

ECE 510 OCE BDDs and Their Applications

Lecture 10. Advanced Topics in Reachability Analysis

April 27, 2000
Alan Mishchenko

Overview

- Variable ordering for reachability analysis
 - Extension of Malik's heuristics for static ordering
 - Combination of static and dynamic reordering
- Generalized cofactor and its flavors
 - Constraint
 - Restrict
- Improvements to image computation
 - Interleaved variable ordering
 - Iterative squaring
 - Transition relation clustering
 - Recursive image computation

Variable Ordering for Sequential Logic

- Good static variable ordering heuristics are known (Malik's fan-in heuristic for functions given as a net list)
- They can be extended to sequential logic by
 - finding a permutation θ of outputs $F_k(x)$, $1 \leq k \leq n$, of the next-state (output) logic blocks that minimizes the cost function:

$$\sum_{1 \leq i \leq n} | \cup_{1 \leq k \leq i} [\text{Supp}(F_{\theta k}(x))] |,$$
 where $\text{Supp}(F)$ is input support of F and $|S|$ is the set size
 - interleaving the input and output variables as follows:

$$\text{Supp}(F_{\theta 1}(x)) < y_{\theta 1} <$$

$$\text{Supp}(F_{\theta 2}(x)) - \text{Supp}(F_{\theta 1}(x)) < y_{\theta 2} < \dots <$$

$$\text{Supp}(F_{\theta n}(x)) - \cup_{1 \leq k \leq n-1} [\text{Supp}(F_{\theta k}(x))] < y_{\theta n}$$

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

3

Static and Dynamic Variable Ordering

- The success of the algorithms critically depends on the size of BDDs, which are very sensitive to var ordering
- In practice, the combination of static and dynamic variable ordering proved to be helpful
- First, a **good static ordering is found**
- Next, **dynamic reordering** is automatically initiated when one of the two criteria is met:
 - the size of the intermediary BDDs has reached a certain threshold value
 - the size of the intermediary BDDs has increased by a certain value since the last call to reordering

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

4

Generalized Cofactor

- **Generalized cofactor** extends the concepts of “classic” cofactor (cofactor of the given function w.r.t. a cube) to the cofactor w.r.t. an arbitrary function
- Generalized cofactor has a number of flavors
- It serves diverse purposes in BDD-based applications:
 - **constraint** and **restrict** are used to reduce the size of the BDD by exploiting don't cares of the function
 - **constraint** is used to create an alternative image computation procedure

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

5

Generalized Cofactor: Constraint

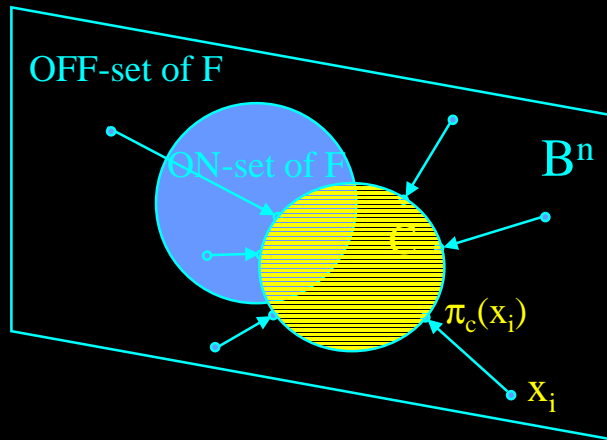
- To define one type of generalized cofactor of **function F** with respect to the **care set C** (the complement of the don't care set of the function), a mapping π (called the **shortest interpretation**) is defined for the domain of **F**:
 - if $C(x) = 1$, $\pi_c(x) = x$
 - if $C(x) = 0$, $\pi_c(x) = \operatorname{argmin}_{y, c(y)=1} [\sum_{1 \leq i \leq n} |x_i - y_i| * 2^{n-i}]$The generalized cofactor **constraint** is defined as
$$F \downarrow C = F(\pi_c(x))$$
- **Constraint** is also known as **image restrictor**, because it reduces image computation to range computation

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

6

Interpretation of Mapping π and $F \downarrow C$

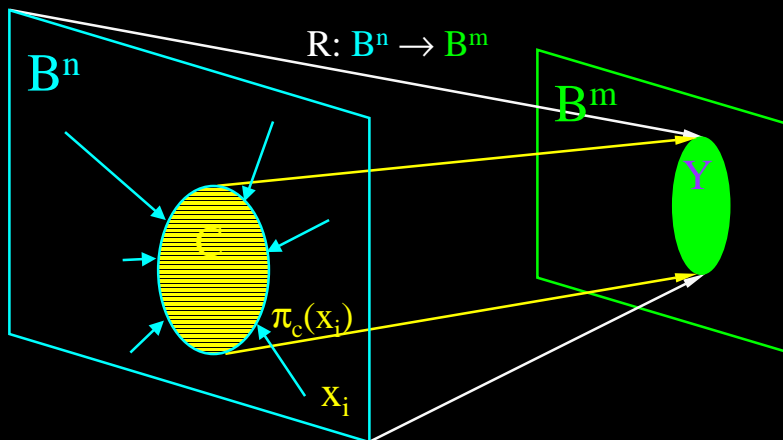


April 27, 2000

ECE 510 OCE: BDDs and Their Applications

7

Interpretation of $Im_R(C) = Im_{R \downarrow C}(B^n)$



April 27, 2000

ECE 510 OCE: BDDs and Their Applications

8

Recursive Computation of Constraint

```
bdd Constraint ( bdd F, bdd C )
{
  assert( c != bddfals );
  if ( C == bddtrue || F == bddtrue || F == bddfals ) return F;
  // check cache for ready results
  int var = bdd_var( C );
  if ( C_{VAR} == bddfals ) return Constraint ( F_{VAR}, C_{VAR} );
  if ( C_{VAR} == bddfals ) return Constraint ( F_{VAR}, C_{VAR} );
  bdd Res = var & Constraint( F_{VAR}, C_{VAR} ) + var & Constraint( F_{VAR}, C_{VAR} );
  // insert the result into cache
  return Res;
}
```

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

9

Generalized Cofactor: Restrict

- It was noticed that constraint is not efficient in reducing the node count of the BDD of the function F when the care set C has a large BDD
- In particular, it happens when C depends on some input variables, on which F does not depend
- The “improved” generalized cofactor (called **restrict**) overcomes this by smoothing (quantifying) away those variables in C that do not belong to the support of F
- As a result, **restrict is more efficient than constraint** in reducing the BDD size
- However, restrict cannot be used as image restrictor

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

10

Recursive Computation of Restrict

```
bdd Restrict ( bdd F, bdd C )
{  assert( c != bddfals );
   if ( C == bddtrue || F == bddtrue || F == bddfals ) return F;
   // check cache for ready results
   int var = bdd_var( C );
   if ( CVAR == bddfals ) return Restrict ( FVAR, CVAR );
   if ( CVAR == bddfals ) return Restrict ( FVAR, CVAR );
   if ( var != bdd_var( F ) ) return Restrict ( F,  $\exists_{VAR} C$  );
   bdd Res = var' & Restrict ( FVAR, CVAR ) + var & Restrict ( FVAR, CVAR );
   // insert the result into cache
   return Res;
}
```

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

11

Interleaved Variable Ordering

- Interleaving input and output variables reduces the BDD size by **exploiting the dependency between codes** of the current state and the next state (it is more noticeable when states that are connected by transitions are encoded using the Grey code)
- Interleaving the variables leads to an **efficient variable replacement procedure**, which allows to substitute next state variables in the image by performing one pass over the BDD representing the image

April 27, 2000

ECE 510 OCE: BDDs and Their Applications

12

Iterative Squaring

- **Iterative squaring** is used to reduce the number of iterations in reachability analysis
- Given the transition relation $R_1(x,y)$, compute the transition relation for states that are reachable in two transitions, four transition, etc. as follows:

$$R_2(x,y) = \exists_z [R_1(x,z) \& R_1(z,y)]$$

$$R_4(x,y) = \exists_z [R_2(x,z) \& R_2(z,y)]$$

- **Iterative squaring** is closely related to the computation of transitive closure

Transition Relation Clustering

- The motivation for **transition relation clustering** is the need to reduce the BDD size for large sequential logic
- It is achieved by exploiting the property of existential quantifier that can be moved across expressions that do not depend on variables being quantified
- Given the clustered transition relation $R(i,x,y) = \prod R_k(i,x,y)$,

$$\begin{aligned} \text{Im}_R(A) &= \exists x,i ([\prod R_k(i,x,y)] \& A(x)) = \\ &= \exists x_n,i_n (R_n(i,x,y) \& \\ &\quad \exists x_{n-1},i_{n-1} (R_{n-1}(i,x,y) \& \dots \\ &\quad \exists x_2,i_2 (R_2(i,x,y) \& \\ &\quad \exists x_1,i_1 ([R_1(i,x,y) \& A(x)])))) \end{aligned}$$

Recursive Image Computation

- This method of recursive computation of the image is based on reduction of image computation to the range computation
- The procedure proceeds by recursively cofactoring w.r.t a variable of the input or the output domain
- Using the abbreviation $rg(f)$ to denote the range of the vector function $f = \{f_1, f_2, \dots, f_n\}$, we get the following:

$$rg(f)(y) = y_1' \& rg(\{f_2, \dots, f_n\}|_{f_1'}) + y_1' \& rg(\{f_2, \dots, f_n\}|_{f_1})$$

or

$$rg(f)(y) = rg(\{f_1, f_2, \dots, f_n\}|_{x_1'}) + rg(\{f_1, f_2, \dots, f_n\}|_{x_1})$$