 = 1 & next(s1) = 0 -> next(m1) = m0)                   -- transfer
& (s1 = 0 & next(s1) = 1 -> next(m1) = nil)                  -- consume
& (s1 = next(s1)         -> next(m1) = m1)                   -- stable
```

Figure 6: SMV-code for message passing between two processes

As can be seen, the size of this BDD is linear in the number $m \triangleq |\mathcal{M}|$ of possible messages. In this example, the linear complexity is caused only by "local diamonds", i.e., nodes branching into m successor nodes, which again join into one successor. This structure arises by the copying instructions `next(m0)=m0`, `next(m1)=m1` and `next(m1)=m0`. Variables m_0 and m_1 can be seen as consisting of w boolean variables $m_{01}...m_{0w}$ and $m_{11}...m_{1w}$, where $w \triangleq \lceil \log m \rceil$ is the *message width*. If we interleave the order of these variables, i.e., use variable ordering (m_{01}, m'_{01}, m_{11}, m'_{11},..., m_{0w}, m'_{0w}, m_{1w}, m'_{1w}), local diamonds are represented with complexity linear in w, see Figure 8. Thus, for the ordering (s_0, s'_0, s_1, s'_1, m_{01}, m'_{01}, m_{11}, m'_{11},..., m_{0w}, m'_{0w}, m_{1w}, m'_{1w}), the BDDs for the above SMV-code are logarithmic in m.

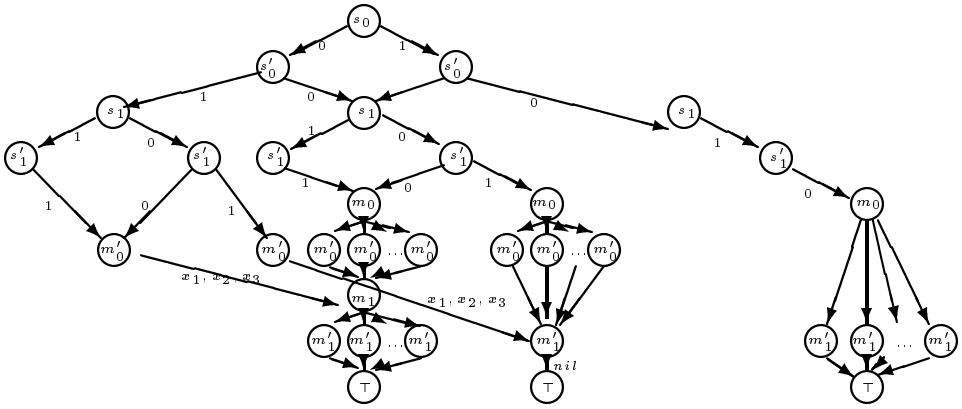

Figure 7: BDD for synchronous message passing

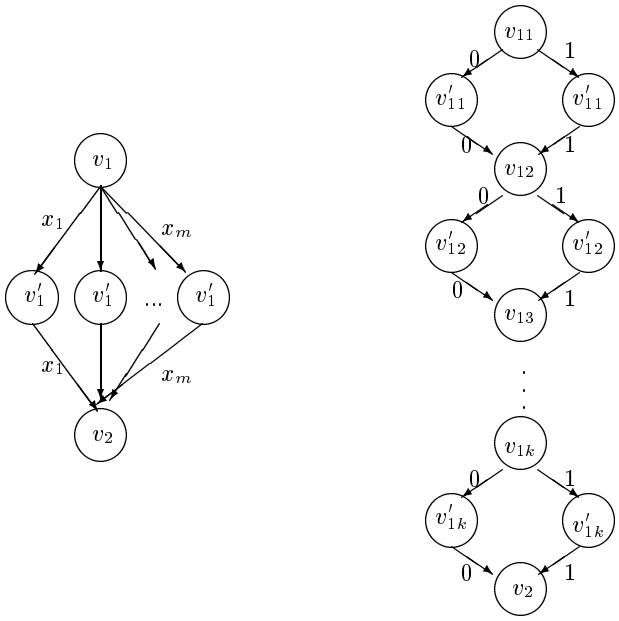

Figure 8: Interleaved encoding of a local diamond

2 Modelling of Message Buffers

Distributed parallel processes often use asynchronous (buffered) communication. Asynchronous message passing can be modelled with global variables by introducing a separate buffer process for each communication line. In many systems, the amount of messages which can be buffered is finite; in such systems buffer overflow often indicates erroneous behaviour of the system. For a fixed message alphabet $\mathcal{M} \triangleq \{nil, x_1, ..., x_{m-1}\}$, the formal specification of a bounded buffer of length n with input and output variables i and o over $\mathcal{M}$ is given in Table 1 on page 7.

i	b	o		i'	b'	o'
nil	$\langle\rangle$	nil			$\langle\rangle$	nil
x	$\langle\rangle$	nil		nil	$\langle\rangle$	x
nil	$\langle x_1, ..., x_\nu\rangle$	nil			$\langle x_1, ..., x_{\nu-1}\rangle$	x_ν
x	$\langle x_1, ..., x_\nu\rangle$	nil		nil	$\langle x, x_1, ..., x_{\nu-1}\rangle$	x_ν
nil	$\langle\rangle$	y			$\langle\rangle$	
x	$\langle\rangle$	y		nil	$\langle x\rangle$	
nil	$\langle x_1, ..., x_\nu\rangle$	y			$\langle x_1, ..., x_\nu\rangle$	
x	$\langle x_1, ..., x_\nu\rangle$	y	$(\nu < n)$	nil	$\langle x, x_1, ..., x_\nu\rangle$	
x	$\langle x_1, ..., x_n\rangle$	y		x	$\langle x_1, ..., x_n\rangle$	

Table 1: Specification of the transition relation of a bounded buffer

In the right half of this table, an empty entry means that the respective variable is set by the environment. An input value of nil in i indicates that there is no message to be sent; in this case the next value of i is determined by the producer. If this process has put a non-nil value $x \in \mathcal{M}$ into i, then this value is appended to the buffer, and i is reset to nil. The last line indicates a condition of buffer overflow: If a message is to be sent with the message buffer already filled, i remains stable. If the output variable o is nil and there is a message to deliver, it is copied into o. The consumer receives a message y from o by resetting o to nil.

The content of the buffer b is given as a sequence $\langle x_1, ..., x_\nu\rangle$ of messages, where $\langle\rangle$ denotes the empty buffer. There are various possibilities to implement sequences of messages with BDDs. The most obvious choice is to use n variables $b_1, ..., b_n$ over $\mathcal{M}$, such that b_1 contains the front element of the message queue, and incoming messages are appended into the smallest b_ν which is empty (contains nil as value). The necessary assignment operation for this modelling is given in Figure 9.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : b[j+1];
  !(i=nil) & !(o=nil) : if !(b[j-1]=nil) & b[j]=nil then i
                        else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j]=nil then nil
                        else if b[j+1]=nil then i
                        else b[j+1] fi fi;
  esac;
```

Figure 9: Bottom-version of buffer slot assignment

In this modelling, we rely on the fact that whenever $b_j = nil$, then for all $k \geq j$, also $b_k = nil$. This assumption only holds for the reachable states of a buffer which is initially empty; there are many transitions from illegal, i.e., nonreachable states to other illegal states in this model. In an explicit representation of the transition relation, one should try to avoid these redundant entrys. With BDDs, however, even though the size of the transition relation is much bigger than the transition relation restricted to the reachable states, its representation is much smaller. Since the value of each buffer slot depends only on its immediate neighbours, in fact the size of the representation is linear in the number of slots.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : if (b[j-1]=nil) then nil else b[j];
  !(i=nil) & !(o=nil) : if (b[1]=nil) then b[j+1] else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j]=nil then nil else b[j+1] fi;
  esac;
```

Figure 10: Top-version of buffer slot assignment

In the above implementation, the buffer content is shifted upon output. We refer to this modelling as the *bottom* version, because sent messages can be imagined to "sink to the ground". A dual implementation of the buffer shifts down the content one slot whenever an input is performed, and inserts the new element into the topmost slot b_n. Consequently, we call this modelling, where messages "float to the surface", the *top-version* of a bounded buffer. To perform an output in this version, the content of the lowest non-nil slot is copied into the output variable o. The respective code segment is given in Figure 10.

A third possibility is to use a *circular* implementation of the buffer: On input, the value of the input variable is copied into slot b_i, where $b_i = nil$ and $b_{i-1} \neq nil$; on output, o is set to b_j, where $b_j \neq nil$ and $b_{j-1} = nil$. To be able to distinguish between first and last element of the queue in this version, we have to make sure that there is at least one slot with content nil; therefore there has to be one more place than the actual capacity of the buffer. In the assignment clause in Figure 11, subtraction and addition of one is to be understood modulo n.

```
next(b[j]) := case
   (i=nil) & !(o=nil) : b[j];
   (i=nil) &  (o=nil) : if b[j-1]=nil then nil else b[j];
  !(i=nil) & !(o=nil) : if !(b[j-1]=nil) & b[j]=nil & b[j+1]=nil
                        then i else b[j] fi;
  !(i=nil) &  (o=nil) : if b[j-1]=nil then nil
                        else if b[j]=nil then i else b[j] fi;
  esac;
```

Figure 11: Circular version of buffer slot assignment

An alternative to the use of an empty slot would be to introduce queue-pointers for the position of the first and/or last element of the queue; this idea can be applied to all three of the above modellings. However, these alternative

versions turn out to be worse than the direct encoding via *nil*-test which is given above. In general, the queue-pointers would be functionally dependent of the content of the buffer; such functional dependencies can blow up the BDD size significantly ([HD93]).

Similarly, we can introduce additional BDD-variables indicating whether the buffer is empty or full; however, these variables tend to increase the size of the representation by a linear factor and usually can be replaced by appropriate boolean macro definitions. On the other hand, such variables can be important if the BDD is represented as a conjunction of partitioned transition relations, see [BCL91].

Finally, it is not always advisable to test whether a slot b_i contains the value *nil* by the test `b[i]=nil`. As we will see in the next section, it can be better to increase the message width w by one, such that the first bit of each message is a kind of checksum, indicating whether this message is *nil* or not.

3 Complexity Considerations

The BDD for the bottom version of a buffer of size n consists of two parts, one for the case that the buffer content remains stable, and one for the case that the buffer content is shifted down by one slot. The first part consists of a sequence of local diamonds for each slot, similar as in the example above. The BDD for the second part is depicted in Figure 12 for the special case $n = 2$ and $\mathcal{M} = \{x_1, x_2, x_3\}$.

As can be seen, for a new buffer slot b_{n+1}, $O(m^2)$ nodes are added to the BDD for a buffer of length n. Therefore the representation is of order $n \cdot m^2$, i.e., linear in the length and exponential in the width of the buffer. Since the transition relation is "almost" a function, a matrix representation would require $O(m^n)$ entrys, whereas a boolean algebra or programming language representation such as the SMV code above, is of order $m + n$ (or even constant, if array subscripts are allowed).

For the top version, the complexity of the representation is comparable to the bottom version. In the circular version, b'_n depends on b_n, i, b_{n-1}, and on b_1. This non-local dependency causes a blowup of factor 2, since the emptyness of b_1 has to be decided while testing b'_n. Moreover, to test whether a buffer is full or not we have to test whether any two adjacent slots are nil. This nonlocal test again blows up the complexity of this modelling.

As was to be expected, the number of reachable states is identical in the bottom and top modellings; of course, this number is exponential in the length of the buffer. For the circular implementation, the number of reachable states is approximately m times as much, since it contains an additional slot.

Table 2 summarizes the size of the BDDs of the transition relation for $m = 4$ (i.e., $w = 2$), and order $i < b_n < ... < b_1 < o$. All results were obtained with the public-domain SMV system; other BDD-based verification tools yield similar results. The buffers were embedded in a simple producer-consumer environment, where the producer and consumer are asynchronous, and the message to be sent or received does not depend upon or influence the state of the sender or receiver, respectively.

In this example, the size of the representation of the set of reachable states was of the same order of magnitude as the representation of the transition relation. Some considerations about this size are given below.

A critical factor in our approach is the message width w. As indicated in Table 2, e.g. the bottom implementation of a buffer of length 5 and width 2 has size 1458. For $w = 3$, this size is 11774, and for $w = 4$, it is 108357. In [BS90]

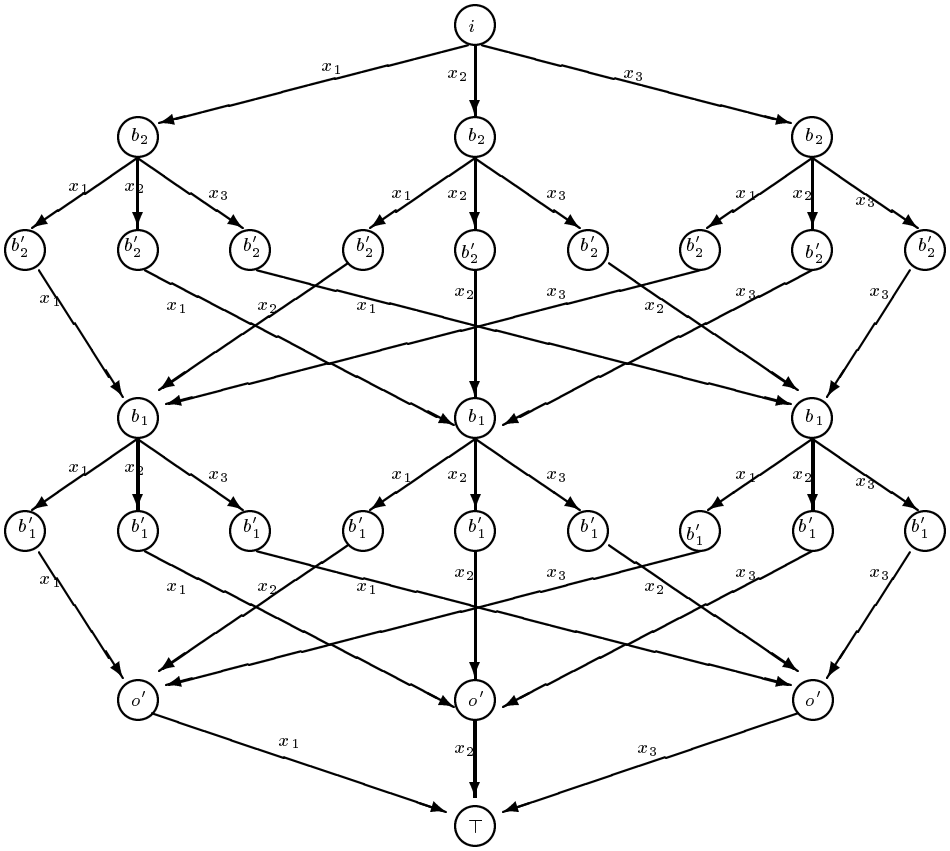

Figure 12: BDD for shifting down the buffer content

it is proved that for any finite function, a BDD of polynomial size exists iff the function can be realized by a polynomially bounded depth circuit. For message buffer, certainly the transition function can be realized by such a circuit; thus there exists a BDD which is polynomial both in n and w.

If there is no constraint on the order of variables, then such a BDD can be constructed by interleaving the bits of all slots: Let $i = i_1...i_w$, $b_j = b_{j1}...b_{jw}$, and $o = o_1...o_w$. Then for each $k \leq w$, $(i_k, b_{nk}, ..., b_{1k}, o_k)$ can be regarded as a buffer of width 1. The only "nonlocal test" in this buffer of length 1 is whether some slot b_j is empty: if this is determined by comparison of b_{j1} and ... and b_{jn}, then we still have an exponential growth. If we introduce additional bits $(i_0, b_{n0}, ..., b_{10}, o_0)$ which are 0 iff the corresponding message is nil, then each bit-slice is linear in the length of the buffer. For the order $i_0 < b_{n0} < ... < b_{10} < o_0 < i_1 < b_{n1} < ... < b_{11} < o_1 < ... < i_w < b_{nw} < ... < b_{1w} < o_w$, these small BDDs

length	3	5	7	9	11
bottom	714	1458	2204	2950	3696
top	599	1113	1627	2141	2655
circular	1038	2350	3999	5307	6833
reach	1400	12740	2^{16}	2^{20}	2^{23}

Table 2: BDD size of transition relation and reachable state set

are simply added, and the overall complexity is $O(w \cdot n)$.

Unfortunately, in many practical examples it is not possible to choose such a bitwise interleaved order. Usually, the input and output variables are imported from other processes, and their order cannot be chosen arbitrarily. An argument similar to the one from Section 1 on page 5 shows that for any order, in which i is before all buffer bits, the representation is exponential in w. Therefore, in practical verification, w should be kept as small as possible. There are several ways to do so:

- For every channel, define a separate message alphabet;
- replace a parametrized message $m(t)$ with $t \in \{t_1, ..., t_k\}$ by a list of messages m_{t1}, ..., m_{tk};
- replace a compound message by a sequence of messages, and
- abstract several different messages into one.

When using the latter two methods, one has to be careful to preserve the semantics of the original model ([CGL92]). Using these techniques, we have been able to verify systems with up to 2^7 different messages.

4 Alternative Approaches

In [GL96],[BG96] it is suggested to extend the BDD data structure for the representation of message buffers. The new data structures are called QBDDs and QDDs, respectively. The basic idea is to replace the consecutive testing of buffer variables by a repeated test of one and only one variable. Therefore, the representation of the transition relation is independent of the buffer size. Moreover, even systems of which the maximum amount of buffer space is not statically known can be verified.

However, as we have shown above, the (static) length of a buffer may not be the most important factor in the representation of the transition relation. Moreover, "buffer overflow" errors in the system can only be detected with a bounded buffer. Even worse, in systems on which a full buffer forces delay of the sender, with QBDDs we have to introduce an additional counter variable. For these type of systems, BDDs seem to be more adequate than QBDDs or QDDs.

Being able to represent the transition relation is only a necessary prerequisite for the verification of a system. Equally important is the size of the representation of the *reachable states* $\mathcal{R}$ of the system. Unfortunately, the size of the BDD for $\mathcal{R}$ has no predictable connection to the size of the BDD for the transition relation.

In many systems both the number of reachable states and its representation are linear in the number of iteration steps of the model, iff the system is correct. This is due to the fact that on reachable states, the transition relation is "almost" functional, yielding either a single or a small number of successor states. On the other hand, from an "impossible" state usually many other "impossible" states are reachable. A drastic example is Valmari's elevator for which the reachable state set (in any representation) explodes as the elevator breaks through the ceiling and skyrockets into the air. Thus an exponential increase in (the representation of) $\mathcal{R}$ after some number of steps almost certainly indicates an error.

In [GL96] it is claimed that "there are cases where the QBDD representation is strictly more concise than the BDD representation". Assume our buffer in a

context where the producer sends one fixed sequence of messages x_1, x_2, ... x_ν. That is, the reachable buffer content is $\{\langle\rangle, \langle x_1\rangle, \langle x_2 x_1\rangle, ..., \langle x_\nu ... x_2 x_1\rangle, \langle x_2\rangle, ... \langle x_\nu ... x_2\rangle, ..., \langle x_\nu\rangle\}$. With the top- and bottom version of the buffer, the representation of this set is quadratic in ν, whereas with the circular representation and also with QBDDs it is linear in ν.

On the other hand, consider the case that the producer can send an arbitrary sequence of messages. In this case, the top- and bottom-versions are of constant size, whereas the QBDD implementation is linear in the number of sent messages.

In practical examples, such extreme cases are rare. In our experiments, we have found no significant difference in the size of the reachability sets of the various alternatives. The number of parallel processes and their relative order has a much bigger impact on the size of the BDD for $\mathcal{R}$ than the actual implementation of the buffer. Typically we can handle systems of up to 5 processes, each with approx. $2^4 - 2^5$ local states, where each process is equipped with a buffer of $n, w \leq 5$. However, there still is a need for heuristics which use dependencies between the processes to obtain a "good" order for the process state variables.

An important observation is that the content of a message buffer used to coordinate processes shows regular patterns, which also depend on the state of the processes. E.g., in a certain process state the buffer might always contain only copies of two different messages from $\mathcal{M}$. As another example, some specific message might always be followed by some other specific message in the buffer. Currently we are investigating methods how these regularities can be exploited to further reduce the size of the representation of the reachability set.

References

[BCDM91] J. Burch, E. M. Clarke, D. Dill, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In 5^{th} *IEEE LICS*, June 1991.

[BCL91] J. Burch, E. M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *Proc. IFIP Conf. on VLSI*, Edinburgh, August 1991.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of* 5^{th} *CAV*, New Brunswick, July 1996.

[Bry92] R. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Comp. Surv.*, Vol 24, No 3:293–318, 1992.

[BS90] R. Boppana and M. Sipser. The complexity of finite functions. In J van Leeuwen, editor, *Handbook of theoretical computer science, Vol. A*, chapter 14, pages 757–805. Elsevier, Amsterdam, 1990.

[CGL92] E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In 19^{th} *ACM POPL*, January 1992.

[GL96] P. Godefroid and D. Long. Symbolic protocol verification with queue BDDs. In *Proceedings of IEEE LICS*, New Brunswick, July 1996.

[HD93] A. Hu and D. Dill. Reducing BDD size by exploiting functional dependencies. In *Proc.* 30^{th} *ACM/IEEE DAC*, 1993.

[McM93] K. McMillan. *Symbolic model checking.* Kluwer, 1993.