Technical Report: BDD Library for Model Checking Hierarchical Systems

Truong Khanh Nguyen[1], Jun Sun[2], Yang Liu[1], and Jin Song Dong[1]

[1] School of Computing, National University of Singapore
{nguyent9,liuyang,dongjs}@comp.nus.edu.sg
[2] Singapore University of Technology and Design
sunjun@sutd.edu.sg

1 Introduction

Binary Decision Diagram (BDD) based symbolic model checking is capable of verifying systems with a large number of states [1]. Its effectiveness was evidenced by the recent success of the Intel i7 project, where BDD techniques have been applied to verify the i7 processor [2]. In this work, we report a BDD library which is designed to facilitate application of BDD techniques to fully hierarchical systems.

The library works by firstly encoding primitive system components and then repeatedly composing encoded system components using a set of well-defined functions. We assume that primitive system components (e.g., a compositional state which contains no other compositional states, or a process which invokes no other processes) are in the form of finite state machines, which are encoded using BDD in the standard way. In order to build a generally useful library, we take into account different ways of communication between system components: communication through shared memory; synchronous/asynchronous channel communication; and multi-party barrier synchronization (e.g., CSP-style). Next, with process algebras like CSP and CCS in mind, a rich set of system composition functions are supported by the library. Using the provided functions, encoded system components can be composed in a variety of ways, including parallel composition, sequential composition, interrupt, choice, etc. The compositions are based on classic process algebras, but extended to support different kind of system communications. A symbolic encoding of a hierarchical system thus can be gradually obtained from bottom up using the library.

2 BDD and ADD's Operators

Before we represent how we encode expressions and systems, we would like to list some important operators of BDD and ADD supported by the CUDD library. Based on these operators, we extend them to describe more complex structure like If, While.

2.1 Logic Operators

Logic operators include three basic operators: and, or, not. Each operator is a function BDD x BDD $\rightarrow$ BDD. These operators are the same as logic operators for boolean

expressions. The second input may be missing in the case of not operator. For example, the and of 2 BDDs is a new BDD representing the expression which is the result of the and of the 2 corresponding expressions of the two BDD inputs.

2.2 Arithmetic Operators

Arithmetic operators include $plus$, $minus$, $times$, $divide$. Each operator is a function ADD x ADD $\rightarrow$ ADD. Similarly these operators are the same as arithmetic operators for arithmetic expression. For example, the $plus$ of 2 ADDs is a new ADD representing the sum of the 2 corresponding expression of the two ADD inputs.

2.3 Relational Operators

Relation Operators include $less$, $less\_equal$, $equal$, $greater$, $greater\_equal$. Each operator is a function ADD x ADD $\rightarrow$ BDD. At each leaf of the BDD, it is true if the configuration satisfies the corresponding relation expression, and false if vice versa. In other words, the result BDD is the representation of the corresponding relational expression.

3 Encoding Expressions

This section is dedicated to discuss how to encode expressions including variable, arithmetic expressions, logical expressions, and relational expression.

3.1 Encode Variables

Our library only supports the integer type and boolean type. Each variable is described as a sequence of boolean variables in the CUDD. Based on the variable's domain, we will calculate the number of boolean variables required for that variable. For example, the variable x can receive value from 0 to 5. Then the number of values that x can receive is 6. So we need 3 boolean variables $x_0x_1x_2$ to describe the value of x.

Boolean type is a simple case of integer type where the lower bound value is 0 and the upper bound is 1. Our translator supports arithmetic and logic operator like add, subtract, multiply, divide, module, and, or, not.

A variable is represented as an ADD. It is a function from the value of variable's boolean variables to the its corresponding value. Let's take an example. There is a variable whose range is (0, 3). Then it needs 2 boolean variables x_0x_1 to represent its value. Fig 1 represents for the variable a.

3.2 Encode Arithmetic Expressions

The arithmetic expression is in the form of $Expression_1\, optExpression_2$ where opt are in the set $\{add, minus, times, divide\}$. Then the ADD for the result expression equals $ADD_1\, optADD_2$" where ADD_1, ADD_2 are the ADDs of the $Expression_1$, $Expression_2$ respectively. Fig 2 below represents the expression $a + b$ where a, b are integer variable whose range is from 0 to 3. We will used 2 bits x_0x_1 for a and 2 bits x_2x_3 for b.

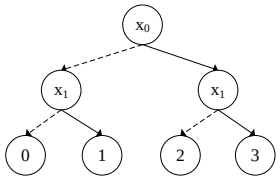

Fig. 1. Example of ADD for variable

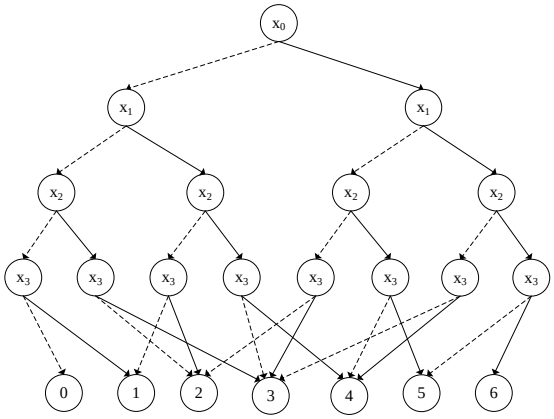

Fig. 2. Example of ADD for arithmetic expression

3.3 Encode Boolean Expressions

Encoding a boolean expression is simpler. The boolean expression is in the form of $Expression_1\, optExpression_2$". The BDD for the result expression equals $BDD_1\, optBDD_2$" where opt is in the set $\{and, or, not\}$, and BDD_1, BDD_2 are the BDDs representing $Expression_1$, $Expression_2$ respectively.

3.4 Encode Relational Expressions

The relational expression is in the form of $Expression_1\, optExpression_2$ where opt is in the set $\{less, less\_equal, equal, greater, greater\_equal\}$. Similarly the result BDD represents the expression which equals $ADD_1\, optADD_2$" where ADD_1, ADD_2 are the ADDs representing $Expression_1$, $Expression_2$. Let's consider the expression a + b $\geq$ 4 where a, b are the variables in the last example. Since the $Expression_2$ is a constant, the encoding is easier. Fig 3 shows the BDD for the a + b $\geq$ 4.

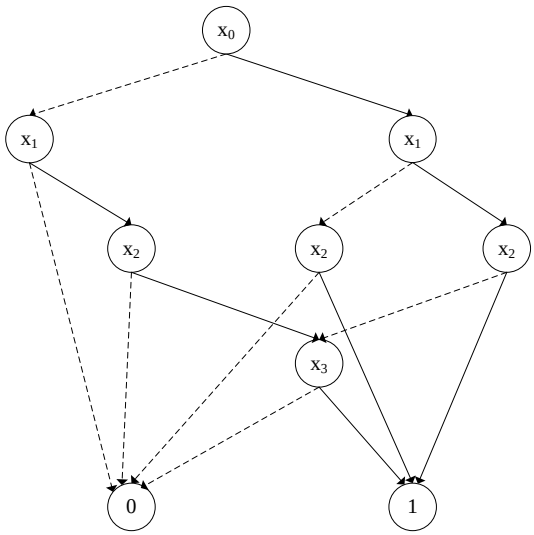

Fig. 3. Example of BDD for relational expression

4 Encode Commands

4.1 Encode Assignment

To encode a transition, each variable needs 2 copies. One represents the variable before the transition and another represents the variable after the transition. Assignments tells how the variables are changed after the action. In other word, assignment is actually the relational expression whose operator is $equal$. The assignment is in the form $variable = value$. The BDD corresponding this assignment equals $ariableADDequalValueADD$ where $VariableADD$, $ValueADD$ are the ADDs of the $variable$ expression on the second copy and the $value$ expression.

4.2 Encode If Command

Basically our library supports encoding variable, arithmetic expression, relational expression, boolean expression, and assignment. With these structures, our library is capable of describing transition, how the variable changed after transition. However, to provide more powerful specification language, our library also support to encode the If, While command to BDD.

The If command is in the form of "If b Then $exp1$ Else $exp2$" (the if command without else is similar). This command will be encoded as $((bBDD\ and\ exp1BDD)\ or\ ((not\ bBDD)\ and\ exp2BDD))$ where $bBDD$ is the BDD of the if condition b, $exp1BDD$ is the BDD of $exp1$, and $exp2BDD$ is the BDD of $exp2$.

4.3 Encode While Command

Supporting While command is more complex. We start with all valuations of variables. At each time we loop the While body part until there is no valuation satisfying the while condition. At each loop, any valuation whose values of variables after the action do not satisfy the while condition will be added to the result. The valuations satisfying the while condition are stilled used in the next loop. Let's take example:
i = 0; while (i $<$ 5) i = i + 2; where i is a variable whose value in the range 0..6. We use the (a, b) to denote the valuations i = a $\wedge$ i' = b. After the first loop, we have 5 valuations $\{$(0,2), (1, 3), (2, 4), (3, 5), (4, 6)$\}$. We add the valuations (3, 5), (4, 6) to the result because the value of i' does not satisfy the while condition. These pairs tell the value of i after exiting the loop based on the value of i before the loop. For example, if the value of i before the while is 3 then the value of i when exiting the loop is 5. Afterward, we continue with the other pairs. At the end of the second loop, we have 3 valuations (0, 4), (1, 5), (2, 6). Similarly we add the valuations (1, 5) and (2, 6) to the result. Last, running the third loop there remains only one valuation (0, 6) which is then added to the result. We will stop the loop since there is no more valuation. So the boolean formula of the above while is "(i = 0 $\wedge$ i' = 6) $\vee$ (i = 1 $\wedge$ i' = 5) $\vee$ (i = 2 $\wedge$ i' = 6) $\vee$ (i = 3 $\wedge$ i' = 5) $\vee$ (i = 4 $\wedge$ i' = 6)".

5 Encoding Hierarchical Systems

This section explains how the library works, i.e., how it encodes primitive system components and how encoded components can be composed in a hierarchical manner.

5.1 Encoding Primitive System Components

Without loss of generality, we assume that a primitive system component takes the form of a *finite state machines*. A finite state machine has finitely many local control states and local variables (with finite domains). A transition is from one local control state to another state, which is labeled with a guard condition (constituted by global/local variables), an optional event and a $transaction$. An event can be a channel input or output, or a (compound) name constituted by local variables as well as global variables. We remark an event (besides channel input/output) can serve as a synchronization barrier (see later on parallel composition). A transaction is a sequential program which possibly updates global/local variables. Note that a finite state machine may communicate with others in different ways.

Finite state machines are encoded using BDD in the standard way. That is, a BDD is used to encode a system configuration (i.e., valuation of variables, channels, etc.) symbolically, e.g., n bits are used to encode K local control states such that $2^n \geq K$. A transition is encoded using two set of Boolean variables $\overrightarrow{x}$ and $\overrightarrow{x}'$, which represent system configurations before and after the transaction. The encoding of transactions is BDDs constituted by $\overrightarrow{x}$ and $\overrightarrow{x}'$. For instance, if the transaction is a simple assignment of the form $y := expr$, then the encoding would denote that variables in $\overrightarrow{x}'$ which encodes y is equivalent to value of $expr$ (based on variables in $\overrightarrow{x}$) and the rest of

$\overrightarrow{x}'$ remains unchanged. Other program constructs like $if\text{-}then\text{-}else$ or $while\text{-}do$ are encoded similarly (refer to [3] for details). An encoded transition is in the form: $g \wedge e \wedge t$ such that g (over $\overrightarrow{x}$) is an encoded guard condition; e is an encoded event and t (over $\overrightarrow{x}$ and $\overrightarrow{x}'$) is an encoded transaction.

A BDD encoding of a finite state machine, which is referred to as a ***BDD machine***, is a tuple $B = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In)$ where $\overrightarrow{V}$ is a set of unprimed Boolean variables encoding global variables, event names and channel names[3]; $\overrightarrow{v}$ is the variables for local variables and local control states; $Init$ is a formula over $\overrightarrow{V}$ and $\overrightarrow{v}$ encoding the initial valuation of the variables; $Trans$ is a set of encoded transitions; Out (In) is a set of encoded transitions labeled with synchronous channel output (input). Note that transitions in Out and In are to be matched by corresponding input/output from the environment.

5.2 Composing BDD Machines

In this section, we show how to compose BDD machines in order to model hierarchical systems. We fix two BDD machines $B_i = (\overrightarrow{V}, \overrightarrow{v}_i, Init_i, Trans_i, Out_i, In_i)$ of process P_i where $i \in \{0, 1\}$ in the following. We assume that $\overrightarrow{v}_0$ and $\overrightarrow{v}_1$ are disjoint (otherwise variable renaming is necessary). Note that $\overrightarrow{V}$ is always shared. The following shows some of the most common composition patterns as examples. Refer to [3] for the complete list.

Parallel Composition The parallel composition of two components B_0 and B_1 is a BDD machine $(\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In)$ such that $v = v_0 \cup v_1$; $Init = Init_0 \wedge Init_1$; and the encoded transitions are defined as follows. $Trans$ contains three kinds of transitions.

- Local transitions: $Trans_i \wedge event' = e \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$ where e is an asynchronous channel input/output or e is an event which is not to be synchronized.
- Synchronous channel communication: $In_i \wedge Out_{1-i}$
- Barrier synchronization: $Trans_i \wedge Trans_{1-i}$

In contains following transitions:

- $In_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$ The channel input of the composition includes the ones of B_i and the local states of B_{1-i} remain unchanged.

Out contains following transitions:

- $Out_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$

Interleave Composition The interleave composition of two components B_0 and B_1 is a BDD machine $(\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In)$ such that $v = v_0 \cup v_1$; $Init = Init_0 \wedge Init_1$. $Trans$ contains two kinds of transitions.

- Local transitions: $Trans_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$
- Synchronous channel communication: $In_i \wedge Out_{1-i}$

[3] Note that $\overrightarrow{V}$ is fixed before encoding the system components.

In contains following transitions:

- $In_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$

Out contains following transitions:

- $Out_i \wedge (\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$

Channel Out Let $B = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In)$ is the BDD machine of $a?exps \rightarrow P_0$. We have $v = v_0 \cup \{temp\}$; $Init = \neg\ temp$. $Trans$ contains the following transition

- $temp \wedge Trans_0 \wedge temp'$

In contains the following transition

- $temp \wedge In_0 \wedge temp'$

Out contains the following transition

- $\neg\ temp \wedge (count_a < L) \wedge [\bigwedge_{i=1..exps.count}(a[top_a][i]' = exps[i])] \wedge (size_a[top_a]' = exps.count) \wedge (count'_a = count_a + 1) \wedge top_a = (top_a + 1)\%L \wedge temp' \wedge Init_0$ where $count_a$ is the number of the elements in the channel buffer, top_a is the position to put new element int the buffer, L is the buffer length of the channel a, and $size_a$ is an array to manage the number of the messages in the buffer. The guard of the channel out transition includes $temp$ is false, and the channel buffer is not full. After the channel in transition, elements from the expression $exps$ is pushed to the buffer. The size of the expression is also updated to $size_a[top_a]$. Moreover the channel buffer updates its size $count_a$ and tail position top_a . $temp$ is set false to constrain the channel out transition to happen once and then pass the control to the process P_0.
- $temp \wedge Out_0 \wedge temp'$

Channel In Let $B = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In)$ is the BDD machine of $[b]a!exps \rightarrow P_1$. We have $v = v_0 \cup \{temp\}$; $Init = \neg\ temp$. $Trans$ contains the following transition

- $temp \wedge Trans_0 \wedge temp'$

In contains the following transition

- $\neg\ temp \wedge b \wedge (count_a > 0) \wedge (size_a[(top_a - count_a)\%L] = exps.count) \wedge [\bigwedge_{i=1..exps.count}(exps[i]' = a[(top_a - count_a)\%L][i])] \wedge (count'_a = count_a - 1) \wedge temp' \wedge Init_0$. The guard of the channel in transition includes $temp$ is false, the guard condition b is satisfied, the channel buffer is not empty and the size of the message in the top of the buffer is equal to the size of the channel in expression. After the transition, variable in the channel in expression is updated with the element in the channel buffer and the buffer also updates its size.
- $temp \wedge In_0 \wedge temp'$

Out contains the following transition

- $temp \wedge Out_0 \wedge temp'$

Unconditional Choice An unconditional choice between B_0 and B_1 is a BDD program $(\overrightarrow{V}_g, \overrightarrow{v}, Init, Trans, Out, In)$ such that $v = v_0 \cup v_1 \cup \{choice\}$ where $choice$ is a fresh Boolean variable, $choice = i$ means B_i is selected; $Init = Init_0 \wedge Init_1$, the variable $choice$ is not initialized and thus B_0 and B_1 can be randomly selected. $Trans$ contains following transition

- $choice = i \wedge Trans_i \wedge choice' = i$ where $i \in \{0, 1\}$

In contains following transition

- $choice = i \wedge In_i \wedge choice' = i$ where $i \in \{0, 1\}$

Out contains following transition

- $choice = i \wedge Out_i \wedge choice' = i$ where $i \in \{0, 1\}$

Sequential The BDD machine of $P0;\ P1$ is $(\overrightarrow{V}_g, \overrightarrow{v}, Init, Trans, Out, In)$ such that $v = v_0 \cup v_1 \cup \{terminated\}$ where $terminated$ is a fresh Boolean variable to manage whether B_0 terminates; $Init = Init_0 \wedge \neg\ terminated$. $Trans$ contains following transition

- $\neg\ terminated \wedge Trans_0 \wedge event' \neq \checkmark \wedge \neg\ terminated'$
- $\neg\ termniated \wedge Trans_0.t \wedge terminated' \wedge Init_1$ where $Trans_0.t$ is created by getting terminative transition of $Trans_0$ then setting the event to tau.
- $terminated \wedge Trans_1 \wedge termniated'$

In contains following transition

- $\neg\ terminated \wedge In_0 \wedge \neg\ terminated'$
- $terminated \wedge In_1 \wedge termninated'$

Out contains following transition

- $\neg\ terminated \wedge Out_0 \wedge \neg\ terminated'$
- $terminated \wedge Out_1 \wedge termniated'$

Interrupt A BDD machine of $P_0 interrupt P_1$ is $(\overrightarrow{V}_g, \overrightarrow{v}, Init, Trans, Out, In)$ such that $v = v_0 \cup v_1 \cup \{terminated\}$ where $terminated$ is a fresh Boolean variable to manage whether B_1 interrupts B_0; $Init = Init_0 \wedge Init_1$. $Trans$ contains following transition

- $\neg\ terminated \wedge Trans_0 \wedge \neg\ terminated' \wedge (\overrightarrow{v}_1 = \overrightarrow{v}'_1)$
- $Trans_1 \wedge termninated'$

In contains following transition

- $\neg\ terminated \wedge In_0 \wedge \neg\ terminated' \wedge (\overrightarrow{v}_1 = \overrightarrow{v}'_1)$
- $In_1 \wedge termninated'$

Out contains following transition

- $\neg\ terminated \wedge Out_0 \wedge \neg\ terminated' \wedge (\overrightarrow{v}_1 = \overrightarrow{v}'_1)$
- $Out_1 \wedge termninated'$

Event Prefix A BDD machine of $e \rightarrow P_0$ is $(\overrightarrow{V}_g, \overrightarrow{v}, Init, Trans, Out, In)$ such that $v = v_0 \cup \{temp\}$ where $temp$ is a fresh Boolean variable to manage whether the event e happens and then pass the control to the process P; $Init = \neg\ temp$. $Trans$ contains following transition

- $\neg\ temp \wedge event' = e \wedge temp' \wedge Init_0$
- $temp \wedge Trans_0 \wedge temp'$

In contains following transition

- $temp \wedge In_0 \wedge temp'$

Out contains following transition

- $temp \wedge Out_0 \wedge temp'$

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.
2. R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In *CAV*, volume 5643 of *LNCS*, pages 414–429. Springer, 2009.
3. T. K. Nguyen, J. Sun, Y. Liu, and J. S. Dong. A BDD Library for Model Checking Hierarchical Systems. Technical report, National Univ. of Singapore, Januray 2011. http://www.comp.nus.edu.sg/˜pat/libreport.pdf.