Model Checking 2nd Edition

Chapter 8: Binary Decision Diagrams and Symbolic Model Checking

Binary Decision Diagrams and Symbolic Model Checking

The model-checking problem and algorithms for CTL

Wang Qirui 1W202047
6/12/2023

Table of Contents

Fairness in Symbolic Model Checking

Extend to include fairness constraints

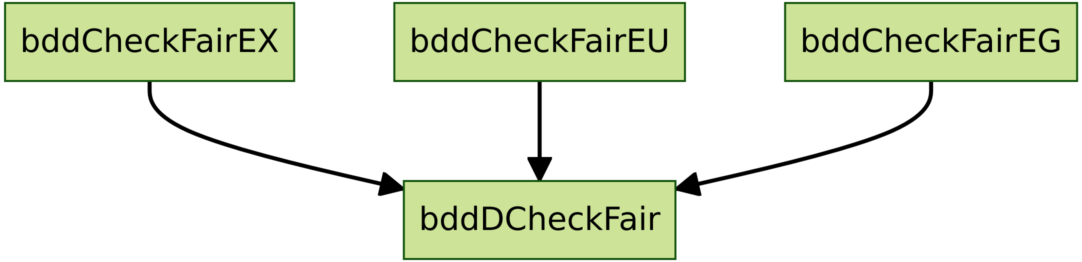
Fairness in Symbolic Model Checking

Introduction

Given fairness constraints as a set of CTL formulas:

$$F = \{P_1, P_2, \ldots, P_n\}$$

A series of procedures for checking CTL formulas relative to the P_is, similar to the procedures with prefix "Check".



Fairness in Symbolic Model Checking

Procedure definitions

Recall that:

$$\mathbf{EG}f = \boldsymbol{\nu Z}.f \wedge \bigwedge_{k=1}^{n} \mathbf{EXE}(f\mathbf{U}(Z \wedge P_k)).$$

Based on that, we can compute the set of states:

$$\begin{aligned}&bddCheckFairEG(f(\bar{v})) = \\ &\quad \boldsymbol{\nu Z}(\bar{v}).f(\bar{v}) \wedge \bigwedge_{k=1}^{n} bddCheckEX(bddCheckEU(f(\bar{v}), \boldsymbol{Z}(\bar{v}) \wedge Check(P_k))).\end{aligned}$$

Each time the above expression is **evaluated**, some fixpoint computations inside $bddCheckEU$ are performed.

Fairness in Symbolic Model Checking

$\mathbf{EX}f$ and $\mathbf{E}(f\mathbf{U}g)$

Checks for $\mathbf{EX}f$ and $\mathbf{E}(f\mathbf{U}g)$ are similar to explicit state model checking. The set of states that are the start of some fair computation is:

$$\mathit{fair}(\bar{v}) = bddDCheckFair(\mathbf{EG}\,\mathit{true}).$$

$\mathbf{EX}f$ is true under fairness constraints in a state s if and only if:

1. there is successor state s'.
2. s' is at the beginning of some fair computation path.

$\mathbf{EX}f$ is equivalent to $\mathbf{EX}(f \wedge \mathit{fair})$ (without fairness constraints). Then define

$$bddCheckFairEX(f(\bar{v})) = bddCheckEX(f(\bar{v}) \wedge \mathit{fair}(\bar{v})).$$

Also, $\mathbf{E}(f\mathbf{U}g)$ is equivalent to $\mathbf{E}(f\mathbf{U}(g \wedge \mathit{fair}))$. So

$$bddCheckFairEU(f(\bar{v}), g(\bar{v})) = bddCheckEU(f(\bar{v}), g(\bar{v}) \wedge \mathit{fair}(\bar{v})).$$

Counterexample and Witness

Counterexamples and witnesses

Introduction

CTL model-checking algorithm is able to find *counterexamples* and *witnesses*.

- Formula with $\mathbf{A}$ is false
  - Find a computation path demonstrating the **negation of the formula** is true
- Formula with $\mathbf{E}$ is true
  - Find a computation path demonstrating **why** the formula is true

For example, if model checker discovers that:

- $\mathbf{AG}f$ is true $\to$ Produce a path to a state which $\neg f$ holds
- $\mathbf{EF}f$ is true $\to$ Produce a path to a state which f holds

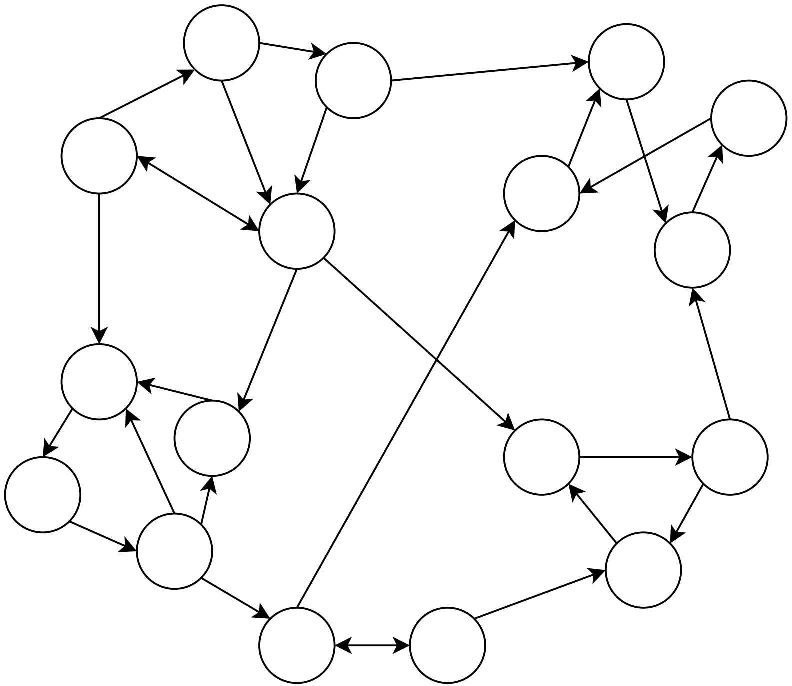
> The *counterexample* for a $\mathbf{A}$ formula is the *witness* for the dual $\mathbf{E}$ formula.

With this, we just need to find witnesses for $\mathbf{EX}$ and $\mathbf{EG}$ and $\mathbf{EU}$.

Finding Witnesses for CTL formula

Explainition of procedure

1. Find the SCCs of the transition graph determined by the Kripke structure.
2. Form a new graph whose nodes are the SCCs and whose edges are the edges between the SCCs in the original graph.
   1. There is no proper cycle in the new graph.
   2. Every infinite path must have **a suffix contained within a SCC**.



Example-1.1

Prev Next

Finding Witnesses for CTL formula

Case for $\mathbf{EG}f$

Find a witness for $\mathbf{EG}f$ under fairness constraints $F = \{P_1, \ldots, P_n\}$.

Recall that:

$$\mathbf{EG}f = \boldsymbol{\nu Z}.f \wedge \bigwedge_{k=1}^{n} \mathbf{EXE}(f\mathbf{U}(Z \wedge P_k)).$$

For a state s in $\mathbf{EG}f$, we want to find a path π:

- starting from s,
- satisfying f in every state,
- visits every set $P \in F$ infinitely often.

It can be constructed using a sequence of prefixes of the path of increasing length **until a cycle is found**.

Finding Witnesses for CTL formula

Case for $\mathbf{EG}f$

1. Evaluate the fixpoint formula. In every iteration of the outer fixpoint computation
   - Compute a collection of least fixpoints associated with $\mathbf{E}(f\mathbf{U}(Z \wedge P))$, for each $P \in F$.
   - For each P, we obtain an increasing sequence of approximations $Q_0^P \subseteq Q_1^P \subseteq \ldots$
     - Q_i^P is a set containing all states, which can reach a state in $Z \wedge P$ in at most i steps while satisfying f.
   - At last iteration, when $Z = \mathbf{EG}f$, save the sequence of approximations Q_i^P for each $P \in F$.
2. Given a state s in $\mathbf{EG}f$, then:
   - It must have a successor in $\mathbf{E}(f\mathbf{U}(\mathbf{EG}f \wedge P))$ for each $P \in F$.
   - Minimize the length of the witness path by choosing the first fairness constraint P that can be reached from s.
     - Looking for a successor of s in $Q_0^P, Q_1^P, \ldots$, for each $P \in F$.
     - Must be found in some Q_i^P.

Finding Witnesses for CTL formula

Case for $\mathbf{EG}f$

2. Continue
   - t has a path of length i to a state in $(\mathbf{EG}f) \wedge P$ so it is in $\mathbf{EG}f$.
   - Then choose any state in intersection of t's successors and Q_{i-1}^P.
   - Repeat until $i = 0$, we get a path from s to some state u in $(\mathbf{EG}f) \wedge P$.
   - Then repeat the process for u and other fairness constraints until all fairness constraints are visited, and let the final state be s'.
3. For the cycle part, we need a non-trivial path from s' to t along with each state satisfying f.
   →That is, a witness for $\{s'\} \wedge \mathbf{EXE}(f\mathbf{U}\{t\})$.
   - If the formula is true, then we found the witness path.
   - Else, the simplest strategy is to restart the procedure from s' using entire F.
     - s' is **in** $\mathbf{EG}f$ but **not in** the SCC of f, which contains t.
     - Eventually find a cycle, or reach a terminal SCC.

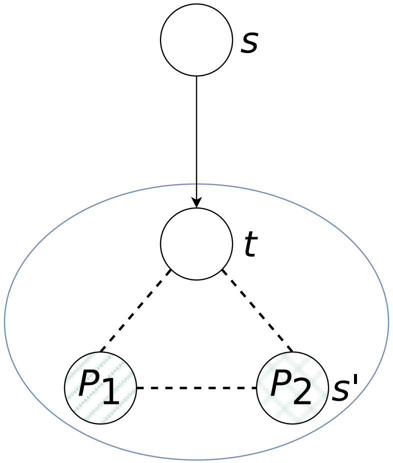


Figure 8.5 Witness is in the first strongly connected component.

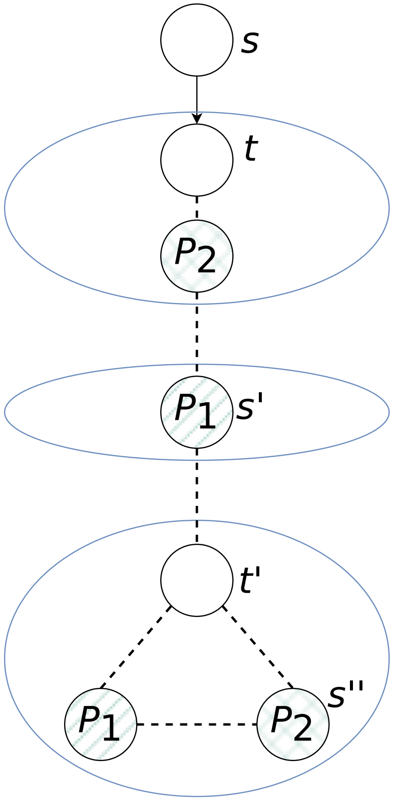


Figure 8.6 Witness spans three strongly connected components.

Finding Witnesses for CTL formula

Case for $\mathbf{EG}f$

- Another way is to **pre-compute** $\mathbf{E}(f\mathbf{U}\{t\})$.
  - The first time we exit the set, we know the cycle cannot be completed.
  - So restart from that state.
- These approaches tend to find short counterexamples but shortest cycle.

For $\mathbf{E}(f\mathbf{U}g)$ and $\mathbf{EX}f$.

> fair denotes the set of states satisfying $\mathbf{EG}\,\mathit{true}$ under fairness constraints F.

- $\mathbf{E}(f\mathbf{U}g)$ under F extends to $\mathbf{E}(f\mathbf{U}(g \wedge \mathit{fair}))$.
- $\mathbf{EX}f$ under F extends to $\mathbf{EX}(f \wedge \mathit{fair})$.

Relational Product Computations

Relational Product Computations

Introduction

Most of the operations used in the symbolic model-checking algorithm are **linear** in the product of the sizes of the operand OBDDs.

- Main exception: relational product operation for $\mathbf{EX}h$:

$$\exists \bar{v}' [h(\bar{v}') \land R(\bar{v}, \bar{v}')]$$

  - It's possible to be implmented with one conjunctioin and multiple existential quantifications.
    - But too slow in practice.
  - The OBDD for $h(\bar{v}') \land R(\bar{v}, \bar{v}')$ is often larger than that for final result.
    - It's better to avoid constructing it.

A special algorithm for relational product in one step.

Relational Product Computations

Special Algorithm for Relational Product

An algorithm for arbitrary OBDDs f and g.

- Cache the results of the recursive calls.
  - With form (f, g, E, r)
- A (f, g, E, r) in the cache means the previous call returns r.
- Exponential complexity in the worst case.
  - OBDD for the result is **exponentially larger** than that for $f(\bar{v})$ and $g(\bar{v})$.
  - In this case, no algorithm can do better.

```
function RelProd(f, g : OBDD, E : set of variables) : OBDD

    if f = 0 ∨ g = 0 then
        return 0;
    else if f = 1 ∧ g = 1 then
        return 1;
    else if (f, g, E, r) is in the result cache then
        return r;
    else
        let x be the top variable of f;
        let y be the top variable of g;
        let z be the topmost of x and y;
        r0 := RelProd(f|z←0, g|z←0, E);
        r1 := RelProd(f|z←1, g|z←1, E);
        if z ∈ E then
            r := Or(r0, r1);
            /* OBDD for r0 ∨ r1 */
        else
            r := IfThenElse(z, r1, r0);
            /* OBDD for (z ∧ r1) ∨ (¬z ∧ r0) */
        end if
        insert (f, g, E, r) in the result cache;
        return r;
    end if

end function
```

Figure 8.7 Relational product algorithm.

Partitioned Transition Relations

Why it is useful?

- The relational product algorithm needs $R(\bar{v}, \bar{v}')$ to be **monolithic transition relation**.
  - Consists of single OBDD. (The construction of this from Kripke structure is shown in 8.2)
  - This OBDD is too large in practice.
- **Partitioned transition relations** provide a much more concise representation.
  - But cannot be used in the relational product algorithm we just saw.

Recall the modeling of synchronous circuits in chapter 3.

- Transition relations are described in form of $\wedge$ or $\vee$ of $R_i(\bar{v}, \bar{v}')$
  - Each piece can be represented by a small OBDD.
  - **Partitioned transition relation**:
    - The model can be represented by a list of OBDDs
      - Imlicitly conjuncted or disjuncted.

Partitioned Transition Relations

Circuit Modeling Example

- For synchronous circuits:

$$R_i(\bar{v}, \bar{v}') = (v_i' \equiv f_i(\bar{v}))$$

  - **Conjunctive partitioned transition relation**.
    - The transition relation is represented by a list of R_i, with implicit conjunction.
- For asynchronous circuits,

$$R_i(\bar{v}, \bar{v}') = (v_i' \equiv f_i(\bar{v})) \wedge \bigwedge_{j \neq i} (v_j' \equiv v_j)$$

  - **Disjunctive partitioned transition relation**.
    - The transition relation is represented by a list of R_i, with implicit disjunction.
    - But the OBDD for R_i may be much larger than that for f_i.
      - up to n times larger, where n is the number of variables used to encode the states.

Partitioned Transition Relations

Techinques for efficient representation

Let

$$N_i(\bar{v}, v_i') = v_i' \equiv f_i(\bar{v})$$

The pair $(N_i(\bar{v}, v_i'), i)$ is used to represent $R_i(\bar{v}, \bar{v}')$.

- Means that v_i' is constrained by N_i, if $j \neq i$, then v_i' is constrained to be equal to v_j.

Then we exploit this during the relational product computation.

$$\begin{aligned} &\exists \bar{v}'[h(\bar{v}') \wedge R(\bar{v}, \bar{v}')] \\ =&\exists \bar{v}'[h(\bar{v}') \wedge (N_i(\bar{v}, v_i') \wedge \bigwedge_{j \neq i}(v_j' \equiv v_j))] \end{aligned}$$

with the equivalent expression

$$\exists v_i'[h(v_i, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n) \wedge N_i(\bar{v}, v_i')].$$

Partitioned Transition Relations

Techinques for efficient representation

Partitioned transition relation with one OBDD per state variable

- is often efficient than constructing a monotlithic transition relation.
- But not always best.
- Better to combine some of the R_i into one OBDD by forming their disjunction or conjunction.
  - May be fewer OBDD nodes if the R_i are combined has similar structure near the root.
  - Combining some of the OBDDs may speed up the relational product computation.

Partitioned Transition Relations

Disjunctive partitioning

For a disjunctive partitioned transition relation, the relational product computed is of the form

$$\exists \bar{v}'[h(\bar{v}') \wedge \bigvee_{i=0}^{n-1} R_i(\bar{v}, \bar{v}')].$$

This can be computed simply distributing the existential quantification over the disjunctions:

$$\bigvee_{i=0}^{n-1} \exists \bar{v}'[h(\bar{v}') \wedge R_i(\bar{v}, \bar{v}')].$$

By this, the computation can be reduced to a series of relational products involving *smaller* OBDDs.

- Large asynchoronous circuits can be verified efficiently than monotlithic transition relations.

Partitioned Transition Relations

Conjunctive partitioning

For a conjunctive partitioned transition relation, the relational product computed is of the form

$$\exists \bar{v}' [h(\bar{v}') \wedge \bigwedge_{i=0}^{n-1} R_i(\bar{v}, \bar{v}')].$$

But the existential quantification cannot be distributed over the conjunctions.

There are two observations:

1. Circuits exhibit locally, many R_i depend on only a few variables.
   - Combine for a dependence on multiple primed variables sometimes advantageous.
2. Subformulas **do not rely on the variables being quantified** can be moved outside.

Based on these, we can

- Conjunct the $R_i(\bar{v}, \bar{v}')$ with $h(\bar{v}')$.
- Early quantification of variables that are not being depended on.

Partitioned Transition Relations

Conjunctive partitioning

Take the modulo 8 counter example in chapter 3, recall that

$$\begin{aligned} R_0(\bar{v}, v_0') &= (v_0' \Leftrightarrow \neg v_0) \\ R_1(\bar{v}, v_1') &= (v_1' \Leftrightarrow v_0 \oplus v_1) \\ R_2(\bar{v}, v_2') &= (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) \end{aligned}$$

The relational product for $\mathbf{EX}h$ is

$$\exists v_0' \exists v_1' \exists v_2' [h(\bar{v}') \wedge (R_0(\bar{v}, v_0') \wedge R_1(\bar{v}, v_1') \wedge R_2(\bar{v}, v_2'))].$$

And can be rewritten as

$$\exists v_2' \exists v_1' \exists v_0' [\Big((h(\bar{v}') \wedge R_0(\bar{v}, v_0')) \wedge R_1(\bar{v}, v_1') \Big) \wedge R_2(\bar{v}, v_2')].$$

- Subformulas can be moved outside the scope of existential quantification.
  - If they don't depend on variables being quantified.

Partitioned Transition Relations

Conjunctive partitioning

Then we can re-express the relational product as

$$\exists v_2' \left[\exists v_1' \left[\exists v_0' [h(\bar{v}') \wedge R_0(\bar{v}, v_0')] \wedge R_1(\bar{v}, v_1') \right] \wedge R_2(\bar{v}, v_2') \right].$$

So this can be computed by starting with $h(\bar{v}')$, at each step:

1. Combine the previous result with $R_i(\bar{v}, \bar{v}')$.
2. Quantify out the appropriate variables.

Thus, the computation has been reduced to a series of small steps.

> The intermediate results may depend both on variables in $\bar{v}$ and on variables in $\bar{v}'$.

Partitioned Transition Relations

Conjunctive partitioning

Principles for reordering the conjuncts:

- Variables in $\bar{v}'$ can be quantified out early.
- Variables in $\bar{v}$ can be added slowly.

This reduce the number of variables in the intermediate OBDDs, and thus the size of the OBDDs.

- Computing the RP for $\mathbf{EX}h$ is computing the predecessor of a set of states.
- Sometimes we need to compute the successor of a state set.
  - Quantify out the present state variables.

The RP for a successor computation has the form:

$$\exists v_0 \exists v_1 \exists v_2 [h(\bar{v}) \wedge (R_0(v_0, \bar{v}') \wedge R_1(v_0, v_1, \bar{v}') \wedge R_2(v_0, v_1, v_2, \bar{v}'))].$$

> Unprimed variables are written explicitly, while primed variables are left implicit.

Partitioned Transition Relations

Successor computation

Then it can be rewritten as

$$\exists v_0 \exists v_1 \exists v_2 \big[\Big(\big(h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')\big) \wedge R_1(v_0, v_1, \bar{v}')\Big) \wedge R_0(v_0, \bar{v}')\big].$$

And then as

$$\exists v_0 \Big[\exists v_1 \big[\exists v_2 [h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')] \wedge R_1(v_0, v_1, \bar{v}')\big] \wedge R_0(v_0, \bar{v}')\Big].$$

In this case

- The number of new state variables v_i' is **independent** of the ordering
- The number of old state variables v_i remaining at each stage **depends** on the ordering.
  - Minimized by reordering.

Partitioned Transition Relations

Generalize successor computation

This method can be generalized to arbitrary **conjunctive partitioned transition relation** with n state variables.

- A user-defined permutation ρ of $\{0, 1, \ldots, n-1\}$.
  - Determines the order in which the partitions $R_i(\bar{v}, \bar{v}')$ are combined.
- For each i, let D_i be the set of variables v_i' that $R_i(\bar{v}, \bar{v}')$ depends on.
- Let

$$E_i = D_{\rho(i)} - \bigcup_{k=i+1}^{n-1} D_{\rho(k)}.$$

  - The set of variables contained in $D_{\rho(i)}$ but not in any $D_{\rho(k)}$ for $k > i$.
  - Pairwise disjoint, and their union contains all the variables.

Partitioned Transition Relations

Generalize successor computation

So the RP for $\mathbf{EX}h$ can be computed as

$$
\begin{aligned}
h_1(\bar{v}, \bar{v}') &= \exists_{v'_j \in E_0} [h(\bar{v}') \wedge R_{\rho(0)}(\bar{v}, \bar{v}')] \\
h_2(\bar{v}, \bar{v}') &= \exists_{v'_j \in E_1} [h_1(\bar{v}, \bar{v}') \wedge R_{\rho(1)}(\bar{v}, \bar{v}')] \\
&\vdots \\
h_n(\bar{v}, \bar{v}') &= \exists_{v'_j \in E_{n-1}} [h_{n-1}(\bar{v}, \bar{v}') \wedge R_{\rho(n-1)}(\bar{v}, \bar{v}')].
\end{aligned}
$$

The final result is h_n.

If some E_i is empty, then

$$h_{i+1}(\bar{v}, \bar{v}') = [h_i(\bar{v}, \bar{v}') \wedge R_{\rho(i)}(\bar{v}, \bar{v}')]$$

and no existential quantification will be used.

Partitioned Transition Relations

Generalize successor computation

The ordering ρ is essential to

- How early state variables can be quantified out.
- The size of the OBDDs constructed.
- The efficiency of the verification.

A good ordering ρ can be searched by

- Using a greedy algorithm to find a good ordering on the variables v_i to be eliminated.
  - There is an obvious ordering on the relations R_i.
    - Variables can be eliminated in the order given by the greedy algorithm.

Partitioned Transition Relations

Algorithm for variable elimination

while $V \neq \phi$ **do**
    For each $v \in V$ compute the cost of eliminating v;
    Eliminate variable with lowest cost by updating $\mathcal{C}$ and V;
end while

Figure 8.7 Relational product algorithm.

1. Start with
   - The set of variables V
   - A collection $\mathcal{C}$ of sets where every $D_i \in \mathcal{C}$ is the set of variables that R_i depends on.
2. Eliminate the variables one at a time.
   - Always choose the variable with the least cost.
   - Update V and $\mathcal{C}$ appropriately.

Partitioned Transition Relations

Cost metric

For convenience, we define

- R_v refers to the relation created when
  1. Eliminating v by taking the conjunction of all the R_i that depend on v
  2. And then quantifying out v.
- D_v refers to the set of variables that R_v depends on.

There are three cost mesures:

1. **Minimum size**: The cost of eliminating a variable v is $|D_v|$.
   - Always trying to ensure that the new relation depends on the fewest number of variables.

Partitioned Transition Relations

Cost metric

2. **Minimum increase**: The cost of eliminating a variable v is

$$|D_v| - \max_{D \in \mathcal{C}, v \in D} |D|.$$

   - Prefer to increase the size of an *already large relation* rather thab create a new one.

3. **Minimum sum**: The cost of eliminating a variable v is

$$\sum_{D \in \mathcal{C}, v \in D} |D|.$$

   - Cost of conjunction depends on the size of the arguments.
   - Approximates it by the sum of R_i dependencies.

Partitioned Transition Relations

Summary to cost metric

- The goal is to minimize the size of the largest BDD created during variable elimination.
  - Find an ordering that minimizes the largest set D_v.
- Locally optimal dose not imply optimal solution.
  - Every cost function has a counterexample.
  - Finding optimal ordering is ***NP-complete***.
- **Minimum sum** seems to be the best choice in practice.
  - With better performance.

Recombining Partitions

Summary

We showed

- A synchronous circuit could be represented by a set of transition relations $R_i(\bar{v}, \bar{v}')$, each depending on exactly one variable in $\bar{v}'$.
- Combining some of the R_i into one OBDD can obtain a **smaller** representation.
- Combining parts of a R **speeds up** the computation of the relational product.

For example, consider a n-bit counter.

- Under usual ordering, the number of nodes in the OBDD is $\mathcal{O}(n)$.
  - Both monolithic and fully partitioned.
- For a $h(\bar{v}')$ representing single state, the computation of relational product for
  - Fully partitioned: $\mathcal{O}(n^2)$
    - It requires n OBDD operations, each of which takes $\mathcal{O}(n)$ time.
  - Monolithic: $\mathcal{O}(n)$