Formal Methods in System Design, , 1–17 (1997)

Symbolic Protocol Verification with Queue BDDs

PATRICE GODEFROID god@bell-labs.com
Bell Laboratories, Lucent Technologies, 1000 E. Warrenville Road, Naperville, IL 60566, U.S.A.

DAVID E. LONG long@bell-labs.com
Bell Laboratories, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

Abstract. Symbolic verification based on Binary Decision Diagrams (BDDs) has proven to be a powerful technique for ensuring the correctness of digital hardware. In contrast, BDDs have not caught on as widely for software verification, partly because the data types used in software are more complicated than those used in hardware. In this work, we propose an extension of BDDs for dealing with dynamic data structures. Specifically, we focus on queues, since they are commonly used in modeling communication protocols. We introduce Queue BDDs (QBDDs), which include all the power of BDDs while also providing an efficient representation of queue contents. Experimental results show that QBDDs are well-suited for the verification of communication protocols.

Keywords: communication protocols, queues, symbolic verification, BDDs, state explosion, state-space exploration, model checking

1. Introduction

Binary Decision Diagrams (BDDs) [5] have proven to be a powerful tool for the verification of digital hardware [7, 25]. The level of abstraction provided by BDDs closely matches that required for modeling hardware. In particular, the value in a latch is often represented by a single bit, corresponding to one BDD variable. For software verification, BDDs have not been as popular. Part of the difficulty is that the data types that must be manipulated for software verification are more complicated than those required for hardware. Variables with simple finite domains, such as fixed-width integers, can be encoded by a vector of booleans in a straightforward manner, but other data types, such as lists, stacks, and queues, are more awkward to deal with. Often, it is difficult to estimate the maximum size of such dynamic structures in order to determine a suitable encoding. Even if a size bound is available, it may be both inefficient and cumbersome to encode the possible values directly.

In this work, we consider one of the most frequent types of software systems encountered in verification: communication protocols. Protocols are hard to design because they usually involve several concurrent processes, which may interact in unexpected ways. Communication protocols are often modeled by a collection of finite state machines communicating by exchanging messages. Since the communication is asynchronous, message queues are used to provide buffering. In this paper, we study symbolic representations for such systems. We show how to extend BDDs with a natural mechanism for representing and reasoning about queue contents. The resulting symbolic representation is called a Queue BDD, or QBDD for short.

Other approaches to the state-explosion problem have been proposed. Examples include abstraction [11, 9, 12, 18, 23] and compositional reasoning [1, 10, 19, 24]. Our approach is orthogonal to these methods, in that we try to concisely represent the full state space of a given system. The system itself may be an abstracted representation or only part of a larger system. Symmetry reductions [8, 14, 22] and partial-order methods [17, 16, 28, 31] explore only parts of the full state space, while still preserving properties of interest. These approaches generally use an explicit representation of the state space. In contrast, our method explores the full space, but uses a symbolic representation.

In the next section, we introduce and precisely define QBDDs. (We assume that the reader is already familiar with BDDs and the algorithms associated with them.) In Section 3, we present algorithms for performing operations on QBDDs, such as computing the effect of enqueueing and dequeueing an element on a queue, and discuss their complexity. We then compare QBDDs with a standard BDD representation for systems containing queues. A symbolic verification system based on QBDDs has been implemented, and results of experiments performed on several existing communication protocols are reported in Section 5. The paper ends with a comparison between our contributions and other related work.

2. Representing Sets of Queue Contents

The value of a queue is a sequence of messages over an alphabet M of possible messages. If we assume that at any time the number of messages in the queue is at most b, then we can represent a set of values of the queue with a BDD involving $b\lceil\log_2(|M|+1)\rceil$ boolean variables. (We have $|M|$ messages, plus a special symbol ϵ representing the absence of a message.) One obvious disadvantage of this representation is that b must be known.

Another natural representation for a set of values for the queue is as a minimal deterministic finite automaton (DFA) over the alphabet M. The queue contents in the set correspond to the strings accepted by the DFA. Note that if the set of values is finite, then the DFA is both finite and acyclic. The DFA representation of queue-contents is referred to as a Queue-content Decision Diagram (QDD) in [4].

Compared to the BDD representation above, the DFA representation is potentially more concise for the following reason. Consider two sequences of messages w_1 and w_2 such that $w_1 w$ is in the set of values for the queue iff $w_2 w$ is also in the set. In the BDD case, the common set of suffixes w that are possible after either w_1 or w_2 will be represented using the same BDD nodes *only if* the lengths of w_1 and w_2 are equal. Otherwise, the boolean variables encoding the suffixes after w_1 will not be the same as the boolean variables encoding the suffixes after w_2, and hence these suffixes will be represented by different BDD nodes. In contrast, this wasteful duplication can be avoided by using the DFA representation described above. Whatever the lengths of w_1 and w_2, the DFA representation will reach the same state after reading either w_1 or w_2, and the representation of the set of suffixes will be unique. Another advantage of the DFA representation is that it does not depend on any predefined knowledge of the bound b.

Below, we introduce a new symbolic representation, called Queue BDD (QBDD), which combines the DFA representation (for the queues in a system) with the BDD representation (for the non-queue parts). The combined representation consists of a sequence of layers. Each layer is either a DFA (representing a queue) or a BDD (representing some non-queue variables). Each layer is joined to the one below it with arcs linking its "accepting" states to the "initial" states of the next one. (For a BDD, the accepting state is the 1 terminal, while the initial state is the root.) The combined representation accepts the encoding of a state of the protocol being verified if the values of the variables for that encoding define a path from the initial state of the first layer to an accepting state of the last layer.

Because queues are often used to pass the values of process variables from one process to another, it is convenient to avoid explicitly translating BDD variable values into queue messages. This can be done by encoding the messages in the DFA layers using BDD variables. For example, suppose that a queue is used to pass messages containing the value of the variable $v \in \{0, 1, 2, 3\}$, and that v is encoded with two BDD variables, v_1 and v_2. If we represent the messages in the queue using two BDD variables q_1 and q_2 with the same encoding used for v, then enqueueing the value of v will basically involve copying the values of v_1 and v_2 into q_1 and q_2 respectively. Similarly, dequeueing a message into v would just involve copying q_1 and q_2 into v_1 and v_2.

This idea brings us to a difficulty though. In a standard BDD, each variable can only occur once along any path from the root to a leaf, while the queue will generally contain multiple messages. Thus, in the previous example, to represent the entire contents of the queue, we must either make multiple pairs of variables (one pair per message in the queue), or we must relax this restriction. The former solution is equivalent to the "standard BDD case" discussed at the start of this section, and suffers from the drawbacks mentioned there. Hence we choose the latter solution, where successive occurrences of a queue variable represent successive messages in the queue.

As in the standard BDD encoding mentioned earlier, we need codes for all the possible messages in the queue, and one additional code ϵ that represents "no message", i.e., the end of the queue. To see why this is necessary, consider trying to represent the set of queue-contents $\{\epsilon, m_1 m_2\}$, where the queue is either empty or contains two messages, m_1 followed by m_2. From the initial node, there must be a path for m_1, as well as a path for the possibility that there is no first message. The latter is represented by the special ϵ value. This value corresponds to the notion of reaching an accepting state in the DFA representation of a queue.

There is one subtlety when encoding the DFA. If we are not careful, the BDD path compression rule can eliminate nodes so as to destroy the duplication of variables that is needed to distinguish successive messages in the queue. (The path compression rule eliminates nodes whose successors are identical.) To avoid this possibility, we encode the ϵ value in a special way. We create a new BDD variable q_ϵ, and assign the codes with $q_\epsilon = 1$ to the value ϵ and the codes with $q_\epsilon = 0$ to the other possible messages.

We now define QBDDs more precisely. For notational simplicity, we consider the case where there is a single queue. The general case is a straightforward extension. Consider a DAG whose non-leaf vertices are labeled with variables of two types: ordinary (non-queue) boolean variables V, and (boolean) queue variables Q. The leaves of the DAG are all either 0 or 1. Each non-leaf node has two outgoing edges, a 0-edge and a 1-edge. There is one distinguished queue variable denoted q_ϵ. (In the general case with several queues, each queue has a separate set Q of queue variables and a separate distinguished queue variable "q_ϵ".)

We define a semantic function Φ that takes as input valuations for the non-queue and queue variables and produces as output either 0 or 1. The valuation for the non-queue variables is given as a mapping $\sigma_V: V \rightarrow \{0, 1\}$. The valuation σ_Q for the queue variables is a bit more complicated, due to the need to represent successive messages in the queue. The inputs to σ_Q are a variable in Q, and a nonnegative integer. Conceptually, the integer will represent the position in the queue. A sequence of messages of length l is represented as follows. For the q_ϵ variable, $\sigma_Q(q_\epsilon, i) = 0$ for $i < l$, $\sigma_Q(q_\epsilon, l) = 1$, and $\sigma_Q(q_\epsilon, i)$ is undefined for $i > l$. For a queue variable $q \neq q_\epsilon$, $\sigma_Q(q, i)$ will be defined exactly when $1 \leq i \leq l$.

Let n be a non-leaf node of the DAG, and let $l(n)$ denote the variable labeling n, $s_0(n)$ be the 0-successor of n, and $s_1(n)$ be the 1-successor of n. We now define $\Phi(n, \sigma_V, \sigma_Q, i)$. For conciseness, we will write $\Phi(n)$ when the last three arguments are understood to be σ_V, σ_Q, and i.

1. $\Phi(0) = 0$ and $\Phi(1) = 1$.

2. For a non-leaf node n:

   (A) If $l(n) \in V$, then $\Phi(n) = \Phi(s_0(n))$ when $\sigma_V(l(n)) = 0$, and $\Phi(n) = \Phi(s_1(n))$ when $\sigma_V(l(n)) = 1$.

   (B) Suppose that $l(n)$ is equal to q_ϵ. If $\sigma_Q(q_\epsilon, i) = 0$, then define $\Phi(n) = \Phi(s_0(n), \sigma_V, \sigma_Q, i+1)$. If $\sigma_Q(q_\epsilon, i) = 1$, then $\Phi(n) = \Phi(s_1(n), \sigma_V, \sigma_Q, i+1)$. If $\sigma_Q(q_\epsilon, i)$ is undefined, then $\Phi(n)$ is undefined.

   (C) Suppose $l(n) = q \in Q$ but $q \neq q_\epsilon$. If $\sigma_Q(q, i) = 0$, then $\Phi(n) = \Phi(s_0(n))$. If $\sigma_Q(q, i) = 1$, then $\Phi(n) = \Phi(s_1(n))$. If $\sigma_Q(q, i)$ is undefined, then $\Phi(n)$ is undefined.

Notice that $\Phi(n, \sigma_V, \sigma_Q, 0)$ is not always defined for an arbitrary DAG. Suppose for example, that the DAG consists of one non-leaf node labeled with $q \neq q_\epsilon$, with leaves 0 and 1. The semantic function is not defined for this DAG when the valuation σ_Q represents the empty queue, i.e., $\sigma_Q(q_\epsilon, 0) = 1$, and σ_Q is undefined otherwise.

We will call a DAG with root node n an *unordered unreduced* QBDD if $\Phi(n, \sigma_V, \sigma_Q, 0)$ is always defined, regardless of σ_V and σ_Q. (Intuitively, this means that $\Phi(n, \sigma_V, \sigma_Q, 0)$ returns either 0 or 1 for any valuation, i.e., for the encoding of any possible state of the protocol.) This definition imposes one ordering requirement on QBDDs immediately. In particular, if a node involves a test on q_ϵ, then the sub-DAG pointed to by the 1-successor of that node cannot contain nodes labeled with q_ϵ or with any other queue variables for that queue. (In the general case with several queues, the semantic function Φ is defined with one valuation function σ_Q and integer i per queue.)

In order to obtain a canonical form, we impose additional ordering and reduction requirements in analogy to those for (reduced, ordered) BDDs. A linear ordering for the non-queue variables V is fixed, just as in the BDD case. The queue variables Q are also linearly ordered, with q_ϵ being first. A global ordering is constructed by inserting the entire set Q into some position in the ordering for V. The only difference from a complete global ordering is that any queue variable may be followed by a test on q_ϵ (except for the constraint on the 1-successor of a q_ϵ node, mentioned above). For notational simplicity, we define the 0 and 1 terminals as the last two elements in the ordering. We use the term "QBDD" to refer to a reduced ordered QBDD, i.e., a QBDD that satisfies the above ordering requirements and that is maximally reduced using the classical BDD reduction rules [5].

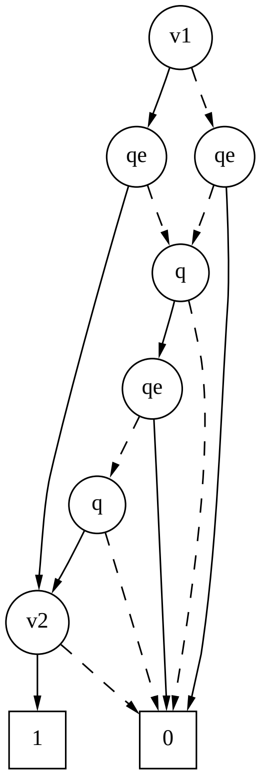


Figure 1. An example QBDD

An example QBDD is shown in Figure 1. In the figure, dashed lines denote 0-successors and solid lines denote 1-successors. We can view this QBDD as representing a state set over two boolean variables v_1 and v_2, and one queue. If we write states as vectors with the v_1 value first, followed by the queue contents and then the v_2 value, then the state set corresponding to the QBDD is

$$\{(1,\epsilon,1),(1,11\ldots,1),(0,11\ldots,1)\}$$

where the latter two tuples denote classes of states where the queue contains two 1's followed by anything. (Hence, there are an infinite number of states in the state set represented by this QBDD.)

3. Operations on QBDDs

Many algorithms for BDD operations, such as conjunction and disjunction, have counterparts for QBDDs. The differences between the corresponding QBDD and BDD algorithms are minor. The only significant change is that the QBDD algorithm must be more careful when comparing the level of variables, due to the repetition of variables that are in Q. New algorithms are needed for queue-specific operations on QBDDs, such as enqueueing and dequeueing. We now consider these new algorithms.

To simplify the discussion, we assume that all (non-queue) variables and expressions in the protocol description have boolean values. The allowed message types for the queue are also assumed to be boolean. In the presentation below, v will represent a boolean non-queue variable, e will represent a boolean expression involving non-queue variables, and q will be the (boolean) queue variable representing a message in the queue. Top(f) denotes the top variable in the BDD or QBDD f, i.e., the label of the root node of f. IfThenElse(v, r_1, r_0) denotes a non-leaf node n such that $l(n) = v$, $s_1(n) = r_1$, and $s_0(n) = r_0$.

Before presenting the enqueueing and dequeueing operations, we give two utility routines that return QBDDs that represent queues containing a specified element. The first routine is shown in Figure 2. The routine takes as input the BDD for an expression e. It returns the QBDD that evaluates to 1 when the queue contains exactly one element whose value is e. The second routine, AtLeastOneElementQueue (Figure 3), returns a QBDD that evaluates to 1 when the queue contains at least one element, and the head element is e. (For clarity, the figures do not show the manipulation of the result cache which is used in all operations to avoid exponential behavior. The manipulation is exactly analogous to that used in BDD routines.)

Now we consider the Dequeue algorithm. The algorithm takes as input a QBDD f which represents a set of states S. If in some states, the queue is empty, we mask out such states by conjoining with the QBDD IfThenElse(q_ϵ, 0, 1), which represents all states where the queue contains at least one element. If in addition, we wish to copy the element to be dequeued into a variable v, we first create a "post-dequeue" value for v, which we denote by v'. We then conjoin f with AtLeastOneElementQueue(IfThenElse(v', 1, 0)), existentially quantify the variable v, and rename v' back to v. This yields a QBDD representing the same states as S, but with the value of v in each state equal to the value at the head of the queue in that state.

```
QBDD OneElementQueue(BDD e)

if Top(e) > q then
      r_0 ← IfThenElse(q_ε, ¬e, 0)
      r_1 ← IfThenElse(q_ε, e, 0)
      return IfThenElse(q_ε, 0, IfThenElse(q, r_1, r_0))
else
      r_0 ← OneElementQueue(s_0(e))
      r_1 ← OneElementQueue(s_1(e))
      return IfThenElse(Top(e), r_1, r_0)
endif
```

Figure 2. Routine for representing single-element queues

```
QBDD AtLeastOneElementQueue(BDD e)

if Top(e) > q then
      return IfThenElse(q_ε, 0, IfThenElse(q, e, ¬e))
else
      r_0 ← AtLeastOneElementQueue(s_0(e))
      r_1 ← AtLeastOneElementQueue(s_1(e))
      return IfThenElse(Top(e), r_1, r_0)
endif
```

Figure 3. Routine for representing queues with a specified head

After these preprocessing steps, the Dequeue algorithm (Figure 4) removes the head element from the queue. This algorithm takes as input a QBDD f which represents a set of states S, in all of which the queue has at least one element. The return value is a QBDD representing the set of states resulting from removing the head of the queue in each state of S. Note that the algorithm is very similar to the method for existential quantification in the case of standard BDDs. The two operations also have the same complexity. In the general case where the contents of the queue are encoded by $k > 1$ boolean variables, the worst-case complexity of the algorithm is $O(|f|^{2^k})$. But like existential quantification, the algorithm almost always behaves reasonably in practice.

```
QBDD Dequeue(QBDD f)

if f = 0 then
    return f
else if Top(f) < q_ε then
    r_0 ← Dequeue(s_0(f))
    r_1 ← Dequeue(s_1(f))
    return IfThenElse(Top(f), r_1, r_0)
else
    — Top(f) must equal q_ε
    — s_1(f) must equal 0
    h ← s_0(f)
    if Top(h) = q then
        return s_0(h) ∨ s_1(h)
    else
        return h
    endif
endif
```

Figure 4. The Dequeue algorithm

We now consider the algorithm for enqueueing the value given by the expression e. The algorithm takes as input a QBDD f, representing a finite set of states, and the QBDD $h = \text{OneElementQueue}(e)$. (The infinite set of states where a queue has an arbitrary number of messages followed by a message e is not representable with QBDDs.) The enqueue operation is slightly more complicated than those considered previously because of the need to find the end position of the queue. The end of the queue is indicated by following the 1-successor of a q_ϵ node. When we encounter such a branch, the QBDD reached represents any remaining constraints on the non-queue part of the state. We conjoin these constraints with h to add the element e to the end of the queue. We then recursively enqueue e for the 0-successor of the q_ϵ node, and union the two state sets. The algorithm is shown in Figure 5. In the figure, the notation f_x denotes the cofactor of f with respect to the variable x. This is equal to the result of setting x to 1 in f. If $x = \text{Top}(f)$, then f_x is just $s_1(f)$, and if $x < \mathrm{Top}(f)$, then f_x is equal to f. (The case $x > \mathrm{Top}(f)$ never happens in the algorithm.) Similarly, $f_{\neg x}$ denotes the result of setting x to 0 in f.

```
QBDD Enqueue(QBDD f, QBDD h)
— Initially h = OneElementQueue(e)
— Recur through f and h

if Top(f) < Top(h) then
    x ← Top(f)
else
    x ← Top(h)
endif
if x < q_ϵ then
    r_0 ← Enqueue(f_¬x, h_¬x)
    r_1 ← Enqueue(f_x, h_x)
    return IfThenElse(x, r_1, r_0)
else
    — We are at q_ϵ in h
    — Recur through f
    if Top(f) > q then
        return IfThenElse(q_ϵ, 0, f)
    else if Top(f) = q_ϵ then
        r_0 ← IfThenElse(q_ϵ, 0, Enqueue(s_0(f), h))
        r_1 ← s_1(f) ∧ h
        return r_0 ∨ r_1        (*)
    else
        — Top(f) is q
        r_0 ← Enqueue(s_0(f), h)
        r_1 ← Enqueue(s_1(f), h)
        return IfThenElse(Top(f), r_1, r_0)
    endif
endif
```

Figure 5. The Enqueue algorithm

The worst-case complexity of the Enqueue algorithm is $O(|f||h|)$. This hinges on the fact that the disjunction performed on the line marked by (*) can be made a constant-time operation. Both operands of the disjunction will have top variable q_ϵ, with 1-successor 0. Traversing the 0-successor of r_0, we may pass through q, but we must eventually reach a node of the form IfThenElse(q_ϵ, 0, g_1). Similarly, if we traverse the 0-successor of r_1, then (after perhaps a q) we reach a node of the form IfThenElse(q_ϵ, g_2, 0). Thus during the disjunction of r_0 and r_1, we form IfThenElse(q_ϵ, g_2, g_1), so we do not compute $g_1 \vee g_2$.

Other operations, such as testing if a queue is empty or (in the case of bounded queues) full, can be done using logical operations. For example, testing for emptiness can be done by conjoining with the QBDD q_ϵ, representing the set of states where the queue is empty.

4. Comparison with BDDs

In this section, we compare QBDDs with a standard BDD representation for systems containing queues. First note that using the queue facilities in QBDDs constrains the representation of successive messages in the queue. To illustrate this, consider a queue whose messages are 32-bit integers, and suppose that the queue currently contains two messages, with the first message stored in the queue having any value, and the second message being equal to the first. Representing this constraint with a QBDD would require about 2^{32} nodes, because the entire first message is represented before any part of the second one. In contrast, the same constraint has linear size if we use a BDD-style representation and we order the variables so that the bits of the messages are interleaved. Thus, for some particular queues, it may be better to use a traditional BDD-style representation rather than the specialized form provided by the QBDDs. Since QBDDs subsume BDDs, we can always use the BDD-style representation where appropriate. Thus, the size of a QBDD need never be larger than the corresponding BDD representation of the same constraint. (Another possibility for situations like the above in the QBDD case is to split the single queue with 32-bit messages into 32 queues, each holding 1-bit messages. Each of these can be represented using the special queue facilities. In this case, the constraint mentioned above has linear size. This representation might be appropriate in situations where we need the interleaved ordering but have no bound on queue size.)

Conversely, suppose that in the BDD ordering, the variables for each message in the queue are together and that the group of variables for message m_{i+1} follows the group of variables for m_i. This ordering corresponds closely to that used for queues in QBDDs. With this ordering, there are cases where the QBDD representation is strictly more concise than the BDD representation. With queues of size k, the QBDD for a constraint may be up to k times smaller than the BDD. Consider for example, sending k distinct messages to a queue that may lose messages. For simplicity, we will number the messages 1 through k. In the QBDD case, the representation of the set of possible queue states after the k messages have been sent has size $O(k)$. A picture of the QBDD representation of this constraint is shown on the left side of Figure 6 for $k = 4$. For conciseness, only the nodes labeled with q_ϵ are shown in the figure. The q variables are indicated abstractly by the labels on the arcs. For example, an arc label of 2 means that the q variables along that arc encode message 2. The 1-successor of each q_ϵ node leads to the 1 leaf; these successors are also not shown. Note that all of the arcs with the same label lead to the same q_ϵ node. In contrast, the BDD for the same constraint has size $O(k^2)$, because this type of sharing is not possible. The ith message in the queue must be represented by the ith set of message variables. Hence, the ith set of variables will be used to encode suffixes beginning with $i, i+1, \ldots, k$. An abstracted view of the BDD for $k = 4$ is shown on the right side of Figure 6. We also note that the BDD can be no larger than k times the QBDD size. The example above demonstrates the worst-case increase for BDDs. In the translation from a QBDD to a BDD, a QBDD queue node which is reachable via m distinct paths, each passing through

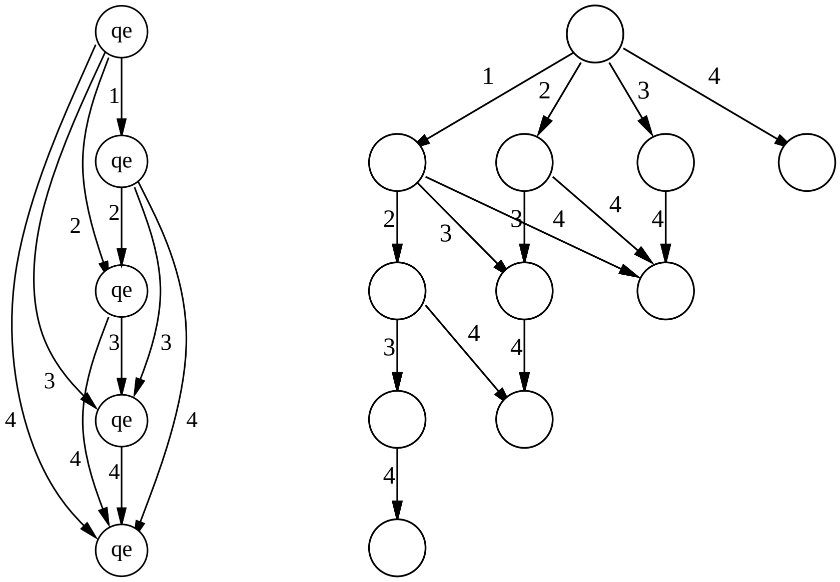


Figure 6. Representation of the contents of a lossy queue using QBBDs (left) and BDDs (right)

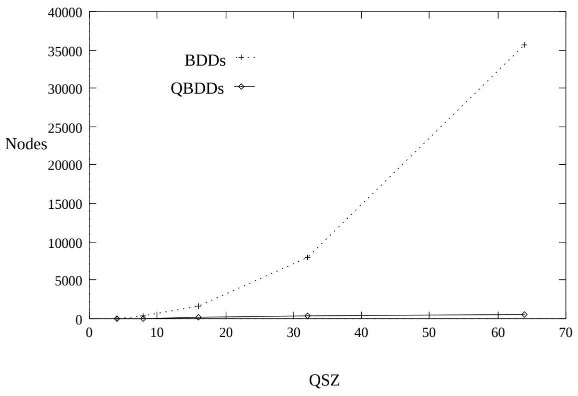
a different number of q_ϵ nodes, will be duplicated m times. Thus, with queues of length k, the maximum duplication for each node is k.

The BDD representation has one advantage when the enqueue and dequeue operations are implemented in terms of relational products. In this case, the reverse operations (e.g., getting all states where it is possible to do an enqueue operation and wind up in a state in some given set), which are used in a variety of symbolic state-space exploration methods, are simply a matter of doing relational products with the converse relations. With QBDDs, special-purpose code is needed for the reverse operations.

5. Experimental Results

We have implemented a symbolic verification system based on QBDDs. The input to the system is a program expressed in a guarded command language. The program describes a network of processes that run asynchronously and that communicate by exchanging messages. The verifier represents the transition system and state space using QBDDs. It checks invariants by doing a forward search using a modified breadth-first strategy [6]. The system could be extended to check other properties, such as temporal logic formulas [13] or process equivalences [26].

We compare in this section the performance of a symbolic search using BDDs and of a symbolic search using QBDDs for three examples of concurrent programs. Experiments were performed on a Sun Sparc20 workstation with 196 Megabytes of RAM. The variable orderings used in the BDD and QBDD cases are the same,



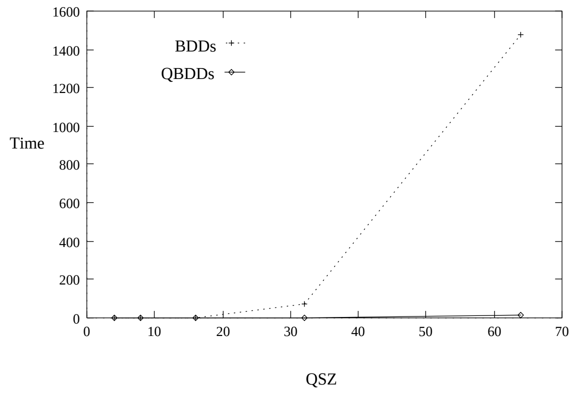
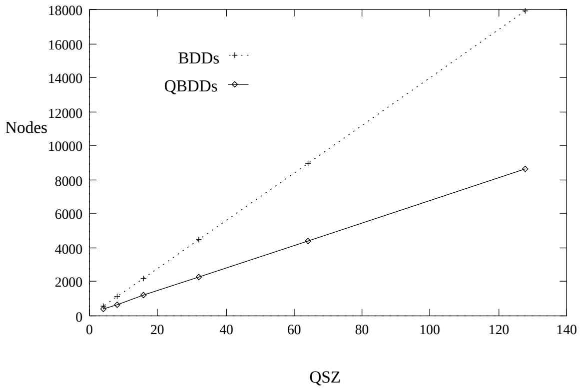


Figure 7. Comparison of performance for the producer-consumer example

except of course that, in the QBDD case, queue variables are repeated rather than duplicated. Nodes is the maximum number of BDD or QBDD nodes used during the search for representing the set of reachable states in the corresponding symbolic representation. Time (in seconds) is user time plus system time as reported by the UNIX time command.

The first example we consider is a simple version of a producer-consumer system [3]: a process (the producer) sends a finite amount of data to a second process (the consumer) via a (non-lossy) bounded FIFO queue of size QSZ. The producer may send data when the queue is not full, while the consumer may receive data when the queue is not empty. Figure 7 presents the results obtained with this example. The search using QBDDs clearly outperform, both in time and memory, the search done using BDDs.

Figure 8 presents results of experiments obtained with the alternating-bit protocol [2]. This protocol is modeled by two processes that communicate via two FIFO queues of size QSZ that may lose messages. The behavior of the two processes is



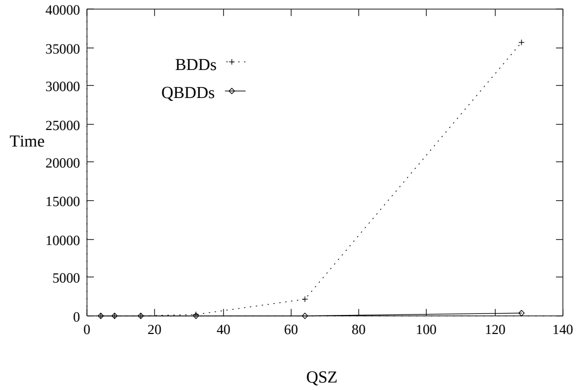


Figure 8. Comparison of performance for the alternating-bit protocol

modeled as in [4]. Again, the results obtained using QBDDs outperform the results obtained with BDDs, even though the increase in the number of nodes is linear in both cases. The tremendous difference in run-time observed for large values of QSZ is mainly due to the computations of results of queue operations, which were not optimized for speed in the BDD case. With standard BDDs, queues of size QSZ are represented by arrays of QSZ elements; enqueuing a message m is modeled by setting the value of the first empty element of the array to m, while dequeueing a message is modeled by copying the value of the first element of the array and then shifting down by one position all the other elements of the array. In contrast, QBDDs and their associated algorithms were designed to concisely represent the contents of queues and to efficiently compute queue operations.

Table 1 reports results of experiments performed with a more complex example of protocol: the full-duplex sliding-window protocol described in pages 232–233 of [30]. This protocol is modeled in about 100 lines of code in the input guarded command language for our tool. It consists of two processes communicating via

Table 1. Comparison of performance for the sliding-window protocol

	Nodes		Time	
QSZ	BDDs	QBDDs	BDDs	QBDDs
3	207,629	183,057	4,080	2,280
6	356,581	282,053	11,160	5,640
9	–	375,850	–	10,140

two FIFO queues (one for each direction) of size QSZ that may lose messages. The purpose of the protocol is to ensure that messages sent from the upper layer of one side are delivered in the same order to the upper layer of the other side. The results reported in Table 1 were obtained with a domain of sequence numbers (used to number messages) of size 4 and with sliding-windows of size 2. These results show that QBDDs outperform BDDs again, both in time and memory, although the difference in performance is less impressive, due to the use of smaller values for QSZ. For values of QSZ greater than 7, the symbolic search using BDDs could not be completed (limited by space). With QBDDs, one can obtain a complete correctness proof of the above protocol with values of QSZ up to 11 (limited by space). An exhaustive depth-first search of the state space, as implemented in SPIN [21], cannot be completed even for QSZ=3 (limited by space). Note that the model of the sliding-window protocol of [30] used here is more accurate than the model used for the experiments reported in [15], the latter model being mistakenly oversimplified.

Experiments performed with several other examples show similar improvements when using QBDDs. Interestingly, the difference in performance between QBDDs and BDDs is also significant for protocols with perfect queues, i.e., when queues do not lose messages. Finally note that, in the BDD case, interleaving queue variables of different queues in the variable ordering strongly deteriorates the performance of the search for the examples considered here. In other words, it seems easy to do worse and hard to do better with respect to the performance reported here for the BDD case.

6. Conclusion and Comparison with Related Work

Much of the nondeterminism in communication protocols comes from the concurrency in the system combined with the imperfections and delays in the communication medium. Because protocols are often meant to cope with lost or reordered messages, or transport delays, the model of the communication medium is highly nondeterministic. As a result, the set of possible queue contents is usually very large, which is one of the main causes of state explosion. Symbolic data structures such as BDDs have proven effective for dealing with large state sets. QBDDs extend this result by providing more concise and effective representations for the contents of the queues in the system. They also solve the encoding problem for queues that are unbounded or have unknown bounds. Since QBDDs include BDDs as a special case, they inherit the power of BDDs for representing and reasoning about large state spaces.

QDDs [4] are another symbolic representation for sets of queue contents. QDDs use general DFAs with arbitrary loops to represent queue contents, while QBDDs are more restricted in the types of queue contents that they can capture. For example, the set of states where a queue has an arbitrary number of m_1 messages followed by an m_2 message is representable with QDDs but not with QBDDs. QDDs can be used to construct a finite and exact representation of infinite state spaces. However, they do not include a symbolic representation for non-queue variables. In contrast, QBDDs are mainly useful for large finite state spaces (because of their limited expressiveness for infinite sets). Interesting future work is to combine the strengths of the two representations and the verification algorithms that are associated with them.

MONA [20] is a tool for manipulating expressions in monadic second-order logic. The representation used internally for finite-state machines in MONA is very similar to the QBDD representation when appropriate formulas are given. However, MONA provides no specialized operations for manipulating the representation.

In the context of BDDs, Minato *et al.* [27] have proposed *variable shifters* as a means of sharing subgraphs whose indexes differ only by a constant shift. For example, given the variable ordering $x_1 < x_2 < x_3$, the use of variable shifters results in the functions $x_1 x_2$ and $x_2 x_3$ sharing the same subgraph. For representing the contents of a single (fixed-size) queue, QBDDs and BDDs with variable shifters would give essentially the same number of nodes. However, the variable shifter representation provides no way to "undo" the shift, so in representing a constraint involving both queue and non-queue variables, the additional sharing is lost (except in the case of a queue which is ordered last).

MDDs [29] are symbolic data structures that include facilities for non-boolean variables. Because the variables are restricted to finite domains, MDDs suffer from the same limitations as BDDs when trying to represent dynamic data structures. In contrast, while we have focused on queues in this work, we believe that the ideas behind QBDDs can be naturally extended to represent other common dynamic structures such as lists, stacks, and trees.

Acknowledgments

We wish to thank Mark Staskauskas and the anonymous referees for helpful comments on this paper. A preliminary version of this paper appeared in the proceedings of the 11th IEEE Symposium on Logic in Computer Science [15].

References

1. H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ-calculus. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, July 1994.
2. K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmissions over half-duplex lines. *Communications of the ACM*, 2(5):260–261, 1969.

3. M. Ben-Ari. *Principles of Concurrent Programming.* Prentice-Hall, 1982.
4. B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. In *Proc. 8th Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12, New Brunswick, August 1996. Springer-Verlag.
5. R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
6. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991.
7. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.
8. E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462, Elounda, June 1993. Springer-Verlag.
9. E.M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992.
10. E.M. Clarke, D. E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 4th Symposium on Logic in Computer Science*, 1989.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.
12. D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.
13. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier/MIT Press, Amsterdam/Cambridge, 1990.
14. E. A. Emerson and A. P. Sistla. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In *Proc. 7th Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 309–324, Liège, July 1995. Springer-Verlag.
15. P. Godefroid and D. E. Long. Symbolic Protocol Verification with Queue BDDs. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 198–206, New Brunswick, July 1996.
16. P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, July 1991.
17. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science.* Springer-Verlag, January 1996.
18. S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 71–84, Elounda, June 1993. Springer-Verlag.
19. S. Graf and B. Steffen. Compositional minimization of finite-state systems. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, Rutgers, June 1990. Springer-Verlag.
20. J.G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. MONA: Monadic Second-Order Logic in Practice. In *Proceedings of Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–73, Aarhus, May 1995.
21. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
22. C. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 1993 Conference on Computer Hardware Description Languages and their Applications*, April 1993.
23. R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, May 1989.

24. David Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, July 1993.
25. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
26. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.
27. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 52–57. IEEE Computer Society Press, June 1990.
28. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, Stanford, June 1994. Springer-Verlag.
29. A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Proc. 1990 IEEE International Conference on Computer-Aided Design*, pages 92–95, Santa Clara, November 1990.
30. A. Tanenbaum. *Computer Neworks.* Prentice Hall, 1989.
31. A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1991.