PAT BDD Library Manual

Truong Khanh Nguyen

School of Computing
National University of Singapore
truongkhanh@comp.nus.edu.sg

1 Introduction

PAT BDD Library is a .Net library for symbolic model checking. It includes a .Net interface to the CUDD package (version 2.4.2). More importantly, it also contains many functions to get BDD encoding of a system easily. A system often consists of many components and primitive components described by transition systems are encoded it first, then these encodings are composed by our supported CSP-like operators.

2 Download and Use

After you download our library and unzip it. The folder will contains 2 DLL files of CUDD interface and the *Source* folder. However you must use the corresponding version of the CUDD interface. For example if your machine is a 32 bit, you need to rename the DLL from 'CUDDHelper 32.dll' to 'CUDDHelper.dll' and can remove the 'CUDDHelper 64.dll'. However Windows 32 bit is recommended to use our library. Then in the *Source* folder is a project which contains the entire source code of our library and another testing project. To use our library, your project must (i) make a reference to our PAT.BDD project and (ii) put the CUDDHelper.dll in the executive folder. Normally you can add this dll to your project and set it to be copied to the executive folder. For more details, try to run PAT.BDD.Release.sln

3 Overview of PAT BDD Library

In this section, we will explain the structure of our library. We will also discuss how to encode from an expression to a system.

3.1 CUDD Library

You can find all of the functions manipulating BDDs in the namespace PAT.Common.Classes.CUDDLib. It includes an interface to CUDD 2.4.2 library and our own friendly interface which helps you to use CUDD library easier. Below are the list of important classes in this namespace.

- Class PlatformInvoke includes all of the necessary functions ported from CUDD. Note that all datastructures in C/C++ are converted to IntPtr in C#. However it is not recommended to call directly these functions.

- Class CUDD provide easier interface to use. General functions are contained in CUDD.cs. Other classes are divided into sub class by its function.
- The corresponding binary decision diagram (BDD) datastructure of CUDD is CUDDNode. It is actually a wrapper of the BDD in CUDD memory with the pointer pointing to the BDD's address. Since CUDD library handles the garbage collection manually, we need to carefully manage the number of references to a certain BDD. Basically a BDD will be removed in the garbage collection process if the its number of references is zero. You can call CUDD.Ref and CUDD.Deref with many overloading functions to increase or decrease the number of references by 1. For each functions, its documentation will show that it changes parameters' number of references. For instance, in the function CUDD.Function.Or, it said '[REFS: result, DEREFS: dd1, dd2]' which means that the parameters $dd1$ and $dd2$ are called Deref while the returned value is called Ref. In other work, the number of references of $dd1$ and $dd2$ are decreased by 1, while the one of the returned value is increased by 1 and actually becomes 1.
- Since CUDD library only works with boolean variables. To encode any variable whose type is finite, we need a set of boolean variables to encode its value. Then CUDDVars will contain these boolean variables encoding a certain variable.
- It is important to know the range of data type we want to encode, specifically the lower bound and the upper bound. Class VariableList will manage variables and their lower bounds and upper bounds.

3.2 Encode a Finite Set

Before giving explanation how to encode an expression, we will briefly describe how to encode a finite set. Essentially given any finite set X, encoding X is to enumerate elements of X in binary and represent them as Boolean functions. Therefore to encode X, we need n boolean variables $x_0, \cdots, x_{n-1}$ where $n = \lceil log_2 |X| \rceil$. Then each element in X is mapped with a bit vector $(x_0, \cdots, x_{n-1})$ by an injective encoding function $f_X : X \to \{0,1\}^n$. Note that this mapping is fixed throughout the BDD encoding. For instance, encoding the set of four elements $X = \{a, b, c, d\}$ requires two boolean variables x_0 and x_1. The encoding functions f_X is defined as $f_X(a) = (0,0)$, $f_X(b) = (0,1)$, $f_X(c) = (1,0)$, and $f_X(d) = (1,1)$. As a result the predicate of the subset $Y = \{a, b\}$ is $((x_0, x_1) = f_X(a) \vee (x_0, x_1) = f_X(b))$. For simplicity we will use the label x to denote the bit vector $(x_0, \cdots, x_{n-1})$. Therefore the predicate of the subset Y can be rewritten shortly as $(x = f_X(a) \vee x = f_X(b))$. Using this technique, we can encode all the data types whose domain is finite, e.g., boolean, integer, array of booleans, and array of integers. To encode transitions, each variable x has another copy called x' which denotes the variable x's value after the transition.

3.3 Encode Expression

Our library provides many types of expressions including boolean expression, arithmetic expression and some program structure like While loop, and If. The result of BDD encoding of any expression is the type of ExpressionBDDEncoding. To get the

BDD encoding of an expression, we can call one of 3 below functions depending on the type of expression:

1. TranslateBoolExpToBDD: You must call this function for a boolean expression. Then the result is a list of BDDs which are OR-implicitly contained in ExpressionBDDEncoding.GuardDDs.
2. TranslateIntExpToBDD: This function is called if your expression is an arithmetic expression. Then the result can be calculated from ExpressionBDDEncoding.GuardDDs and ExpressionBDDEncoding.ExpressionDDs. The encoding can be translated as 'if the guard condition in $GuardDDs[i]$ is true, then the value of the expression is encoded as $ExpressionDDs[i]$. The reason of the returned type ExpressionBDDEncoding is to support array with variables in the index. For example, the encoding of the expression $a[i]$ where $i \in \{1, 2\}$ is the ExpressionBDDEncoding where $GuardDDs$ is a list of 2 BDDs encoding of $i = 1$ and $i = 2$, and $ExpressionDDs$ is a list of 2 BDDs encoding of variables $a[1]$ and $a[2]$.
3. TranslateStatementToBDD: In modeling system, after a transition is taken, variables are updated according to some algorithm. This update can be simple assignment or even a complex program. Our library allows programs to contains statements where they may depend on the previous statements, or programming structures like $While$ loop or If. We provide $TranslateStatementToBDD$ to encode these kinds of programs.

3.4 Encode a Finite State Machine(FSM)

In this section we will define FSM and its encoding. We extend FSM to encode timed systems by using tick-transitions. If you use our library for encoding untimed-systems, you can ignore the part related with $tick$ transition. An FSM is a tuple $\mathcal{M} = (Var, S, init, Act, T)$ such that Var is a set of finite-domain variables; S is a finite set of control states; $init \in S$ is the initial state; Act is the alphabet of events and channels; and T is a labeled transition relation. A transition label is of the form $[guard]evt\{prog\}$ where $guard$ is an optional guard condition constituted by variables in Var; evt is either an event name, a channel input/output or the special $tick$ event (which denotes 1-unit elapsed time); and $prog$ is an optional transaction, i.e., a sequential program which updates global/local variables. A transaction (which may contain program constructs like $while$-do) associated with a transition is to be executed atomically. A non-atomic operation can be broken into multiple transitions. A transition is possible if the $guard$ is true given current valuation σ of Var. Moreover a transition labeled with channel input/output can not occur by itself but must be synchronized with the transition labeled with corresponding channel output/input.

SymbolicLTS is the class to describe an FSM. Similarly State and Transition are used to describe states and transitions in an FSM. You can build an FSM by creating a new SymbolicLTS and adding states, and transitions to it. These classes can be found in namespace PAT.Common.Classes.SemanticModels.LTS.BDD.

The BDD encoding of an FSM, referred to as a *BDD machine*, is a tuple $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$. $\overrightarrow{V}$ is a set of unprimed Boolean variables encoding global variables, event names and channel names, which are fixed for the whole

system before encoding. $\overrightarrow{v}$ is a set of variables encoding 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 the encoding of transitions *excluding synchronous channel input/output and tick-transitions*. Out (In) is the encoding of synchronous channel output (input). Note that transitions in Out and In are to be matched by corresponding transitions in In and Out respectively from the environment and are thus separated from the rest of the transitions. $Tick$ is also the encoding of transitions labeled with $tick$. Then the final transition function of an FSM is taken from $Trans$ and $Tick$. In other words, it can engage an action or idle one time unit. We still calculate Out and In and separate them from $Trans$ and $Tick$ because transitions from Out and In can be useful if they are synchronized.

Let *BDD machine* $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ be the encoding of an FSM $\mathcal{M} = (Var, S, init, Act, T)$ where

- $\overrightarrow{V} = V_1 \cup Events$ where V_1 and $Events = \{event_0, \cdots, event_{n-1}\}$ are the sets of boolean variables to encode global variables and the alphabet Act respectively. Let $event$ denote the bit vector $(event_0, \cdots, event_{n-1})$.
- $\overrightarrow{v} = v_1 \cup States$ where v_1 and $States = \{state_0, \cdots, state_{m-1}\}$ are the sets of boolean variables to encode local variables and the set of states S respectively. Similarly let $state$ denote the bit vector $(state_0, \cdots, state_{m-1})$. Moreover for any global or local variable x, let the same label x denote the corresponding bit vector of boolean variables to encode that variable. Note that these labels x are different. The former x is the variable declared in the model while the latter x is a shorthand for a bit vector in the BDD encoding functions.
- $Init = (state = f_S(init))$
- $Trans = \bigvee(state = f_S(s_0) \wedge g_{bdd} \wedge event' = f_{Act}(e) \wedge prog_{bdd} \wedge state' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with $[g]e\{prog\}$ (where $e \neq tick$). For simplicity, we skip how we encode guard expression g to g_{bdd} and program block $prog$ to $prog_{bdd}$. Interested readers can refer to [**?**].
- $Out = \bigvee(state = f_S(s_0) \wedge g_{bdd} \wedge event' = f_{Act}(e) \wedge prog_{bdd} \wedge state' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with a synchronous channel output e, guarded with g and attached with transaction $prog$.
- $In = \bigvee(state = f_S(s_0) \wedge g_{bdd} \wedge event' = f_{Act}(e) \wedge prog_{bdd} \wedge state' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with a synchronous channel input e, guarded with g and attached with transaction $prog$.
- $Tick = \bigvee(state = f_S(s_0) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_1))$ for all tick transitions from state s_0 to state s_1.

AutomataBDD is the class to describe the structure BDD machine. Moreover this class also contains many compositional functions to achieve the BDD encoding of an untimed-system from the known BDD encoding of sub systems. For timed-systems, you can find these functions in the class TimeBehaviors.

4 A Tutorial for PAT BDD Library

This section will show you how to use our library to do symbolic model checking. Most of the instructions are for our functions. If you want to use CUDD package directly, you should refer to CUDD manual, and CUDD tutorial

4.1 Initialize CUDD

To initialize the CUDD library, we need to call CUDD.InitialiseCUDD. We often initialized with 2GB, $CUDD\_UNIQUE\_SLOTS = 256$, and $CUDD\_CACHE\_SLOTS = 262144$. You can find these constants in the class CUDD. The library can also be initialized from our interface by call Model's constructor function.

4.2 Create Variables

There are two kinds of variables, local variables and global variables. The difference is that for any transition, if global variables are not updated, they still keep the same value while the fact that local variables are not updated means that they are not cared and can receive any value in their range after the transition. You can create variables by calling functions in $Model$: $AddLocalVar$, $AddLocalArray$, $AddGlobalVar$, and $AddGlobalArray$. As our rule, for any array element $A[i]$, its corresponding name is $A + Model.NAME\_SEPERATOR + i$

Since our library is used for model checking purpose, for any variable x created, its prime version x' is also created. Boolean variables representing for variable x are called *row variable* while ones for variable x' are called *column variables*. You can get these boolean variables by using the function $GetRowVars$ and $GetColVars$ in also class $Model$. All row variables and column variables are contained in $Model.AllRowVars$, and $Model.AllColVars$ respectively.

4.3 Create Expression

All of the Expression classes can be found in the name space PAT.Common.Classes.Expressions.ExpressionClass.

- Assignment: Create an assignment statement in the program part of a transition
- BoolConstant: Create boolean constants, $True$ and $False$.
- If: To create if-then-else statement
- IntConstant: To Create integral constant expression
- PrimitiveApplication: This is the most important class which contains many kinds of expression. Refer to the constant strings in this class to know how to create new expressions
- Sequence: This class is used to create the program in a transition. A program is a sequence of statements. Note that there are only 3 kinds of statements which can appear in the program of transition, Assignment, While loop and if-the-else condition.
- Variable: To get a variable by its name.
- VariablePrime: To get variable after the transition (prime version) by its name

- While: To create a while loop
- WildConstant: This is a special expression which do not indicate any specific value. It is often used in an update of a variable which means that after the update, the variable value is not important and can receive any value in its range.

4.4 Create And Encode FSMs

You can create a FSM by creating a new $SymbolicLTS$ and adding new $State$ and $Transition$ to it. Note that FSMs are action-based graph where transitions are labeled with guard condition ($Transition.GuardCondition$), action name($Transition.Event$), and transaction ($Transition.ProgramBlock$). The string $Model.EVENT\_NAME$ is reserved to encode the set of action names. Before starting to encode your model, you need to provide the number of action names used in your model. We also support parameters in action names like $get.i.i$ where i is a certain variable. If your model requires parameters in action names, you need to configure the maximum number of parameters in an action name ($Model.MAX\_NUMBER\_EVENT\_PARAMETERS$) and the lower bound/ upper bound for each parameter in $Model.MIN\_EVENT\_INDEX$, and $Model.MAX\_EVENT\_INDEX$ respectively.

To get the encoding of a system, first you get the SymbolicLTSs of primitive components and encode them. Then you will combine these encoding compositionally by our functions in $AutomataBDD$ and $TimeBehaviors$.

4.5 Search in FSMs

Our library provides many algorithms to search whether it is reachable to state satisfying some condition. This searching can be conducted in forward search, backward search or in both directions. These searching algorithms are in the classes $Graph$ and $GraphTA$. These class also contains a lot of functions to get pre-image and post-image of a set of states.

4.6 Debug

The function $CUDD.Print.PrintMinterm$ is very helpful for you to debug. It will print all of the configurations making the BDD true.