(Lec05) BDDs Applied: Finite State Machine Verific

△ What you know
  ▶ Representations: Cube lists, BDDs
  ▶ Manipulations: URP attacks on cubes, implementation for BDDs
  ▶ Useful computations
    ▶ Are these 2 blocks of logic doing the same thing for all inputs?
    ▶ Build a BDD for each and see if they are identical

△ What you don’t know
  ▶ Cool ways people apply BDDs out in real world
  ▶ Example: Verifying logic with any kind of time-varying behavior
  ▶ Important application: Finite State Machines
    ▶ I give you 2 different FSM implementations
    ▶ You tell me: are they behaviorally equivalent...
    ▶ ...ie, for same input stream, will they make identical outputs?
  ▶ This is a whole new problem...

Copyright Notice

© Rob A. Rutenbar, 2001
All rights reserved.
You may not make copies of this material in any form without my express permission.
Where Are We?

Something really new, made possible by BDDs:  Formal Verif.

<table>
<thead>
<tr>
<th>M</th>
<th>T</th>
<th>W</th>
<th>Th</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug 27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>Sep 3</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>Oct 1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nov 5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Thnxgive 19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>Dec 3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

Handouts

Physical

- This lecture -- Lec05 Formal Verification

Electronic

- Project 1 (will be shortly…) on the web site. In project 1, we give you the skeleton of a BDD package in JAVA, and you get to complete it, and then try to apply it to a portfolio of common gate-level logic test/verification problems.
- HW2 is also (still out) on the web site. HW2 covers lectures 3, 4 (BDD basics and internals) but not lecture 5 (FSM verification).
  - Some HW2 bug fixes will also appear shortly

© R. Rutenbar 2001, CMU 18-760, Fall 2001
Readings

- De Micheli
  - Doesn’t really do much on formal verification

- Bryant's Symbolic Analysis Paper from Comp Surveys
  - Does a lot of this material...
  - ...but it’s very dense. Go thru the lecture, work thru all the algebra, go back and look at Bryant's paper.

Some Terminology

- Verification
  - You give me a logic implementation of some design, and some specification you guarantee is correct
  - I figure out -- somehow -- whether the implementation is correct

- Strategy: Simulation
  - Validate that the logic implementation works for all the inputs that you simulate
  - Problem: it might NOT work for an input you DON’T simulate!

- Strategy: Formal Verification
  - Prove that there DOES NOT exist an input (or a sequence of inputs over time) that causes the logic to make a wrong answer
  - Or, FIND an input that causes it to make a wrong answer
**Formal Verification: Who Cares...?**

Intel, for one...

- They simulated the divider a whole lot
- They still missed some inputs that made errors
- Result, the Pentium FDIV bug, lots of bad press, lots and lots of money lost
- Formal verification techniques are now capable of finding errors like these in complex designs

---

**RAR’s Amazing (and Coincidental) Link to This...**

- Dan Atkins:
  - In 1972 becomes Prof at U Michigan, and later Dean of Engineering.
  - His last Phd student: Rob Rutenbar

Intel FDIV Bug:
- Intel, in a mistaken attempt to save Si area on a Pentium, over-simplifies a crucial piece of logic in Atkins’ High-Radix Quotient-digit algorithm.
- Result: “the Pentium bug”
Formal Verification: This Lecture

- Pick one significant problem: FSM verification
  - Review finite state machines (from basic digital designs)
  - Show why this is a hard problem
  - Show a clever attack on the verification problem that exercises a lot of what you know about BDDs and their capabilities

- New stuff here
  - Dealing with temporal behavior: we want to know the FSM works for all patterns of inputs, over all future clock ticks
  - Representing this temporal behavior using Boolean functions
  - Reasoning about this behavior
  - Turning the whole shebang into a sequence of symbolic computations you could do with C code + a good BDD package

FSM Verification: Reminders

- What’s in an FSM?
  - States -- unique bit pattern represents each state; FFs store state
  - External inputs: inputs from the outside world
  - Next state logic: from current state and external inputs, this makes the inputs to the FFs that determine the next state at the next clock tick
  - Output logic: from states (and maybe the inputs) makes combinational outputs for the FSM
  - Clock: synchronizes everything; only states changes on clock tick


FSM Verification: What’s the Problem?

Several scenarios

- You have a “trusted” implementation of the machine that you know is correct (but maybe not optimal as hardware) and a “real” implementation as gates. Are they the same?
- You have an “old” implementation in one technology, and a “new” implementation in another technology (e.g., a different tech library). Is old == new?

Why is this hard?

- Because of what “same” means here
- Has to do with behavior over time -- this temporal dependence is new for us...
- We need to be more precise

Equivalent FSMs

Means this:

- Start them in some known “equivalent” states; clock starts running
- For every possible combination of inputs over future clock ticks, two machines will have identical outputs
- (Doesn’t say anything about logic delays or low level stuff like that; just think about ideal clock ticks here...)
FSM Equivalence: Easy Case

Sometimes this isn’t too hard: special case

- Suppose 2 FSMs have identical state encodings: same #bits, same unique bit pattern for each state, same kinds of FFs
- Then this reduces to combinational equiv. checking

Combinational Equivalence Checking

Reminder

- This is the easiest “formal verification” sort of problem
- You have 2 different implementations of the same function
- They have identical input and output variables
- Your task: determine if they give identical outputs over all possible inputs, or find a counterexample where the outputs differ

Why is this easy?

- Build a BDD for each function. If they are identical, you get the identical same BDD pointer for each one.
- If not identical, build BDD for (function F) ⊕ (function G) and find satisfying inputs for this new function. These inputs make F != G!
- Easy since it uses all the standard BDD stuff.
- No notion of time in here, no sequences of inputs over clock ticks
FSM Equivalence: Hard Case

FSMs not always so easy to check. More general case:
- 4 states. 2 state variables p,q. 1 input x. 1 output z. D FFs.

State transitions:

<table>
<thead>
<tr>
<th>State</th>
<th>Input x</th>
<th>pq=00</th>
<th>p+q+</th>
<th>NS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B/0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C/0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D/1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Input logic:

<table>
<thead>
<tr>
<th>Input x</th>
<th>p+q+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Output logic:

<table>
<thead>
<tr>
<th>Output z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FSM1 Implementation

It looks like this...

State

<table>
<thead>
<tr>
<th>State</th>
<th>D Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Q</td>
</tr>
<tr>
<td>D</td>
<td>Q</td>
</tr>
</tbody>
</table>

Logic

- Next State Logic
- Output Logic
Let's implement it as a 1-hot machine

- Now, 4 state variables: abcd; again D FFs, but now 4 of them

State: abcd = 0001 abcd = 0010 abcd = 0100 abcd = 1000

A/0

B/0

x = 0

x = 1

x = 0, 1

x = 1

C/0

D/1

abcd = 0010 abcd = 0001

Can just read off the NS logic

I hot example transition...

It looks like this

State

D Q

D Q

D Q

D Q

Logic

Next State Logic

Output Logic
FSM Equivalance Checking

This is why it’s hard
- Can’t just look at the logic now and say “yeah, they’re the same”
- Need a whole new, systematic method that deals with temporal aspect of things here, and the possible differences in encodings

FSM 1 Logic
- Next State Logic
  \[ p^+ = pq' + p'x \]
  \[ q^+ = p'q + p'x' \]
- Output Logic
  \[ z = pq \]

FSM 2 Logic
- Next State Logic
  \[ a^+ = d \]
  \[ b^+ = ax' + bx' \]
  \[ c^+ = ax + c \]
  \[ d^+ = bx \]
- Output Logic
  \[ z = d \]

FSM Formal Verification Strategy

4 Big Ideas

Sets as Boolean functions
- Use BDDs to represent sets of things

Symbolic representation of FSMs
- Represent them as sets of allowable transitions

Reachability analysis
- Represents the sets of FSM states you can get to from the start state on 0 clock ticks, 1 clock tick, 2 ticks, etc.

Cross-product FSMs
- Take 2 FSMs you want to compare for equivalence, and make 1 single new special machine, on which reachability analysis == verification
1. Sets as Boolean Functions

- We already saw this idea on an early HW
  - Suppose your objects in your sets are: a b c d e f
  - Assume these are now Boolean vars; var = 1 means “an object in set”
  - Represent set as function; value = 1 for patterns that == one obj in set

\[
\begin{array}{ccccccc}
 a & b & c & d & e & f & S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Since it’s a Boolean function, we can also represent this as a BDD, even for very large sets

Sets as Boolean Functions

- BDD representation lets you do some neat stuff
  - Suppose A(a,b,c,d,e,f), B(a,b,c,d,e,f) are sets represented as BDDs

\[
\begin{array}{ccc}
\text{What is } A \cup B? & \text{What is } A \cap B? \\
\text{Is } A \subseteq B? & \text{Is } A \text{ the empty set?} \\
\end{array}
\]
2. Symbolic Representation of FSMs

Idea is to represent the set of all legal transitions

Legal transitions
FROM ON Input TO

Symbolic FSM Representation

How do we do this

- First, what NOT to do: can't do it by enumerating all transitions and "adding logic" to represent each one
- A machine with 100 FFs + 10 inputs has approx. $1000 \times 2^{100}$ transitions!

What do we want?

- A new Boolean function, called $\delta(\ )$, the "transition relation"

$$\delta(\text{state vars for current state, input vars, state vars for next state}) = \begin{cases} 0 & \text{if transition not legal} \\ 1 & \text{if transition is legal} \end{cases}$$

Let's look at some examples
Transition Relation Examples

Look at FSM 1: it will be \( \delta(p, q, x, p^+, q^+) \)

\[
\begin{align*}
\delta(p=0, q=0, x=0, p^+=0, q^+=1) &= \\
\delta(p=0, q=0, x=1, p^+=1, q^+=0) &= \\
\delta(p=1, q=1, x=1, p^+=1, q^+=1) &= \\
\delta(p=0, q=1, x=1, p^+=0, q^+=1) &= 
\end{align*}
\]

OK, now FSM 2: it will be: \( \delta(a,b,c,d,x,a^+,b^+,c^+,d^+) \)

\[
\begin{align*}
\delta(a=0, b=0, c=0, d=1, x=0, a^+=0, b^+=0, c^+=0, d^+=1) &= \\
\delta(a=1, b=0, c=0, d=0, x=0, a^+=0, b^+=1, c^+=0, d^+=0) &=
\end{align*}
\]
**Constructing the Transition Relation**

What do we want?  a BDD for $\delta(\cdot)$
- There is a great, simple trick here
- The next state logic is already most of the logic you need

```
```

```
```

**About the Transition Relation**

- Bryant reports that, as a BDD, it can be very big
  - Much bigger than then next state logic itself, which is inside of it
  - Various tricks to get around having the build the whole thing

For us though, we'll just press on
- Knowing how to do this in the most straightforward, “frontal assault” is fine for now.
About the Transition Relation

- One subtlety: May be some "bogus" stuff in $\delta$
  - What happens if not all bit patterns are legal states?
  - Will find also all the $\delta$( illegal old state, input, illegal new state) too
  - $\delta$ tells you everything about FSM. Ignores legal start state assumptions.

State: $abcd=1000 \, abcd=0100$

Legal = 1000, 0100, 0010, 0001
All others illegal

Next State Logic

$z = d$

Output Logic

$z = d$

3. Reachability Analysis

- What's the idea?
  - Build a Boolean function that represents all the FSM states you can reach in up to $K$ clock ticks, starting from a known initial state at clock tick 0
  - Sets called "reachability" sets: $R_0, R_1, R_2, \ldots R_K$
  - Example of what we expect to happen.
Reachability Analysis

Observations

- $R_k$ is a function only of the state variables, since all you want to know is if the state bit pattern you input can be reached from the start state in not more than $k$ clock ticks.

- $R_0$ is easy to compute, since there is only 1 state in it, the assumed initial (reset) state for the FSM.

The hard part: going from $R_k$ to $R_{k+1}$

- What do we know to start? Ex: FSM 1
  - Transition relation: $\delta(p, q, x, p^+, q^+)$ (we have a BDD for this)
  - $R_0 = R_0(p^+, q^+) = (p^+) (q^+)$ (we have a BDD for this too)

- What do we want to do?
  - Compute $R_1(p^+, q^+)$ using $R_0$ and $\delta(p,q,x,p^+,q^+)$
  - Then, compute $R_2(p^+, q^+)$ using $R_1(p^+, q^+)$ and $\delta(p,q,x,p^+,q^+)$
  - Then $R_3(p^+, q^+)$ from $R_2(p^+q^+)$ and $\delta(p,q,x,p^+,q^+)$ ...
  - ...keep going until $R_{k+1} = R_k$ for some $k$, and we are done
  - Note: easy to tell if $R_k = R_{k+1}$ given BDDs!

Called iterated reachability analysis
Iterated Reachability Analysis

Mechanically, how?

- Need to look close at the structure of the R sets
- Think about sets like Venn diagrams: what's inside what?
- If you know $R_k$, what else is there to get to $R_{k+1}$...

\[ R_k \quad R_{k+1} \]

Observation #1

Observation #2

© R. Rutenbar 2001, CMU 18-760, Fall 2001
So, for FSM1 example we can write \( R_k \) like this:

\[
R_{k+1}(p^+, q^+) = R_k(p^+, q^+) + \text{NEWTRAN}(p^+, q^+)
\]

Focus on the hard part here: what is “NEWTRAN” here?

Consider this function, made of pieces we know:

\[
R_k(p, q) \cdot \delta(p, q, x, p^+, q^+)
\]

What conditions make this function == 1?
Iterated Reachability Analysis

OK, this guess is close, but...
- Why can't $[R_k(p, q) \cdot \delta(p, q, x, p^+, q^+)]$ be the piece we want?
- It has to depend only on the variables $p^+, q^+$
- In other words, we want a function that == 1 just for the new state patterns that are reachable in 1 transition from $R_k$...
- ... we don't care from which exact state, or with which input

\[
R_{k+1}(p^+, q^+) = R_k(p^+, q^+) + [R_k(p, q) \cdot \delta(p, q, x, p^+, q^+)]
\]

Quantification to the rescue!
- (Another reason we keep hammering this elusive yet powerful idea...)

Say in English what we want:

\[
[R_k(p, q) \cdot \delta(p, q, x, p^+, q^+)]
\]
Iterated Reachability Analysis

▶ OK, now say it precisely, *mathematically*

\[ R_{k+1}(p^+, q^+) = R_k(p^+, q^+) + \left\{ (\exists \ p, q, x) \ [ R_k(p, q) \cdot \delta(p, q, x, p^+, q^+) \right\}(p^+, q^+) \]

General solution for FSM 1

\[ R_{k+1}(p^+, q^+) = R_k(p^+, q^+) + \left\{ (\exists \ p, q, x) \ [ R_k(p, q) \cdot \delta(p, q, x, p^+, q^+) \right\}(p^+, q^+) \]

In general...

\[ R_{k+1}(\text{next state vars}) = \]

\[ R_k(\text{next state vars}) \]

\[ + \left\{ (\exists \ \text{current state, inputs}) \ [ R_k(\text{current state vars}) \cdot \delta(\text{current state, inputs, next state}) \right\} \]
Iterated Reachability Analysis: BDD Subtlety

Look close at $R_k$ functions

$$R_{k+1}(p^+, q^+) = R_k(p^+, q^+) + \{(\exists p, q, x) \ [ R_k(p, q) \delta(p, q, x, p^+, q^+) \}] (p^+, q^+)$$

This is the same function $R_k(\ )$, but with 2 different sets of input variables

When you make the BDDs, you get 2 separate BDDs here, one representing $R_k(p^+, q^+)$, and the other representing $R_k(p, q)$

You can do this kind of thing with the “composition” op from the homework

Iterated Reachability Analysis

Randy Bryant suggests we draw it like this...

Actually think about it again like a big piece of hardware
Iterated Reachability: Example FSM1

What's $\delta(p, q, x, p+, q+)$?

What's $R_0$? Assume start state is A

Example: FSM1

Do the messy quantification part of the eqn

$$R_1(p+, q+) = R_0(p+, q+) + \{ (\exists p, q, x) [ R_0(p, q) \cdot \delta(p, q, x, p+, q+) ] \}$$

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$x$</th>
<th>$[p'q'] \cdot [p+ \overline{p}(pq' + p'x)] \cdot [q+ \overline{q}(p'q + p'x')]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$(\exists p, q, x) [ R_0(p, q) \cdot \delta(p, q, x, p+, q+) ] = OR all 8 of these$
Example: FSM1

- So, it works!
  - Result is $R_1(p^+, q^+) = (p^+ \cdot q^+) + (p^+ \cdot q^+) + (p^+ \cdot q^+)$
  
- If you do it again, and compute $R_2$, will find == 1 (do it!). Why?
- Also find all subsequent $R$'s == $R_2$. Why?

4. Cross Product Machine

- Where are we?
  - We can compute a Boolean function (represented as a BDD) that tells us all states in the FSM reachable in at most $K$ ticks of the clock
  - Can do this mechanically using BDDs, and the next state logic

- Where do we want to be?
  - Given 2 different FSMs, can we tell if they are equivalent

- Now what?
  - Make a new FSM that combines the 2 FSMs we which to check for equivalence....
  - ...and we do the construction so that reachability analysis gives us the verification that we want.
  - Construction == cross-product machine
Given 2 FSMs to check...

- Just FSM1, FSM2, with FSM2's states given new names...

Cross Product Machine

State: pq=00 pq=01
A/0 x=0
B/0 x=0
x=0,1 C/0 pq=10
D/1 pq=11

Next State Logic
p
q
x

Output Logic
z = pq

FSM 1 Logic

State: abcd=1000 abcd=0100
abcd=0010 abcd=0001
A'/0 x=0
B'/0 x=0
x=0,1 C'/0 abcd = 0010
D'/1 abcd=0001

Next State Logic
a+ = d
b+ = ax' + bx'
c+ = ax + c
d+ = bx

Output Logic
z = d

FSM 2 Logic

Cross Product Machine: FSM1 × FSM2

- What are the states in FSM1 × FSM2? (× == “cross”)

- All pairs of possible states (FSM1 state, FSM2 state)

State: pq=00 pq=01
A/0 x=0
B/0 x=0
x=0,1 C/0 pq=10
D/1 pq=11

State: abcd=1000 abcd=0100
abcd=0010 abcd=0001
A'/0 x=0
B'/0 x=0
x=0,1 C'/0 abcd = 0010
D'/1 abcd=0001

FSM1 × FSM2 has 4×4=16 states
What are the state variables for FSM1 \( \times \) FSM2

- Easy, just “all” the state vars from FSM1 and from FSM2

\[ \text{State} \]
\[ \begin{array}{llll}
\text{DQ} & p \\
\text{DQ} & q \\
\text{DQ} & a \\
\text{DQ} & b \\
\text{DQ} & c \\
\text{DQ} & d \\
\end{array} \]

\[ \text{Logic} \]
\[ \begin{array}{l}
\text{Next State Logic} \\
\quad ??? \\
\text{Output Logic} \\
\quad ??? \\
\end{array} \]

So, what then is the next-state logic for FSM1 \( \times \) FSM2?

- Easy again: just the 2 separate next-state blocks from FSM1, FSM2

\[ \text{State} \]
\[ \begin{array}{llll}
\text{DQ} & p \\
\text{DQ} & q \\
\text{DQ} & a \\
\text{DQ} & b \\
\text{DQ} & c \\
\text{DQ} & d \\
\end{array} \]

\[ \text{Logic} \]
\[ \begin{array}{l}
\text{Next State Logic} \\
\quad \text{FSM1 Next State Logic} \\
\quad \text{FSM2 Next State Logic} \\
\text{Output Logic} \\
\quad ??? \\
\end{array} \]
Cross Product Machine: FSM1 × FSM2

- So, what is the output logic for FSM1 × FSM2
  - Now, it's different.
  - The cross-product machine is really (up to now) just the 2 separate machines FSM1 and FSM2 running side-by-side.
  - What we really want to know is, if we start them both in the same “equivalent” state, will we ever reach a “combined” state (FSM1 state vars, FSM2 state vars) where the outputs of the 2 machines are actually different.

![Diagram of FSM1 × FSM2](image)

Cross Product Machine: FSM1 × FSM2

- So, what is the output logic for FSM1 × FSM2?

<table>
<thead>
<tr>
<th>State</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Q p</td>
<td>Next State Logic</td>
</tr>
<tr>
<td>D Q q</td>
<td>FSM1 Next State Logic</td>
</tr>
<tr>
<td>D Q a</td>
<td>FSM2 Next State Logic</td>
</tr>
<tr>
<td>D Q b</td>
<td>Output Logic</td>
</tr>
<tr>
<td>D Q c</td>
<td>FSM1 Output Logic</td>
</tr>
<tr>
<td>D Q d</td>
<td>FSM2 Output Logic</td>
</tr>
</tbody>
</table>

Should always be == 1 if FSM1 = FSM2
Cross Product Machine Reachability Analysis

Where are we?

- We can build, mechanically, the cross product machine

What good is this?

- FSM1 and FSM2 are equivalent if, when started in the same initial “equivalent” states, it is **NEVER POSSIBLE** to reach a state where the output of the cross product machine == 0
- Put another way: output of cross product machine should always be ==1, for any combined state (FSM1 state, FSM2 state) we can reach from (FSM1 start, FSM2 start)

How do we do this?

- We already know how...

---

Cross Product Reachability Analysis

Build the cross product machine

- Means build the next state logic, and the output logic for it

Do reachability analysis on [ FSM1 × FSM2 ]

- In our little example, start with A=00, A' = 1000 states:

  \[ R_0(p^+,q^+,a^+,b^+,c^+,d^+) = \]

  Build \( R_1, R_2, \ldots, R_K \) until it stops changing
  - At this point, you know pairs of states it is possible to reach from (A, A') start state...
Cross Product Reachability Analysis

What does this tell you?

- Pairs of states you can get to from \((A, A')\)
- Example

Example: 2 ticks, \(A \rightarrow B \rightarrow D\)

Example: 2 ticks, \(A' \rightarrow B' \rightarrow B'\)

So, in \(R_0\) we have \((A, A')\), and in \(R_2\) we will have...

Verification via Reachability

OK, we can build \(R_k\) for this \(FSM1 \times FSM2\)

Now what?

- Want to know if we can reach a combined state where cross-product output \(==0\); consider this logic...

Output Logic

\[ R_k \]

\[ p+ q+ a+ b+ c+ d+ \]

==1 if \((p+ q+, a+ b+ c+ d+)\) is a reachable pair

Equality

\[ ==1 \] if output from \(p+ q+\) same as output from \(a+ b+ c+ d+\)
Verification via Reachability

Look what happens

- Build a BDD for this ckt; it has 6 inputs \( p^+ q^+ a^+ b^+ c^+ d^+ \), has 1 output
- If the 2 FSMs are equivalent, then this output \( \equiv \) constant 0 BDD
- If these 2 FSMs NOT equivalent, this is some BDD with a 1 node in it
- All we have to do is see if this BDD \( \neq \) constant 0 BDD, and we are done!

Verification via Reachability

What did we do here?

- We turned the complex temporal problem of verification of equivalence for all inputs over all subsequent clock ticks...
- ...into a series of BDD exercises, ending in a satisfiability check on a single (big, nasty) BDD; if any pattern of inputs makes this BDD \( \equiv \) 1, then machines not equivalent

Amazingly cool result

- Doable mechanically with a good BDD package
- Definitely NOT something you want to try to do by hand.
- On the HW, you get to do it for a similar pair of machines using the KBDD package to do the nasty Boolean manipulations
Simpler Example

A/0
x=0
p=0

B/1
x=1
p=1

State p=0

Next state:
p⁺ = p+x

Output
z = p

FSM 1

x=0
x=1
x=0,1

C/0
x=0
q=0

D/1
x=1
q=1

State q=0

Next state:
q⁺ = q ⊕ x

Output
z = q

FSM 2

Simple Example: Cross Product Reachability

R₀ = { (A,C) }

R₁ = reachable on 0 or 1 clock tick =

R₂ = reachable on 0, 1, 2 clock ticks =

R₃ = reachable on 0, 1, 2, 3 clock ticks =
Cross Product Reachability Analysis

- So, we get $R_k = \{ AC, BD, BC \}$ ... what is Boolean func?
- OK, let’s make the cross product satisfiability logic

Output Logic

- $R_k$
  - $p^+$, $q^+$ = 1 if $(p^+, q^+)$ reachable
  - Output Logic
    - FSM1: $z = p$
    - FSM2: $z = q$
  - $= ?$
    - $= 1$ if output from $p^+$ same as output from $q^+$

Cross Product Reachability: Satisfiable?

- Well, can you get a 1 out of this...?
Cross Product Reachability: Satisfiable?

Aha!
- Pattern p=1 q=0 satisfies this BDD, makes it \(=1\)
- Since you can get a \(1\) here, FSMs NOT equivalent
- What does \(p=1\), \(q=0\) signify here?

Summary

- FSM verification is important, but different
  - Have to worry about time, about equivalent outputs over all possible future inputs
  - Not just simple block-to-block combinational equivalence
- Big ideas
  - Symbolic representation of FSMs -- transition relation
  - Reachability analysis
  - Cross product machine with “special” satisfiability output
  - Transforms the temporal problem into another series of do-able BDD exercises
- Hugely important application of BDD analysis out in the real world these days...