The BDD Space Complexity of Different Forms of Concurrency

Michael Baldamus  Klaus Schneider  
University of Karlsruhe, Institute for Computer Design and Fault Tolerance  
{Michael.Baldamus,Klaus.Schneider}@informatik.uni-karlsruhe.de

Abstract

Symbolic representations using binary decision diagrams (BDDs) are popular means to cope with extremely large state spaces. However, it may be the case that the BDD representation itself is prohibitively large. We consider the BDD representations of synchronous, asynchronous and interleaved processes with communication via shared variables and present upper bounds for their sizes. For this reason, we introduce a general representation strategy where a possible exponential growth of the BDD representation can only be due to the specifics of communication and/or write conflict resolution; it is never due to the number of processes or the concurrency discipline. Moreover, certain conditions on the communication and the write conflict resolution are postulated that lead to polynomial sized BDD representations.

1. Introduction

The design of concurrent systems is a relatively error-prone task so that there is a real need to provide designers with appropriate tools. In particular, computer-aided verification is well–accepted and is nowadays used in many areas such as the design of safety–critical embedded systems.

There are many verification techniques and they are based on different specification formalisms (see [10] for an overview). Model checking procedures [7, 15] have proved to be advantageous for the verification of concurrent finite state systems. The concurrent system is described as a model of the specification logic, so that the verification task is to check whether the specifications are satisfied by that model. However, all of these techniques suffer from the state space explosion problem of large systems.

A lot of approaches have been devised for tackling the problem of state space explosion in model checking. A particularly successful one is symbolic model checking [6], where the transition relation of the system is implicitly represented by a boolean formula: any satisfying variable assignment of this formula corresponds to a transition of the system. Moreover, state sets are also represented by boolean formulas in the same symbolic manner. The decisive point is that boolean formulas can be efficiently manipulated as binary decision diagrams (BDDs) [3]. BDDs are obtained by successive case distinctions on the variables that occur in the formulas (cf. Figure 1) until all variables are eliminated. If a particular variable ordering is provided for all case distinctions, the ensuing BDDs are normal forms for boolean formulas [3]. Moreover, boolean operations on BDDs can be efficiently computed. These properties make BDDs very attractive for symbolic model checking procedures.

However, the size of a BDD may critically depend on the chosen variable ordering. In particular, the size of a BDD can be exponential in the number of its variables. There are even systems that only have exponentially sized BDDs [4]. Clearly, symbolic model checking procedures can handle these systems only to a limited extent. It is therefore desirable to estimate BDD sizes in advance, that is, before constructing the BDDs, relative to a given variable ordering. There are tight estimates for all boolean operations on BDDs, and also for further operations like composition and quantification. However, there are not many results that pertain to the BDD size of a system’s representation in terms of the structure of the system. Having a deeper knowledge of the factors that lead to exponential BDD sizes could help to design systems in such a way that tools can handle these systems more efficiently.

The contribution of the present paper is a detailed analysis of the BDD representations of concurrent finite state systems. In particular, our estimates depend (1) on the kind of concurrency, that is, interleaved, synchronous or asynchronous parallel execution, (2) on the complexity of the communication between the processes, and (3) on the way write conflicts are resolved. Having estimations in terms of these factors, we will determine their impact on the size of the BDD representation. It turns out that the number of processes has only a limited influence on the BDD size: An exponential BDD growth can only result from complex communication and/or complex write conflict resolution.

Based on our analysis, we can furthermore define classes
of concurrent systems that can be polynomially represented with BDDs: For example, the overall BDD size of a system with $n$ processes grows polynomially if the following conditions are met: first, the number of shared variables does not depend on $n$, and second, their bit length grows only logarithmically with $n$. The resolution of write conflicts, on the other hand, may be arbitrary in this case (cf. Section 6).

Our estimations necessarily depend on certain variable orderings. An essential characteristic of our approach is the use of mutually disjoint sets of boolean variables to represent each individual process. Shared variables appear as local copies that are equated afterwards to guarantee the consistency of the model. This allows us to consider process-wise constraints for variable orderings, in the sense that good orderings used to represent individual processes are not broken up. What positive effects these orderings have on the BDD representation of individual processes translates to positive effects on the BDD representation of the entire system. In this sense, our analysis constructs as a side effect compositional heuristics for variable orderings for the entire system out of constraints for process-wise variable orderings.

We are not aware of any other work on upper bounds for BDD sizes in representing shared variable systems. There is, however, related work on representing combinatioric and sequential digital circuits [5, 12, 13]. Both [5] and [12] consider the width of circuits, which is defined as the maximum number of wires through any cut through the circuits netlist. It turns out that the size of the representation BDD is exponential in this measure, while the number of gates appears as a linear factor. Hence, there is the same observation: Not the number of system components, but the communication aspect is responsible for exponential BDD sizes. This result was also obtained in the setting of BDD trees [13], which generalises that of BDDs. As [5, 12, 13] is concerned with digital circuits. It turns out that circuit representations with BDD trees can be one order of magnitude smaller than with BDDs given that the circuits itself have an appropriate tree structure. It is beyond the scope of the present paper to consider BDD tree representations. Another difference is that the work presented here is not dependent on tree-like communication topologies.

Similar to [1], our work might be seen as a systematic broadening of [12]. In [1] disjoint sets of variables are used to encode distinct processes, in a similar vein as here. However, the finite state machines of [1] communicate via transition predicates, as opposed to our communication via shared variables. It is therefore almost trivial to work with disjoint variable sets in contrast to our more difficult setting.

Finally, [8] comes closest to our work: [8] considers the representation of systems of CCS processes without value-passing [14]. The estimation given in [8] yields an upper bound which is exponential in the number of so-called actions and linear in the number of processes, which is a somewhat similar result as ours. Due to the CCS setting, however, the only concurrency discipline considered in [8] is interleaving, while we consider also synchronous and asynchronous parallel execution. Moreover, we consider a more general communication over shared variables.

The paper is organised as follows: In the next section we briefly go through preliminaries required for the rest of the paper. In Section 3, we show how systems consisting of synchronous, asynchronous and interleaved processes are represented in terms of transition systems, and Section 4 refines this to boolean encodings used for the BDD construction. Section 5 is concerned with general size estimates of these BDD representations, which is our main result. We then identify in Section 6 classes of concurrent systems that will have a polynomial BDD representation. Section 7 is devoted with an application of our results, the verification of mutex protocols. Section 8 concludes the paper.

2. Technical Preliminaries

Symbolic state space traversal as well as the symbolic representation of transition relations of finite state concurrent systems crucially depends on efficient methods to manipulate large propositional formulas. In particular, canonical normal forms are required to efficiently check the equivalence of formulas. In the past decade, binary decision diagrams (BDDs) [3] have proved to be well suited for this purpose [6, 2, 12].

BDDs are constructed by successive case distinctions on the truth value of the variables that occur in the formula. We consider the two cases, where a variable $x$ is either true or false, and construct the two cofactors $[ x ]^0$ and $[ x ]^1$ of a formula $\phi$, where $x$ has been replaced by either or . By repeating this Shannon decomposition, we finally end up with the constants and . BDDs can be pictorially illustrated as shown, for example, in Figure 1. It is important that for any fixed variable ordering used for the case distinctions, all equivalent formulas yield the same BDD. The size of the BDD can be furthermore reduced when common subgraphs are shared and unnecessary case distinctions, that is, those where the cofactors $[ x ]^0$ and $[ x ]^1$ are equivalent, are eliminated. Nowadays, highly optimized implementations of BDD data structures and algorithms are available in the form of BDD packages in the internet (see e.g. [9]).

In the following, we denote the BDD for a boolean formula by $\text{BDD}(\phi)$, and the size of a BDD, that is, the number of its nodes by $|\text{BDD}(\phi)|$. A lot of research has been directed toward the efficient construction and manipulation of BDDs. Upper bounds are well-known for all boolean operations and also for composition and quantification. The following lemma states one result that we will often use in the following.
Lemma 2.1 Let \( \phi \) be any binary boolean operation such as conjunction or disjunction. Then, for all boolean formulas \( \varphi \) and \( \psi \), the size of the BDD of \( \varphi \odot \psi \) is at most \( |BDD(\varphi)| \cdot |BDD(\psi)| \).

We will make extensive use of finite vectors of boolean variables. For every finite vector \( \mathbf{x} \), its length is denoted by \( |\mathbf{x}| \), and \( x_i \) denotes the \( i \)-th component of \( \mathbf{x} \), given that \( i \leq |\mathbf{x}| \). We abbreviate for any natural number \( n \) the set \( \{ 0, \ldots, n \} \) by \([n]\). Furthermore, if \( \mathbf{y} \) and \( \mathbf{z} \) are vectors of boolean variables, where \( |\mathbf{y}| = |\mathbf{z}| \) holds, then we abbreviate by \( \mathbf{y} = \mathbf{z} \) the following formula:

\[
\bigwedge_{i=1}^{n} (y_i \land i) \lor (\neg y_i \land \neg i)
\]

The following lemma taken from [8] is concerned with the size of the BDD representation of the formula \( y = \mathbf{z} \) (cf. Figure 1).

Lemma 2.2 Suppose that \( \mathbf{y} \) and \( \mathbf{z} \) are vectors of boolean variables, where both have length \( n \geq 1 \). Furthermore, suppose that these variables are ordered so that \( y_i \) occurs directly before \( z_i \), for each \( i \in [n] \). Then the BDD representation of the formula \( y = \mathbf{z} \) has size \( n \). Moreover, all BDDs for variable orderings where all \( y_i \)’s appear before all \( z_i \)’s have an exponential size in \( |\mathbf{y}| \).

Figure 1. BDD for \( y = \mathbf{z} \).

3. Modeling Interleaved, Asynchronous and Synchronous Concurrent Systems with Shared Data Variables

We use finite transition systems to model both shared variable processes in isolation and entire systems of such processes. Technically, a transition system can be seen as a triple \((\text{Init}, S, \rightarrow)\) where \( S \) is a state set, \( \text{Init} \subseteq S \) is a set of initial states and \( \rightarrow : S \rightarrow S \) is a transition relation.

We assume that every transition is labelled with a boolean formula. The intuition is that such a formula is an enabling condition. We write \( s \rightarrow s' \) for \((s, s') \in \rightarrow\) with label \( \rightarrow \).

To model shared variable processes in isolation, we assume that every process \( P \) has a finite set of control flow locations \( L \) and a finite set of variables \( V \), where each \( x \in V \) has a finite set of possible values. A \( P \)-assignment \( a \) is a function from \( V \) to the union \( \mathcal{P} \) of all \( P \)-assignments denoted by \( \bigcup P \). As usual, the update of a variable \( x \) by a value \( \phi \) in an assignment \( a \) is denoted by \( a[x := \phi] \) and is defined by

\[
a[x := \phi](y) = \begin{cases} a(y) & \text{if } x \neq y \\ a(y) & \text{if } x = y.
\end{cases}
\]

An empty update \( a[\ ] \) is given by \( a[\ ] = a \).

Then \( P \) is modeled as a transition system \((\text{Init}, S, \rightarrow)\) as follows: The intuition is that transitions correspond to the execution of \( P \), where at most one variable is updated in each transition.

- The set of states is given by \( S = \mathcal{P} \), that is, a state is assumed to consist of a \( P \)-assignment and a control flow location.

- Transitions must be such that \( (a, \ell) \rightarrow (a', \ell) \) if and only if \( a = a[\ ] \) or \( a = a[x:=\ ] \) for some \( x \) and \( a \in \mathcal{P} \), where \( \ell \) is a formula over \( V \).

A \( P \)-update is defined to be of the form \( x := \phi \) with \( x \in V \) and \( \phi \in \mathcal{P} \). The set of all \( P \)-updates is denoted by \( U \).

We write \( (a, \ell) \rightarrow (a', \ell) \) whenever \( (a, \ell) ) \rightarrow (a', \ell) \) with \( a = a[\ ] \), and we write \( (a, \ell) \rightarrow (a', \ell) \) whenever \( (a, \ell) \rightarrow (a', \ell) \) with \( a = a[x := \phi] \). Note that \( a[\ ] = a[x := a(x)] \) since that either update does not occur in parallel with another update of \( a \) by some other process.

For the remainder of this paper, we fix some arbitrary collection \( P_1, \ldots, P_n \) of processes with \( P_i = (\text{Init}_i, S_i, \rightarrow_i), n \geq 1 \). Unless stated otherwise, \( i \) and \( j \) range over the set \([n]\). We write \( i, j, \ldots \) for \( 1, \ldots, n \), respectively. Without loss of generality, we assume \( i < j \) whenever \( i \neq j \). As we consider shared variable processes, we do not assume \( i < j \), of course. The set of all variables is defined to be the union \( \bigcup_{i=1}^{n} V_i \) and a global assignment is a function, say,
from the union $\in$ with $(x) \in$ for each $x \in$. The set of all global assignments is denoted by $GA$; the update of a global assignment is defined analogously to the update of a local assignment; the restriction of a global assignment to $i$ for some $i$ is denoted by $|i|$, that is, $|i|$ is a $P_i$-assignment. Then, for each $x \in$, the set of $x$-updates is given by $= \{x := | \in \}$. Furthermore, the set $Own$ is given by $Own = \{i | x \in i\}$, that is, $Own$ is the set of all indexes of processes that share $x$. Finally, we write $LV$ for the set of all control flow location vectors of the given collection of processes, that is, $LV$ is defined to be the cartesian product of various resolution schemes for such conflicts.

**Definition 3.2** (Conflict Resolution Schemes)

1. For each $x \in$, a conflict resolution scheme for $x$ is a binary relation between
   
   (a) the set of nonempty families $(i)_{i \in}$ with $Own$ and $i \in$ for each $i \in$ and
   
   (b) the set $\cup$.

2. A (global) conflict resolution scheme is a family $( ) \in$ where, for each $x \in$, is a conflict resolution scheme for $x$.

Here, the idea is that concurrent writes $(i)_{i \in}$ to some $x$ by some $P_i$’s with $i \in$ can yield the update whenever $(i)_{i \in}$. We fix some arbitrary global conflict resolution scheme $( ) \in$ for the remainder of the paper. Instead of $( ) \in$, we write $( )$. Note that non–conflicting writes will usually take effect without conflict resolution. To model this, it suffices to fix $( )$ so that $(i)_{i \in}$ whenever is a singleton set.

We proceed with defining the asynchronous and the synchronous composition of shared variable processes. In the asynchronous case the idea is that a global transition arises from local transitions by processes $P_i$, where $i \in$ for some non–empty index set. The global transition updates variables according to the conflict resolution scheme and leaves all $P_i$’s with $\notin \in$ their current state.

**Definition 3.3** (Asynchronous Composition of Processes with Shared Variables) The asynchronous composition of $P_1, \ldots, P_n$ is a transition system $= (Init, S, \rightarrow)$ such that:

1. $S = GA \cup LV$ (A global state consists of a global assignment and a vector of local control flow states.)

2. $init = \{(i, i) | (i, i) \in Init, for each i \in [n]\}$ (A global state is initial if, for every $i$, its projection on the $i$–th process is an initial state of that process.)

3. $(i, i) \rightarrow (i, m)$ whenever, for some $i \in [n]$, there is an $m \in i$ so that:

    (a) $|i, i| \rightarrow (|i, m|)$

    (b) $i = m$ and $j = j$ for all $\neq i$

    (c) $\subseteq \emptyset$ The idea is that an interleaved global transition arises from some local transition by some process $P_i$. The local transition may update the global assignment and leaves all processes distinct from $P_i$ in their current state.

If we do not have interleaving, then it is possible that distinct processes want to update the same variable at the same time. There are various resolution schemes for such conflicts. One possibility is to use priorities, meaning that some arbitration is in place to select one particular update for coming into effect. Another possibility is non–determinism, which means that one particular update is selected randomly. Still another possibility is what we may call completely non–deterministic, meaning that a write conflict may yield an arbitrary update. A fourth possibility consists of aborting the system whenever there is a write conflict. We do not consider this particular scheme but allow any of the other ones.

To model write conflict resolution schemes, the following definition can be instantiated.

**Definition 3.4** (Synchronous Composition of Processes with Shared Variables) The synchronous composition of $P_1, \ldots, P_n$ is a transition system $= (Init, S, \rightarrow)$. The
To sum up, we can distinguish interleaving, asynchrony and synchrony by the set of indexes of processes that are active in a global transition. Interleaving means that is a singleton set; asynchrony means that is a subset of the set of all indexes; synchrony means that equals the set of all indexes. Another important aspect is that we decidedly do not subsume interleaving under asynchrony; the reason is that we are not only interested in verifying safety but also in verifying liveness properties. As for liveness properties, the succession of states becomes important and not only what states are reachable. That succession differs depending on what concurrency discipline is in place.

4. Boolean Representation of the Interleaved, Asynchronous and Synchronous Transition Relations

Based on the system models of the previous section, we will now consider their encoding with boolean formulas. We use separate boolean formulas for individual processes, and also for communication and write conflict resolution. These formulas can be combined in simple ways to yield boolean representations of entire systems. Moreover, they allow us to find our upper bounds in representing entire systems with BDDs.

We begin with the boolean representation of individual transition relations, using mutually disjoint sets of boolean variables for distinct processes. Clearly, variables to encode the locations of the control flow of individual processes must be disjoint. Shared variables, however, can not be treated locally, at least at a first glance. We can, however, nevertheless achieve separation or, in other words, compositionality in analysing the system. The means of that is to introduce a unique copy of each shared variable for each process having access to that variable. Besides these copies, we retain each shared variable proper. We often speak of a formula on the system level if those actual shared variables are involved in it.

Technically, let \( \text{width} \) be the number of boolean variables necessary to represent . In other words, \( \text{width} \) is given by \( \text{width} = \log_2 | \cdot | \). Moreover, let \( x \) be a vector of such variables, that is, \( x = (x_1, \ldots, x_{\text{width}}) \). Further still, let \( x^{\text{ati}} = (x^{\text{ati}}_1, \ldots, x^{\text{ati}}_{\text{width}}) \) be the above-mentioned unique copy of \( x \) for each \( i \in \text{Own} \).

Further technicalities are as follows: For each \( i \in \cdot \), a unique encoding of each \( \in \cdot \) as a vector of length \( \text{width} \) over \{0, 1\} must be given. A masking operator on vectors of boolean variables and bit vectors is then given by

\[
y = \bigwedge_{i=1}^{n} (-y_i) \text{ if } i = 1
\]

where \(|y|\) must be equal to or smaller than \(|\cdot|\). Moreover, \( \mathbf{X} \) is the next operator, which yields a new, unique, boolean variable \( \mathbf{X} \) for every ordinary boolean variable \( \cdot \). As usual, those next variables are used to encode state changes. In accordance with that, elementary variables are called current variables. We introduce a local next operator \( \mathbf{LX} \) to encode local variable updates. Analogously to \( \mathbf{X} \), it yields a new, unique, boolean variable \( \mathbf{LX} \) for every ordinary boolean variable \( \cdot \). For every vector \( y = (y_1, \ldots, y\cdot) \) of boolean variables, \( \mathbf{X}y \) or \( \mathbf{LX}y \) denotes the vector \((\mathbf{X}y_1, \ldots, \mathbf{X}y\cdot)\) or \((\mathbf{LX}y_1, \ldots, \mathbf{LX}y\cdot)\), respectively. Finally, a (boolean) update flag \( i \) is introduced for every \( i \in \cdot \) and every \( i \in \text{Own} \). We use such an \( i \) to mark that the \( i \)-th process updates \( x \).

The \( i \)-th transition relation can then be represented as a boolean formula as follows:

\[
\bigvee_{f \in \cdot} \left( \bigwedge_{i \in \cdot} \left( \begin{array}{c}
\neg z_i \\
\neg \text{if } i \notin \cdot \\
\text{if } i \in \cdot
\end{array} \right) \right) \\
\land \text{TRUE \ if } is \ of \ the \ form \\
\mathbf{LX}x^{\text{ati}} \ if \ is \ of \ the \ form \ x := \\
\bigwedge_{i \in \cdot} \left( \begin{array}{c}
\neg x \ \text{if } \neq \\
\text{if } = \end{array} \right) \\
\bigwedge_{i \in \cdot} \left( \begin{array}{c}
\neg \mathbf{X} \ \text{if } \neq \\
\mathbf{X} \ \text{if } = \end{array} \right)
\]

This formula is a straightforward disjunction over all transitions in \( \neg i \). Every disjunct represents the effect of the corresponding transition on the control flow state (lower part) and its updates (upper half). The condition is rendered as a sub-formula over the boolean encodings of the variables . This particular transformation is a straightforward matter of structural induction; we do not go into its details.

We continue with two boolean formulas necessary to represent asynchrony and interleaved concurrency. The first formula expresses that the \( i \)-th process performs an idling transition, where it remains at its current control flow
location and does not perform any variable update.

\[
(\bigwedge_{i \in I} \neg s_i) \land (\bigwedge_{i \in I} x_i = x_{\text{initial}}) \quad \text{(Idle)}
\]

The second formula is already located at the system level. Its purpose consists of scheduling interleaved processes. As a prerequisite, a (boolean) scheduling-variable \( s_i \) is introduced for each \( i \in [n] \). The formula states that exactly one \( s_i \) must be true at any time.

\[
\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \neg s_j \text{ if } i \neq j \land s_j \text{ if } i = j \quad \text{(Sched)}
\]

Before we can put everything together, we still need to represent the conflict resolution scheme. This formula looks certainly somewhat convoluted. In reality, everything is pretty straightforward. The formula is a conjunction over all \( x \); the conjunct for some \( x \) represents the conflict resolution scheme for \( x \). This conjunct, in turn, is a disjunction whose second disjunct covers the case “no update;” the first disjunct is a disjunction over all elements of the relation \( \equiv \).

\[
\bigwedge_{i \in I} \bigvee_{x \in \text{Own}_x} x \equiv x_{\text{at} i} \quad \text{(Comm)}
\]

Before we can put everything together, we still need to represent the conflict resolution scheme. This formula looks certainly somewhat convoluted. In reality, everything is pretty straightforward. The formula is a conjunction over all \( x \); the conjunct for some \( x \) represents the conflict resolution scheme for \( x \). This conjunct, in turn, is a disjunction whose second disjunct covers the case “no update;” the first disjunct is a disjunction over all elements of the relation \( \equiv \).

\[
\bigwedge_{i \in I} \bigvee_{x \in \text{Own}_x} x \equiv x_{\text{at} i} \quad \text{(Comm)}
\]

Boolean representations of entire systems are as stated in Definition 4.1. The synchronous case is the most elementary one; we only need to form the conjunction of \( (\neg s_i) \) for all \( i \), \( \text{Comm} \) and \( ((\ )) \). In the asynchronous case, \( (\neg s_i) \) is replaced by \( (\neg s_i) \lor \text{Idle} \), to represent that idling may occur at any time; in the interleaving case, \( (\neg s_i) \) is replaced by \( (s_i \land (\neg s_i)) \lor (\neg s_i \land \text{Idle}_i) \) and to adjoin \( \text{Sched} \) to enforce the scheduling discipline.

**Definition 4.1** *(Boolean Representation of the Interleaved, Asynchronous and Synchronous Transition Relations)* The boolean representation of \( \rightarrow, \rightarrow || \) and \( \rightarrow \) can be as follows:

\[
\left( \bigwedge_{i=1}^{n} (\neg s_i) \right) \land \text{Comm} \land ((\ )) \quad (\rightarrow)
\]

\[
\left( \bigwedge_{j=1}^{n} (\neg s_j) \lor \text{Idle}_i \right) \land \text{Comm} \land ((\ )) \quad (\rightarrow ||)
\]

\[
\left( \bigwedge_{i=1}^{n} (s_i \land (\neg s_j)) \lor (\neg s_i \land \text{Idle}_i) \right) \land \text{Sched} \land \text{Comm} \land ((\ )) \quad (\rightarrow)
\]

Note that the formula for write conflict resolution, \( ((\ )) \), appears within the formula for interleaving, \( (\rightarrow) \), although real-world interleaving does not lead to write conflicts in the first place. The reason for that appearance is that \( ((\ )) \) describes the relationship between local and global updates, regardless of whether any true write conflicts occur.

As a final remark, space problems in representing transition systems with BDDs are to our experience normally due to space problems in representing the corresponding transition relations. For this reason, we have omitted the representation of sets of initial states already in this section.

5. Upper Bounds for BDD Sizes

We now turn to the BDD normal forms of the boolean formulas from the preceding section. To find upper bounds for their respective sizes, we analyse what some BDDs look like. Another technique that we use consists of applying the lemma about the size of BDD normal forms for compound boolean formulas (Lemma 2.1). As explained earlier, all results are based on block-wise variable ordering constraints.

The prototype constraint, \( 1 \), leaves everything open apart from that distinct blocks may not overlap and apart
from that each scheduling variable $s_i$ must directly precede the corresponding block. As for defining it, let $i$ be the variable block used to encode the transition relation of the process $P_s$, that is, $i$ is given by

$$i = \{ i, x_1^{ati}, Lx_1^{ati}, \ldots, x_{width_s}^{ati}, Lx_{width_s}^{ati} \} \in i \cup i \cup \{ X \mid \in i \}.$$ 

The constraint is given by the following scheme:

| $s_1$ | 1 | $s_n$ | n | 1 |

The following lemma is our first application of Lemma 5.1. Its topic is the precise size of the BDD that represents the scheduling formula $\text{Sched}$.

**Lemma 5.1** Relative to any variable ordering satisfying $1$, the size of $\text{BDD}(\text{Sched})$ is

$$|\text{BDD}(\text{Sched})|.$$ 

**Proof.** The BDD for $\text{Sched}$ consists of a top vertex and of two strands of length $n$ (see Figure 2). Hence, the BDD contains a total of $n$ vertices. □

![Figure 2. BDD for Sched.](Image)

The next lemma is concerned with the size of the BDD that represents the idling formula $\text{Idle}_i$, $i \in [n]$. The lemma requires variable ordering constraints where

and $X$ occur next to each other for every location $i$, $i \in [n]$.

Before we state the lemma, we recall that keeping together ordinary variables and their respective counterparts with $X$ is almost always the best. For this reason, $2$ should be compatible with just about every practically useful variable ordering. In particular, $2$ is compatible with $1$. — One can construct examples where keeping together ordinary variables and their counterparts with $X$ is not the best. For this reason, we can not stipulate that this strategy is the best one in every case. It has, however, been advantageous in many practical applications of symbolic model checking to date. This situation is the reason why we do not consider any other one.

**Lemma 5.2** Relative to any variable ordering satisfying $2$, the size of $\text{BDD}(\text{Idle}_i)$ is

$$|\text{BDD}(\text{Idle}_i)|.$$ 

**Proof.** If $i$ is empty, then $\text{Idle}_i = \bigwedge_{i \in i} X \equiv$ and the result follows from Lemma 2.2. If $i$ is not empty, then $\text{BDD}(\text{Idle}_i)$ is obtained from $\text{BDD}(\bigwedge_{i \in i} X \equiv)$ by adding exactly one vertex for each variable in $i$. (See Figure 1. For every $x \in i$, the vertex added to $\text{BDD}(\bigwedge_{i \in i} X \equiv)$ is labelled with $i$.)

□

Having put these lemmas in place, we can state our first main result. Methodologically, the theorem is about upper bounds for BDD representations of the synchronous, asynchronous and interleaved transition relations introduced in Section 3; technically, it is about upper bounds for the BDD normal forms of the corresponding boolean formulas from Definition 4.1.

**Theorem 5.3** (Upper Bounds for BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Relative to any variable ordering satisfying $1$ and $2$, the respective size of $\text{BDD}(\rightarrow)$, $\text{BDD}(\rightarrow\parallel)$ and $\text{BDD}(\rightarrow\parallel\parallel)$ is bounded as follows:

$$\left(\left(\sum_{i=1}^{n} \text{BDD}(\rightarrow)\right) \right)\left(\text{BDD}(\text{Comm})\right) \left(\text{BDD}(\rightarrow\parallel)\right)$$

$$\left(\left(\sum_{i=1}^{n} \text{BDD}(\rightarrow)\right) \right)\left(\text{BDD}(\text{Comm})\right) \left(\text{BDD}(\rightarrow\parallel)\right)$$
 Proof. As the other two cases are similar but easier, we consider only the upper bound for representing the interleaved transition relation. First of all, we apply Lemma 2.1 and Lemma 5.1 to Definition 4.1, obtaining

\[
|\text{BDD}(\rightarrow)\|
\leq \left( \sum_{i=1}^{n} \text{BDD}(\neg i) \right) \cdot |\text{BDD}(\text{Comm})| \cdot |\text{BDD}(\text{Idle})|
\]

It remains to obtain an upper bound for

\[
\left( \sum_{i=1}^{n} (s_i \land \neg i) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

To this end, consider Figure 3. First of all, the BDD for \((\bigwedge_{i=1}^{n} (s_i \land \neg i) \lor (\neg s_i \land \text{Idle}_i))\) has the structure depicted in (a), where \(i\) stands for the BDD for \((s_i \land \neg i) \lor (\neg s_i \land \text{Idle}_i)\) without leafs; this structure is so since the variable sets of distinct \(i\)'s do not overlap. Hence, the upper bound is

\[
\left( \sum_{i=1}^{n} \text{BDD}(s_i \land \neg i) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

Now it remains to obtain an upper bound for \(|\text{BDD}(s_i \land \neg i) \lor (\neg s_i \land \text{Idle}_i)|\). To this end, consider (b) and (c) in Figure 3. The size of \(\text{BDD}(s_i \land \neg i)\) is \(|\text{BDD}(\neg i)|\); the size of \(\text{BDD}(\neg s_i \land \text{Idle}_i)\) is \(|\text{BDD}(\text{Idle}_i)|\). Thus, together with Lemma 2.1 and Lemma 5.2, we have

\[
\text{BDD}(s_i \land \neg i) \lor (\neg s_i \land \text{Idle}_i)
= \left( \text{BDD}(\neg i) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

In practice, scalable systems can often be represented so that the BDD representation of any individual process grows at most polynomially as the number of processes grows. Thus, by Theorem 5.3, the upper bound for the size of an overall representation BDD will usually grow polynomially if the upper bounds for \(\text{BDD}(\text{Comm})\) and \(\text{BDD}(\text{Idle})\) grow polynomially. Theorem 5.3 thus follows through on our claim that disadvantageous communication and conflict resolution are the principal reasons for exponential growth of overall representation BDDs. Our block–wise variable ordering constraints are decisive in obtaining this result; the reason is that, without them, we could only use Lemmas 2.1 and 5.1 and not the BDD in Figure 3(a). This situation would require to replace the sum over \([n]\) by a product over \([n]\). With that sum, we have the following corollary to Theorem 5.3. It states our results in complexity–theoretic terms.

**Corollary 5.3.a** (Space Complexity of BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Relative to any variable ordering satisfying 1 and 2, the space complexity of \(\text{BDD}(\rightarrow)\), \(\text{BDD}(\rightarrow||)\) and \(\text{BDD}(\rightarrow)\) is as follows:

\[
\text{BDD}(\rightarrow)
= \left( \text{BDD}(\rightarrow ||) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

In practice, scalable systems can often be represented so that the BDD representation of any individual process grows at most polynomially as the number of processes grows. Thus, by Theorem 5.3, the upper bound for the size of an overall representation BDD will usually grow polynomially if the upper bounds for \(\text{BDD}(\text{Comm})\) and \(\text{BDD}(\text{Idle})\) grow polynomially. Theorem 5.3 thus follows through on our claim that disadvantageous communication and conflict resolution are the principal reasons for exponential growth of overall representation BDDs. Our block–wise variable ordering constraints are decisive in obtaining this result; the reason is that, without them, we could only use Lemmas 2.1 and 5.1 and not the BDD in Figure 3(a). This situation would require to replace the sum over \([n]\) by a product over \([n]\). With that sum, we have the following corollary to Theorem 5.3. It states our results in complexity–theoretic terms.

**Corollary 5.3.a** (Space Complexity of BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Relative to any variable ordering satisfying 1 and 2, the space complexity of \(\text{BDD}(\rightarrow)\), \(\text{BDD}(\rightarrow||)\) and \(\text{BDD}(\rightarrow)\) is as follows:

\[
\text{BDD}(\rightarrow)
= \left( \text{BDD}(\rightarrow ||) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

In practice, scalable systems can often be represented so that the BDD representation of any individual process grows at most polynomially as the number of processes grows. Thus, by Theorem 5.3, the upper bound for the size of an overall representation BDD will usually grow polynomially if the upper bounds for \(\text{BDD}(\text{Comm})\) and \(\text{BDD}(\text{Idle})\) grow polynomially. Theorem 5.3 thus follows through on our claim that disadvantageous communication and conflict resolution are the principal reasons for exponential growth of overall representation BDDs. Our block–wise variable ordering constraints are decisive in obtaining this result; the reason is that, without them, we could only use Lemmas 2.1 and 5.1 and not the BDD in Figure 3(a). This situation would require to replace the sum over \([n]\) by a product over \([n]\). With that sum, we have the following corollary to Theorem 5.3. It states our results in complexity–theoretic terms.

**Corollary 5.3.a** (Space Complexity of BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Relative to any variable ordering satisfying 1 and 2, the space complexity of \(\text{BDD}(\rightarrow)\), \(\text{BDD}(\rightarrow||)\) and \(\text{BDD}(\rightarrow)\) is as follows:

\[
\text{BDD}(\rightarrow)
= \left( \text{BDD}(\rightarrow ||) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

In practice, scalable systems can often be represented so that the BDD representation of any individual process grows at most polynomially as the number of processes grows. Thus, by Theorem 5.3, the upper bound for the size of an overall representation BDD will usually grow polynomially if the upper bounds for \(\text{BDD}(\text{Comm})\) and \(\text{BDD}(\text{Idle})\) grow polynomially. Theorem 5.3 thus follows through on our claim that disadvantageous communication and conflict resolution are the principal reasons for exponential growth of overall representation BDDs. Our block–wise variable ordering constraints are decisive in obtaining this result; the reason is that, without them, we could only use Lemmas 2.1 and 5.1 and not the BDD in Figure 3(a). This situation would require to replace the sum over \([n]\) by a product over \([n]\). With that sum, we have the following corollary to Theorem 5.3. It states our results in complexity–theoretic terms.

**Corollary 5.3.a** (Space Complexity of BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Relative to any variable ordering satisfying 1 and 2, the space complexity of \(\text{BDD}(\rightarrow)\), \(\text{BDD}(\rightarrow||)\) and \(\text{BDD}(\rightarrow)\) is as follows:

\[
\text{BDD}(\rightarrow)
= \left( \text{BDD}(\rightarrow ||) \lor (\neg s_i \land \text{Idle}_i) \right).
\]

In practice, scalable systems can often be represented so that the BDD representation of any individual process grows at most polynomially as the number of processes grows. Thus, by Theorem 5.3, the upper bound for the size of an overall representation BDD will usually grow polynomially if the upper bounds for \(\text{BDD}(\text{Comm})\) and \(\text{BDD}(\text{Idle})\) grow polynomially. Theorem 5.3 thus follows through on our claim that disadvantageous communication and conflict resolution are the principal reasons for exponential growth of overall representation BDDs. Our block–wise variable ordering constraints are decisive in obtaining this result; the reason is that, without them, we could only use Lemmas 2.1 and 5.1 and not the BDD in Figure 3(a). This situation would require to replace the sum over \([n]\) by a product over \([n]\). With that sum, we have the following corollary to Theorem 5.3. It states our results in complexity–theoretic terms.

**Corollary 5.3.a** (Space Complexity of BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Relative to any variable ordering satisfying 1 and 2, the space complexity of \(\text{BDD}(\rightarrow)\), \(\text{BDD}(\rightarrow||)\) and \(\text{BDD}(\rightarrow)\) is as follows:

\[
\text{BDD}(\rightarrow)
= \left( \text{BDD}(\rightarrow ||) \lor (\neg s_i \land \text{Idle}_i) \right).
\]
The rest of this section is concerned with devising an upper bound for the BDD normal form of \( \text{Comm} \), the communication formula. Let \( \text{Own} = \{ 1, \ldots, \text{Own}_x \} \) for each \( x \in \). An ordering constraint is then given by

\[
\bigwedge_{i \in \text{Own}_x} x^\text{st}_1 \cdots x^\text{st}_{\text{width}_x},
\]

This constraint pertains only to bit vectors used to encode shared variables. It is compatible with \( 1 \). Moreover, it involves the system level in that \( x^\text{st}_1, \ldots, x^\text{st}_{\text{width}_x} \) must be part of the ordering for each \( x \). Note that \( x^\text{st}_1, \ldots, x^\text{st}_{\text{width}_x} \) must be ordered oppositely to the rest. This stipulation helps illustrating the proof of the following theorem; it does not influence the upper bound.

**Theorem 5.4** Relative to any ordering satisfying \( u \), the size of \( \text{BDD}(\text{Comm}) \) is bounded as follows:

\[
\text{upper bound for } |\text{BDD}(\text{Comm})| \leq \left( (|\text{Own}| - u) \text{width} \right)^{|\text{Comm}|}.
\]

Before we prove this upper bound, we recall the widely-held belief that binary decision diagrams are often unwieldy in representing data paths. This opinion is confirmed by the presence of \( \text{width} \) in exponent position. We note, however, that \( \text{width} \) does not need to grow linearly with \( n \). We will exploit this fact in the next section.

**Proof of Theorem 5.4.** By Lemma 2.1 applied to \( \text{Comm} \),

\[
|\text{BDD}(\text{Comm})| \leq \text{BDD} \left( \bigwedge_{i \in \text{Own}_x} x \equiv x^\text{st}_i \right).
\]

It remains to obtain an upper bound for

\[
|\text{BDD}(\bigwedge_{i \in \text{Own}_x} x \equiv x^\text{st}_i)|,
\]

\( x \in \). Actually, however, we determine the exact size of each such a BDD. To this end, we observe that such a BDD can be thought of as consisting of \( |\text{Own}| \) sections. The topmost section corresponds to the process with the lowest index in \( \text{Own} \); it has \( \text{width}_x \) vertices. The next \( |\text{Own}| \) sections correspond to the other processes whose indices are in \( \text{Own} \); each of them has \( \text{width}_x \) vertices. The second–to–last section corresponds to the system level; it has \( \text{width}_x \) vertices. The bottom section consists of the two leafs. To conclude the proof, all that is left adding up these vertex counts. \( \square \)

### 6. A Scenario with Polynomial Upper Bound

We now consider a scenario where our results from the preceding section yield polynomial upper bounds for BDD sizes in representing entire systems. We choose the completely non-deterministic resolution of write conflicts (cf. Section 1). Specifically, we have a polynomial upper bound for the BDD representation of the boolean formula describing that discipline, given that another block-wise variable ordering constraint is in place. The sought–after upper bound is then obtained by conjoining all variable ordering constraints and by invoking what we have discussed after the proof of Theorem 5.3. Just two additional preconditions are required: first, the number of shared variables must remain constant; second, the width of the boolean encoding of every shared variable \( x, \text{width}_x \), must grow at most logarithmically with the number of processes.

We begin with the technical definition of the completely non-deterministic resolution of write conflicts. This definition has two parts: first, it must be the case that \(( i)_i \in i \)

implies \( i = \), meaning that write operations without conflict take effect unchanged; second, \(( i)_i \in \) \( \geq \) must imply \(( i)_i \in \) \( = \) \( \forall \), meaning that every write conflict may result in an update by an arbitrary bit pattern. Technically, we only require that \( \geq \) be an element of \( = \), so a write conflict will actually result in an update by an arbitrary value from \( = \). Without loss of generality, however, we may require \( = \text{width}_x \). To proceed, the above–mentioned ordering constraint is as follows:

\[
\bigwedge_{i \in \text{Own}_x} \ldots LX x^\text{st}_1 \cdots LX x^\text{st}_{\text{width}_x}.
\]

This constraint pertains to update flags, to boolean variables with \( LX \) and to bit vector variables (with \( X \)) on the system level. As \( 2 \) and \( = \), it is compatible with our prototype constraint \( 1 \); as \( = \), the system level variables are ordered oppositely to the rest. This stipulation again helps illustrating the proof of the theorem we have in mind; it does not influence the upper bound.

**Theorem 6.1** Suppose \(( \_ ) \) resolves write conflicts completely non-deterministically. Relative to any ordering satisfying \( = \), the size of \( \text{BDD}(\_ ) \) is then bounded as fol-
Proof. Similar to the proof of Theorem 5.4. By Lemma 2.1 applied to \(BDD(0_0)\),

\[
BDD(0_0) \leq BDD(0_0) \cdot width_x \cdot width
\]

It remains to obtain an upper bound for \(BDD(\rightarrow_i)\), \(x \in n\). Actually, however, we determine the exact size of each such BDD. To this end, we observe that such a BDD can be thought as consisting of \(\text{Own} \mid \) sections (cf. Figure 4). The topmost section corresponds to the process with the lowest index in \(\text{Own} \); it has \(width_x\) vertices. The next sections correspond to all other processes whose index is in \(\text{Own} \); each of them has \(width_x\) vertices. The second–to–last section corresponds to the system level; it has \(\bigoplus_{i \in [n]} width_x \cdot width\) vertices. The bottom section consists of the two leaves. To conclude the proof, all that is left adding up these vertex counts.

We can now reconsider the claim from the beginning of this section. For simplicity, we consider only the synchronous case. We assume a variable ordering that satisfies \(1, 2 \) and \(3\), and we assume that write conflicts are resolved completely non–deterministically. Theorems 5.3, 5.4 and 6.1 can then be applied to the representation of the synchronous transition relation. The ensuing upper bound is as follows:

\[
\text{upper bound for } \left| BDD(0_0) \right|
\]

\[
\in \left( |\text{Own} \mid \right) width \cdot width_x
\]

Now suppose the system under consideration is scalable. As discussed in Section 5, for polynomial growth we assume that \(BDD(\rightarrow_i)\) grows at most polynomially with \(n\). Moreover, we need to assume that the number of shared variables remains constant, and that \(width\) grows at most logarithmically with \(n\) for every \(x\). Every factor or summand belonging to the upper bound is then at most polynomial in \(n\), so we have proved our second main result (cf. Corollary 5.3.a):

**Theorem 6.2 (Space Complexity of BDD Sizes in Representing the Interleaved, Asynchronous and Synchronous Transition Relations) Under the assumptions from the preceding paragraph, the space complexity of the BDD representations of the interleaved, asynchronous and synchronous transition relations is at most polynomial in the number of processes.**

It is straightforward to estimate the degree of the polynomial given that one knows the degree of the polynomials bounding the growth of each individual \(BDD(\rightarrow_i)\). In the case of the refined upper bound for \(BDD(\rightarrow)\), for example, one multiplies the maximum degree of the polynomials bounding \(BDD(\rightarrow_i)\), \(i \in [n]\), with the degrees of polynomials bounding

\[
\left( |\text{Own} \mid \right) width \cdot width_x
\]

and

\[
\left( |\text{Own} \mid \right) width_x \cdot width, x \in n
\]

7. Verifying Mutex Protocols

The scenario from the preceding section typically applies to mutex protocols that are based on utilising lock variables. A well–known example of such a protocol is Fischer’s protocol [11], where a lock variable is used to store the index of the process that is allowed to enter its critical section. The scenario basically applies in these cases since the number of lock variables (e.g. \(x\)) is constant and since their width grows only logarithmically with the number of processes. Moreover, interleaved processes do not generate write conflicts. Hence, the respective model can be so formulated as to assume the completely non–deterministic resolution discipline. As for synchronous and asynchronous processes, completely non–deterministic write conflict resolution can be assumed as an abstraction of any concrete scheme at work in the system under consideration. If the ensuing model has the mutex property, then the system will usually have it too.

As an example, we have verified a Fischer–like mutex protocol for synchronous processes. Below we present statistics. Before that, let us note that the state space grows exponentially in the number of processes (Figure 5). We note also that the curve in Figure 5 has small jumps whenever an additional bit is required to encode the lock variable.

For verification, we have represented the system exactly as described in Section 4, and we have used a variable ordering that conforms to all our block–wise heuristics \(1, 2, \) and \(3\). The diagram in Figure 6 on the second–to–next page shows how many BDD vertices are necessary.
Figure 4. Illustration of the proof of Theorem 6.1. The diagram depicts the conflict resolution BDD for any $x$ with $width = 2$ and $\{Own\} = \{1, 2, 3\}$. Outgoing edges not shown lead directly to the $0$-leaf. Dangling outgoing edges lead directly to the $1$-leaf.

Figure 5. State spaces of mutex protocol, drawn on a semi-logarithmic scale. The number grows non-linearly but still fairly moderately with the number of processes. As in the case of Figure 5, we can also observe that there is a jump whenever an additional bit is required to encode the lock variable. Another interesting observation is that in each case only a few additional BDD vertices are used to verify the mutex property (gray curve), which entails computing all reachable states.

8. Conclusions

To summarise, the basis of what has been presented here is the model of interleaved, synchronous and asynchronous concurrent systems with shared data variables introduced in Section 3. The boolean representations of those systems as introduced in Section 4 then lay the groundwork
for examining the complexity of representations with BDDs (Sections 5 and 6). A technical novelty is the use of mutually disjoint sets of boolean variables to represent individual processes in the presence of communication via shared data variables. This approach allows one to treat various aspects in isolation, namely concurrency, communication and write conflict resolution. The individual contributions of these aspects to the space complexity of the BDD representations of concurrent systems with shared data variables can then be singled out. In certain scenarios a polynomial overall complexity arises (Section 6). These settings are practically important for the analysis of mutex protocols by formal verification (Section 7).

As for future work, it would certainly be desirable to consider additional application classes. It would also be desirable to improve the upper bounds and complexity results from Sections 5 and 6, possibly by relying less on Lemma 2.1 and more on BDD analysis such as in Figure 4. Another important topic would be what upper bounds can be obtained if one works without local copies of shared variables. Still another important topic would be what BDD sizes arise if one does not deal with write conflicts in the way we do. They could be considered failures instead.

References


