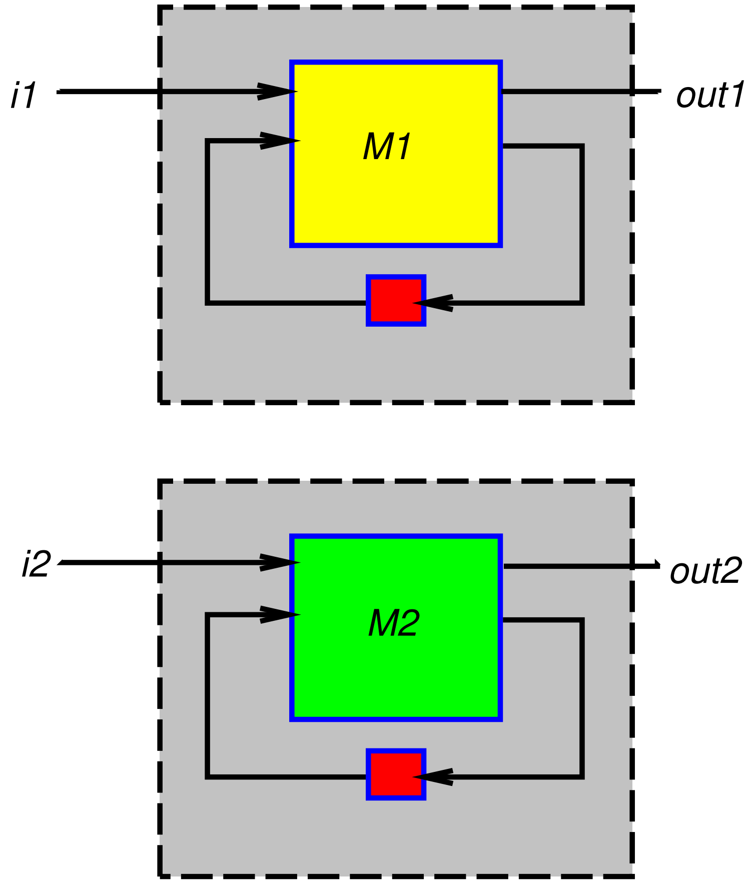
Verification of FSM Equivalence

Goal: Verify that two sequential circuit implementations produce the same outputs given the same inputs



Does $out1 = out2$ for all possible input sequences?

Why are state enumeration techniques important?

- State traversal techniques are essential to:
  - synthesis of sequential circuits
  - verification of sequential circuits

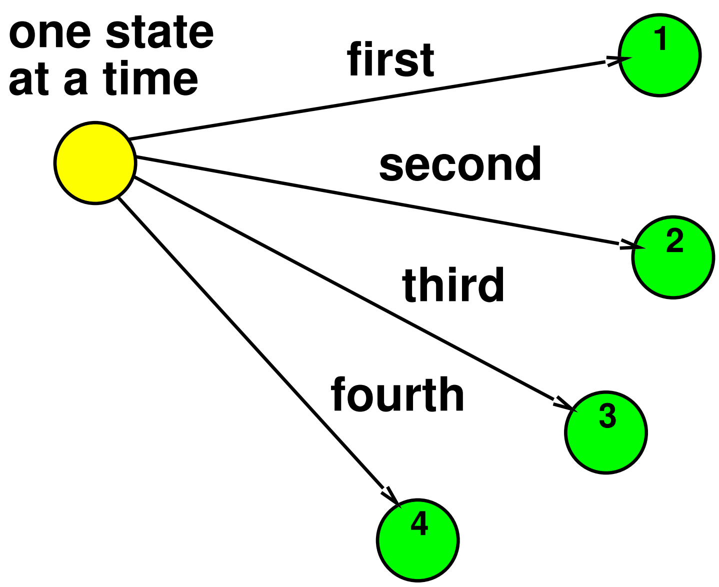
such as:

- sequential testing [Cho, Hachtel and Somenzi ITC'91]
- extraction of sequential don't care
- implementation verification:
  - checking sequential equivalence [Coudert and Madre ICCAD'90] [Touati, Lin, Brayton and Sangiovanni ICCAD'90]
- design verification:
  - checking language containment of ω-regular languages [Touati, Kurshan and Brayton 1991]
  - CTL model checking [Burch, Clarke, McMillan and Dill DAC'90]

Explicit FSM Traversal

Explicit enumeration

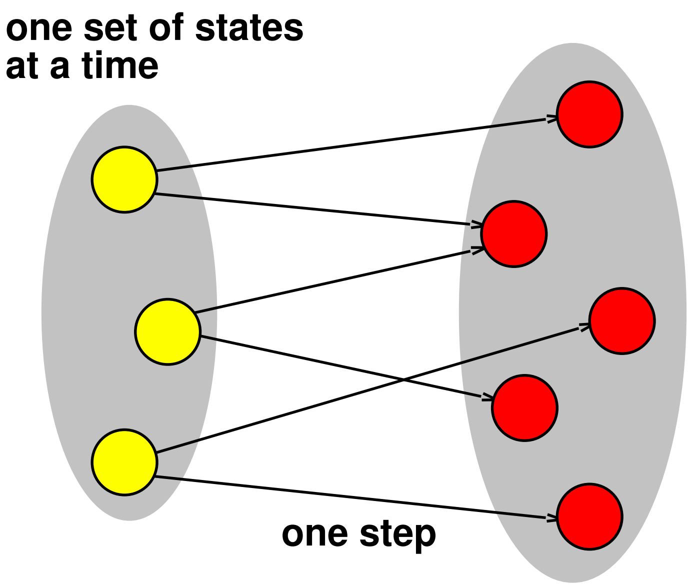
- process one state at a time
- for each state, next states are enumerated one by one
- complexity depends on the number of states and the number of input values.
- methods that enumerates the states space explicitly cannot handle FSM's with a large number of states



Implicit FSM Traversal

Implicit enumeration

- manipulates set of states directly in a breath-first search manner
- exponential speed-up over explicit enumeration methods (sometimes)
- exploits the fact that the number of states does not usually reflect system complexity
- implicit state enumeration may take a long time for sequentially deep circuits e.g. those with counters



Transition and Output Relations

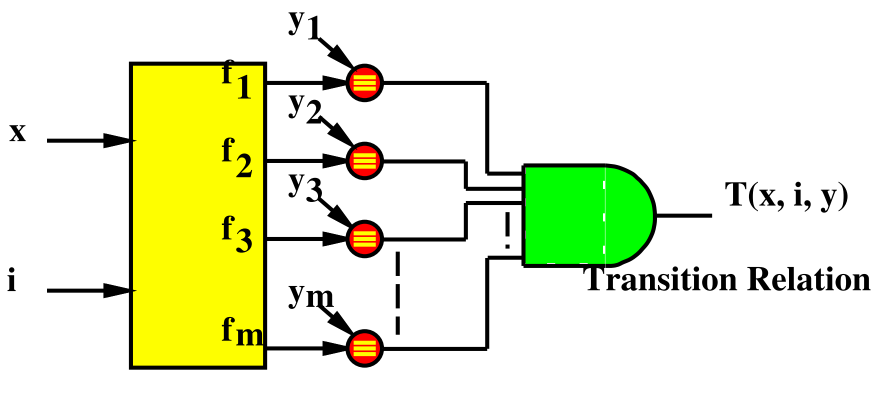
- Let x be the present state, i the inputs, y the next state, and o the outputs.
- Transition function vector: $f(x, i) = [f_1(x, i), \ldots, f_m(x, i)]$.
- Output relation:

  $O(x, i, o) = 1 \leftrightarrow$ output is o at state x and input i
- Transition relation:

  $T(x, i, y) = 1 \leftrightarrow y =$ next state from state x on input i.

  Note: transition relation can represent non-deterministic FSM but transition functions cannot.

$$T(x, i, y) = \prod_{k=1}^{m} (y_k \equiv f_k(x, i))$$



Range, Domain, Image and Inverse Image

- $R(f)$ =**range** of f and $D(f)$ = **domain**.
  $f(x)$ is a unique point $y \in R$, but $f^{-1}(y)$ is in general a set.

- The **image** of a function $f(x)$ with respect to a domain constraint $C(x)$ is denoted

  $$Img(f, C) = \{y | \exists x \in C : y = f(x)\}.$$

  In terms of relation $T(x, i, y)$ and BDD operators:

  $$Img(T, C)(y) = \exists x(C(x) \cdot T(x, i, y))$$

- The **inverse image** of a function $f(x)$ with respect to a domain constraint $C(y)$ is denoted

  $$Pre(f, C) = \{x | \exists y \in C : y = f(x)\}.$$

  In terms of relation $T(x, i, y)$ and BDD operators:

  $$Pre(T, C)(x) = \exists y(C(y) \cdot T(x, i, y)).$$

- $Img(T, C)$ and $Pre(T, C)$ are sets represented by BDD's.

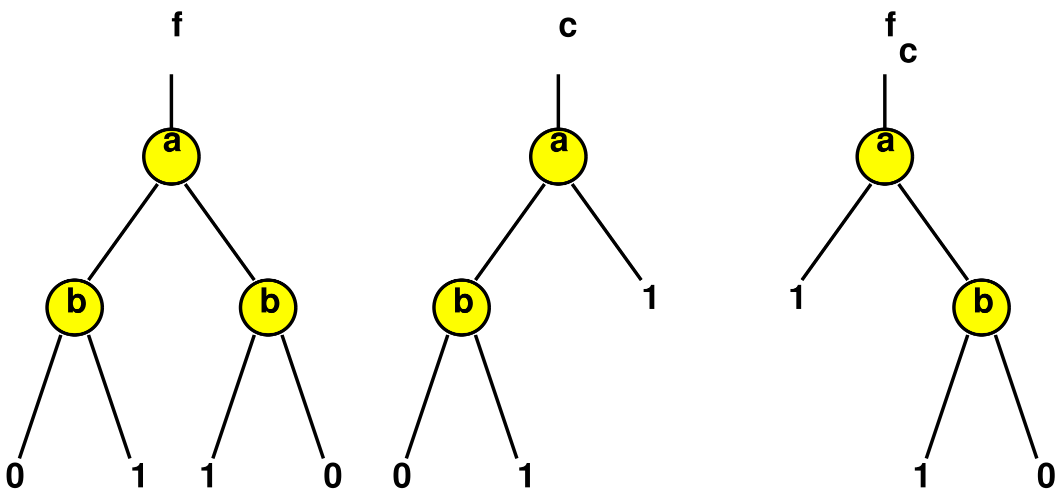
Generalized Cofactor

Recall:

- If c is a cube, the Shannon cofactor is the cover of the generalized cofactor cof(f, c) that is independent of the variables of the cube c.
- There are many other functions which cover cof(f, c).
- Want: heuristic to select a representative function of cof(f, c) that has a small BDD representation.
- The **constrain** operator [Coudert ICCAD'90] is a representation of generalized cofactor which, in most cases, has a small BDD representation.
- When c is a cube, constrain operator gives the same result as the usual Shannon cofactor.

Constrain Operator

function **constrain**(f, c)
    assert$(c \neq 0)$
    if $(c = 1$ or **is_constant**$(f))$ return f
    else if $(c_{\overline{x}_i} = 0)$, return **constrain**(f_{x_i}, c_{x_i})
    else if $(c_{x_i} = 0)$ return **constrain**$(f_{\overline{x}_i}, c_{\overline{x}_i})$
    else return x_i **constrain**$(f_{x_i}, c_{x_i})+\overline{x}_i$ **constrain**$(f_{\overline{x}_i}, c_{\overline{x}_i})$
}



Restrict Operator

function **restrict**(f, c)
    assert$(c \neq 0)$
    if $(c = 1$ or **is_constant**$(f))$ return f
    else if $(c_{\overline{x}_i} = 0)$, return **restrict**(f_{x_i}, c_{x_i})
    else if $(c_{x_i} = 0)$ return **restrict**$(f_{\overline{x}_i}, c_{\overline{x}_i})$
    else if $(f_{x_i} = f_{\overline{x}_i})$, return **restrict**$(f_{x_i}, c_{x_i} + c_{\overline{x}_i})$
    else return x_i **restrict**$(f_{x_i}, c_{x_i}) + \overline{x}_i$ **restrict**$(f_{\overline{x}_i}, c_{\overline{x}_i})$
}

Restrict has the property that if f is independent of x_i then restrict(f, c) is also. However, it does not have the other nice properties of constrain. Experiments show that generally, restrict is better than constrain in terms of producing a smaller BDD.

Properties of Constraint Operator

Following properties of the constrain operator make it very useful in implicit image computations. We denote **constrain**(f, c) by f_c.

1. $(g \circ f)_c = g \circ f_c$
   In particular, $(f + g)_c = f_c + g_c$ and $(\overline{f})_c = (\overline{f_c})$.

2. $\exists x \; f(x, y) \cdot c(x) = \exists x \; f_c(x, y)$

3. c is contained in f iff f_c is a tautology.

4. If c is the characteristic function of a set $A \subseteq B^n$, then $f_c(B^n) = f(A)$. i.e. $R(f_c) = Img(f, A)$

5. Given $\mathcal{F}(f, d, r)$, $f_{\overline{d}}$ usually has a smaller BDD representation than f.

Constrain Operator: Another Perspective

- Let $c : B^n \rightarrow B$ be a non-null Boolean function, and $x_1 \prec x_2 \prec \ldots \prec x_n$ an ordering of its input variables. The mapping π_c is defined as follows:

$$\pi_c(x) \quad = \quad \begin{cases} 1 & \text{if} \quad c(x) = 1 \\ y & \text{if} \quad c(x) = 0, \text{ where} \\ & \qquad y = \text{argmin}_{c(y)=1} \sum_{i=1}^{n} |x_i - y_i| 2^{n-i} \end{cases}$$

- π_c is a projection that maps a minterm $x \in B^n$ to a minterm y in the ON-set of c, which has the closest distance to x, according to the distance metric d.

- The particular metric used guarantees the uniqueness of y in the definition above.

- Since π_c maps all points in B^n into the ON-set of c, then $f_c(x) = f(\pi_c(x))$. If $f = (f_1, \ldots, f_m)$, then $f_c = ((f_1)_c, \ldots, (f_m)_c)$.

Fixed Point Iteration via Predicate Transformers

- Let (p, q) be two predicates on a finite set S. Let F be a predicate transformer, i.e. $F(p) = r$ where r is a new predicate on S. Suppose F is monotone:

  $$p \subset q \rightarrow F(p) \subset F(q)$$

  We associate a sequence $\{p_i\}$ as follows:

  $$\begin{aligned} p_0 &= \emptyset \\ p_{i+1} &= F(p_i) \\ p_\infty &= p_i \text{ if } p_{i+1} = p_i \end{aligned}$$

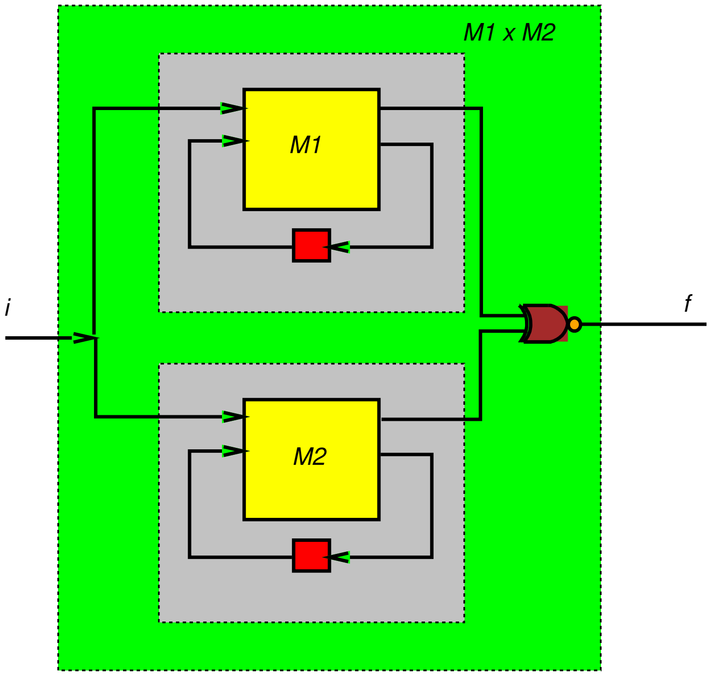
  p_∞ is called the **least fixed point** of the predicate transformer F.

- Note that $p_i \subseteq p_{i+1}$, since $\emptyset \subseteq F(p_0) \rightarrow F(\emptyset) \subseteq F(F(\emptyset))$. Since S is finite $(p_i)_{i \geq 0}$ converges after a finite number of iterations.

- If $p_0 = S$, the corresponding iteration gives the **greatest fixed point** of F.

- All sets can be represented by BDD's, and the predicate transformer $F(p_i)$ can be computed using BDD operations.

FSM Equivalence on the Product Machine



- Form the product machine:

$$
\begin{aligned}
x &= (x_1, x_2) \\
y &= (y_1, y_2) \\
i &= i_1 = i_2 \\
Init(x) &= (Init_1(x_1) \times Init_2(x_2)) \\
T(x, i, y) &= T_1(x_1, i, y_1) \cdot T_2(x_2, i, y_2) \\
f(x, i) &= (f_1(x_1, i) \equiv f_2(x_2, i))
\end{aligned}
$$

where $\equiv$ denotes XNOR operation.

Implicit State Enumeration

$$
\begin{aligned}
T(x,i,y) &= T_1(x_1,i,y_1) \cdot T_2(x_2,i,y_2) \\
f(x,i) &= (f_1(x_1,i) \equiv f_2(x_2,i))
\end{aligned}
$$

- The following fixed point algorithm (using a monotone predicate transformer) computes the set of **reachable state pairs** $R(x)$ of the product machine:

  $$
  \begin{aligned}
  R_0(x) &= Init(x) \\
  R_{k+1}(x) &= R_k(x) + [y \to x]\exists x,i\ (R_k(x) \cdot T(x,i,y))
  \end{aligned}
  $$

  where $[y \to x]$ represents substitution of x for y.

- The iteration terminates when $R_{k+1}(x) = R_k(x)$ and the least fixed point is reached.

- The FSM's are equivalent iff for all inputs, their outputs are identical on all reachable state pairs. Equivalence is verified by the following predicate:

  $$\forall x \in R(x),\ \forall i,\ f(x,i) \equiv 1$$

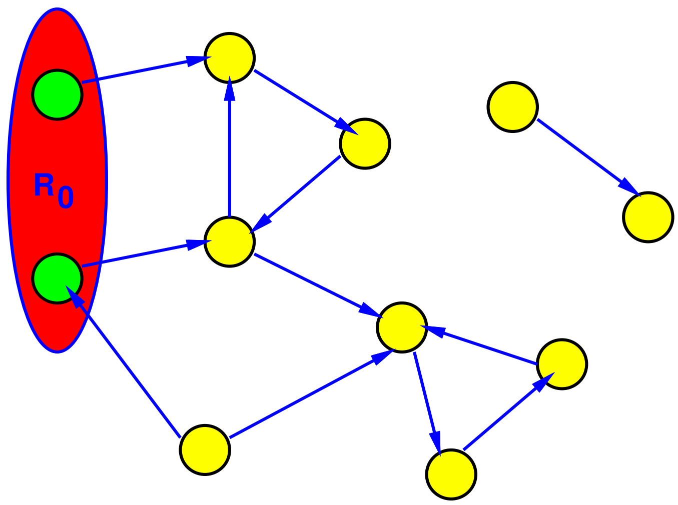
  Easy computation with BDDs. Simply compute

  $$\overline{R}(x) + f(x,i)$$

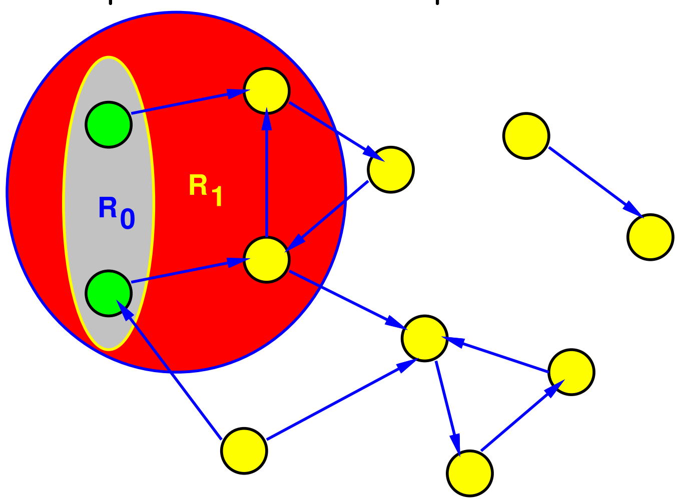
  and check if this is identically 1.

Finding Reachable States in an FSM

R_0 is the set of initial states

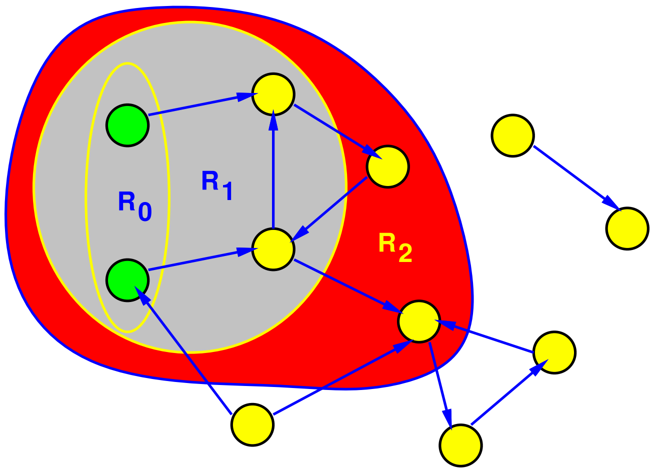


R_1 is the set of states reachable from the initial states in less than or equal to one steps.

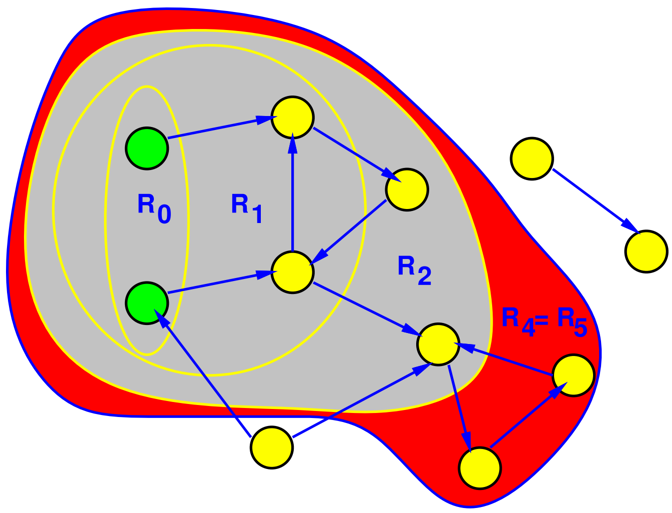


Finding Reachable States in an FSM (cont.)

R_2 is the set of states reachable from the initial states in less than or equal to two steps.



The fixed-point iteration terminates after finding $R_4 = R_5$ and the resultant set is the **set of reachable states**.



Partial Product Method

- Do not need to compute the transition relation $T(x, i, y)$ explicitly as a single monolithic BDD.
- Can keep $T(x, i, y)$ in product form (called "partitioned translation relation"):

$$\exists x, i(T(x, i, y) \cdot R_k(x)) \quad = \quad \exists x, i \prod_{1 \le j \le n} (y_j \equiv (f_j))R_k(x)$$

Using constrain

- Can form the **constrain** with respect to $R_k(x)$ of each $(y_j \equiv f_j)$ before forming the final AND. This usually reduces the size of the final AND.

  $$\exists x, i \prod_{1 \leq j \leq n} (y_j \equiv (f_j)) R_k(x) = \exists x, i \prod_{1 \leq j \leq n} (y_j \equiv (f_j))_{R_k(x)}$$

- The final AND operation can be decomposed into a balanced binary tree of Boolean ANDs and intermediate variables can be quantified out as soon as no other product depends on those variables.

Using restrict

- Can pre-cluster groups of $T_j(x_j, i_j, y_j)$ (defined as $(y_j \equiv f_j(x_j, i_j)))$ together

$$C_j = \exists i_{k_j}(T_{j_1}, \ldots T_{j_k})$$

where the i_{k_j} that we can quantify out here are the ones that do not appear in any other T_l.

- Using constrain is a bad idea since $(C_j(x_j, i_j, y_j))_{R_k(x)}$ may result in a BDD with more variables. It is better to use **restrict**).

- Linearly order (heuristicaly) the C_j and multiply from the right by $R_k(x)$ one at a time, quantifying variables x_q, i_q as soon as they appear in no other terms.)

$R_{i+1} = R_i(x) + [y \to x]\exists x_p, i_p C_{l_1}(...(\exists x_q, i_q C_{l_q} P_i(x)) \ldots)$

where $P_i(x) = \text{restrict}(R_i, \overline{R_{i-1}})$.

Traversal by Recursive Image Computation

- The method uses CONSTRAIN which transforms an image computation into a range computation.
- Let $g : B^n \to B^m$, $g = [g_1, g_2, \ldots, g_m]$, and $A \subseteq B^n$, and c be the characteristic function of A.
- First step: use the CONSTRAIN operator to form $f = g_c = [(g_1)_c, (g_2)_c, \ldots, (g_m)_c]$.
- Second step: the range of f is determined. Let $\mathcal{R}(f)$ denote the range of the function f
- Using Shanon's decomposition, the range computation can be written recursively as:

  $$\mathcal{R}(f)(y) = y_1\mathcal{R}([f_2, \ldots, f_m]_{f_1}) + \overline{y}_1\mathcal{R}([f_2, \ldots, f_m]_{\overline{f}_1})$$

- The efficiency of the procedure can be improved by *caching* the results of intermediate computations. Also, if at any step, the functions $[f_1, \ldots, f_m]$ can be grouped into sets with disjoint support, then the range computation can proceed independently on each set.