Fundamental Algorithms for System Modeling, Analysis, and Optimization

Stavros Tripakis
UC Berkeley
EECS 144/244
Fall 2016

Synchronous Composition

Thanks to Edward A. Lee for several slides

Composition of discrete systems

Two major paradigms:

Synchronous:

All subsystems move together, in "lock-step".

Application domains: synchronous circuits, embedded control, …

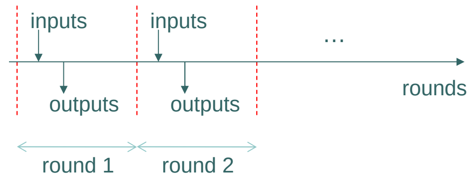
Asynchronous:

Each subsystem moves at its own pace: interleaving.

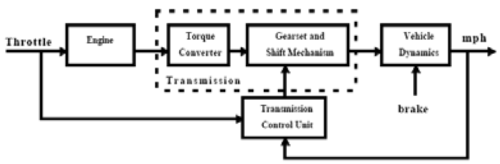
Application domains: concurrent software, distributed systems, …

Fundamental characteristic of synchronous systems

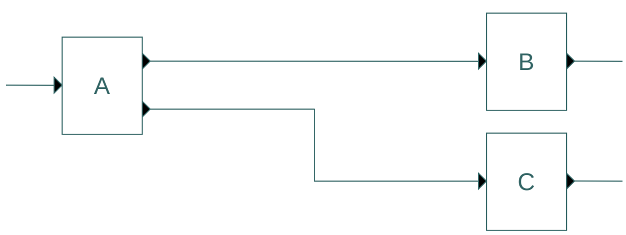
Notion of ***synchronous round*** (or ***cycle***, or ***reaction***)

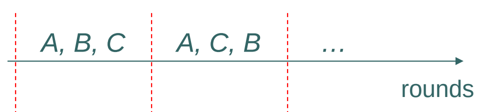


All subsystems synchronize at beginning/end of round.

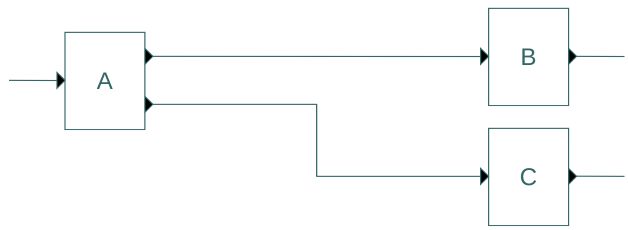


Example: synchronous block diagram

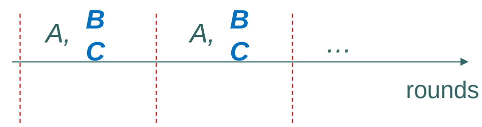




Example: synchronous block diagram

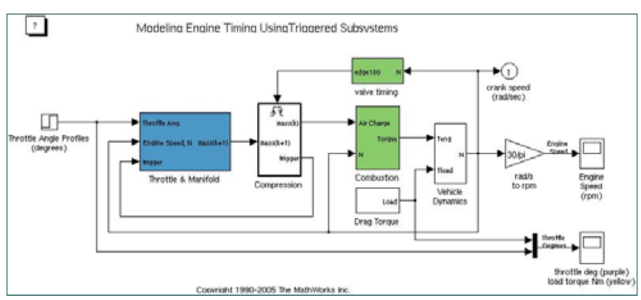


can also execute B, C in parallel



Deterministic concurrency (contrast to threads)

What about models with feedback?



Engine control model in Simulink

Copyright The Mathworks

Defining the semantics of synchronous feedback

Two basic approaches:

Non-deterministic semantics: used for verification (e.g., tools like NuSMV)

Deterministic semantics: used for implementation (e.g., circuits or synchronous languages)

Non-deterministic semantics

Main idea:

composition = **conjunction** of transition relations

Composition as conjunction of transition relations

Systems A and B described symbolically:

Sets of variables (some may be common): X_A, X_B

Initial state formulas: $init_A(X_A\ ),\ init_B(X_B\ )$

Next state formulas: $tr_A(X_A, X_A'),\ tr_B(X_B, X_B')$

Composition as conjunction of transition relations

Composite system described by:

Set of variables: $X_A \cup X_B$

Initial state formula: $init_A(X_A\ ) \wedge init_B(X_B\ )$

Next state formula: $tr_A(X_A, X_A') \wedge tr_B(X_B, X_B')$

Example: a model with feedback in NuSMV

```
MODULE identity(input)
VAR
 output : boolean;
TRANS
 output = input
```

```
MODULE inverter(input)
VAR
 output : boolean;
TRANS
 output = !input
```


Example: a model with feedback in NuSMV

$x = y$

$y = \neg x$

```
MODULE identity(input)
VAR
 output : boolean;
TRANS
 output = input
```

x

y

```
MODULE inverter(input)
VAR
 output : boolean;
TRANS
 output = !input
```

Together: $x = y \wedge y = \neg x$ No solution.

NuSMV issues warning about “fair states set” being empty.

Example: a model with feedback in NuSMV

$x = \neg y$ $\qquad\qquad$ $y = \neg x$

```
MODULE inverter(input)          MODULE inverter(input)
VAR                        x    VAR
 output : boolean;        --->   output : boolean;
TRANS                      y    TRANS
 output = !input          <---   output = !input
```

Together: $x = \neg y \wedge y = \neg x$ Two solutions.

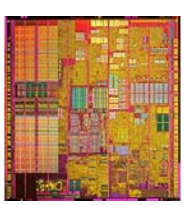
NuSMV considers both states as reachable.

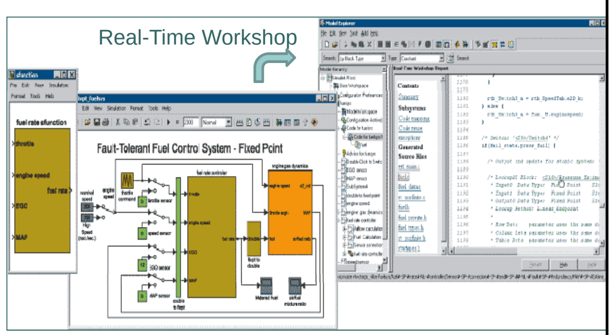
Modeling (for verification) vs. programming (implementing)

Non-deterministic semantics OK for verification: can be seen as over-approximation of all possible behaviors.

Synchronous models essential also for **programming**:

Verilog/VHDL





Modeling (for verification) vs. programming (for implementation)

When programming, semantics need to be well-defined and implementable.

E.g., what circuit are we supposed to synthesize from this model?

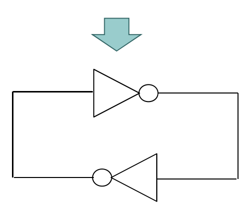
$$x = \neg y \wedge y = \neg x$$

Modeling (for verification) vs. programming (for implementation)

When programming, semantics need to be well-defined and implementable.

E.g., what circuit are we supposed to synthesize from this model?

$$x = \neg y \wedge y = \neg x$$



ambiguous behavior

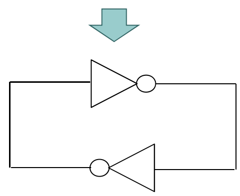
Modeling (for verification) vs. programming (for implementation)

When programming, semantics need to be well-defined and implementable.

E.g., what circuit are we supposed to synthesize from this model?

$$x = \neg y \wedge y = \neg x$$

Guaranteed to stabilize?



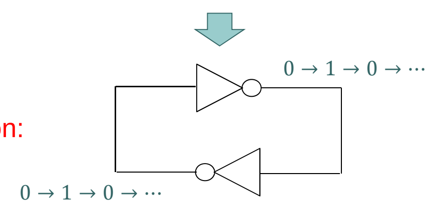
Modeling (for verification) vs. programming (for implementation)

When programming, semantics need to be well-defined and implementable.

E.g., what circuit are we supposed to synthesize from this model?

$$x = \neg y \wedge y = \neg x$$

Possible oscillation:



Defining the semantics of synchronous feedback

Two basic approaches:

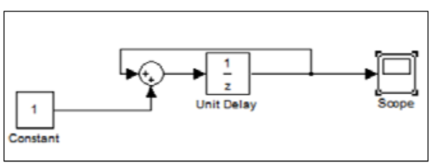
Non-deterministic semantics: used for verification

Deterministic semantics: used for implementation – two approaches to ensure determinism:

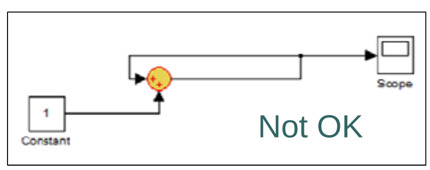
1. **Strict approach**: Forbid instantaneous feedback: e.g., as in Lustre, SCADE, Simulink (unless if algebraic loops explicitly enabled)
2. **Non-strict approach**: Constructive fixpoint semantics: e.g., as in Esterel, Ptolemy

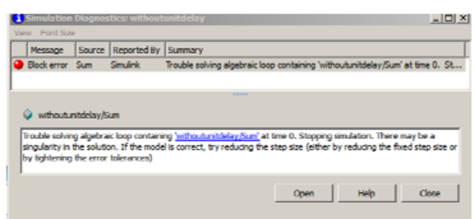
"Strict" approach: forbid instantaneous feedback

Forbid feedback unless if "broken" by unit-delay components (Moore machines)



OK



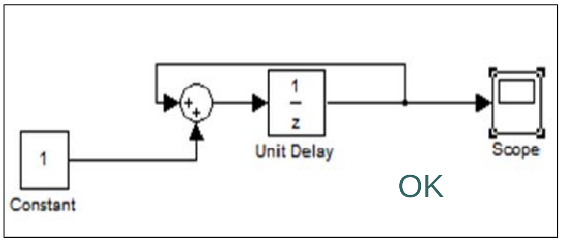


More about solving algebraic loops in lectures on continuous systems

“Strict” approach: forbid instantaneous feedback

Forbid feedback unless if “broken” by unit-delay components (Moore machines)

Why does this work?



At the beginning of each cycle, outputs of Moore machines are known => acyclic dependency graph.

Defining the semantics of synchronous feedback

Two basic approaches:

Non-deterministic semantics: used for verification (e.g., tools like NuSMV)

Deterministic semantics: used for programming – two approaches to ensure determinism:

1. **Strict approach**: Forbid instantaneous feedback
2. **Non-strict approach**: Constructive fixpoint semantics

Strict approach is sometimes too strict

Some circuits have cycles, but their output is well-defined for all inputs.

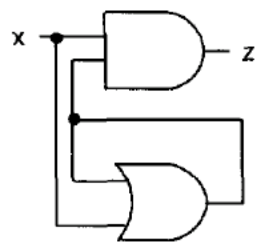


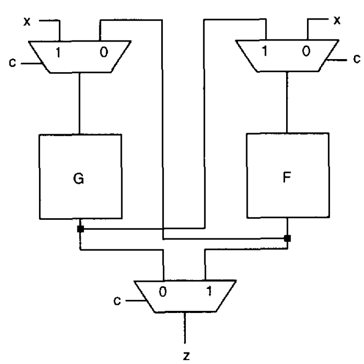
Fig. 1. Cyclic Combination Circuits: A Simple Example.

[Malik, Trans. on CAD, 1994]

Practical cyclic combinational circuits

$$z = if\ (c) then\ F\big(G(x)\big)\ else\ G\big(F(x)\big)$$

Is there an equivalent acyclic circuit?



[Malik, Trans. on CAD, 1994]

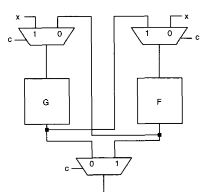
Esterel:
A Synchronous/Reactive Programming Language

```
present I then
  present S then emit T end
else
  present T then emit S end
end
```

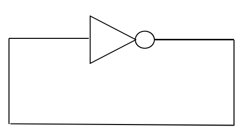
"There is a path from S to T and a path from T to S, hence a cycle. However, it is obvious from the source code that only one path can be used at a time, and, therefore, that the circuit is well-behaved."

[Shiple, Berry, and Touati, 1996]

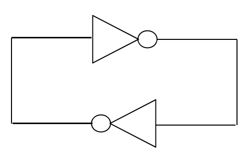
"Good" and "bad" cyclic circuits



good

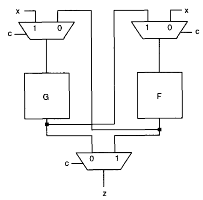


bad: no solution, oscillation

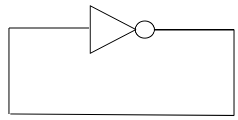


bad: two solutions, possible oscillation

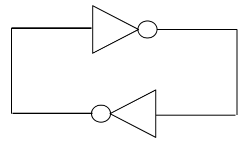
How to analyze cyclic circuits?



good



bad: no solution, oscillation



bad: two solutions, possible oscillation

Constructive fixpoint semantics

Analyzing cyclic circuits using the constructive fixpoint approach: basic idea

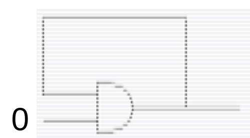
Start with all signal values unknown (denoted $\perp$: “bottom”)

Try to derive known values based on circuit logic.

Iterate until no more known values can be derived.

If all values known, circuit is good.

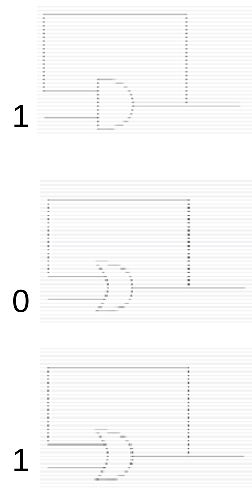
Analyzing cyclic circuits using the constructive fixpoint approach: example



Analyzing cyclic circuits using the constructive fixpoint approach: example

Fixpoint reached.

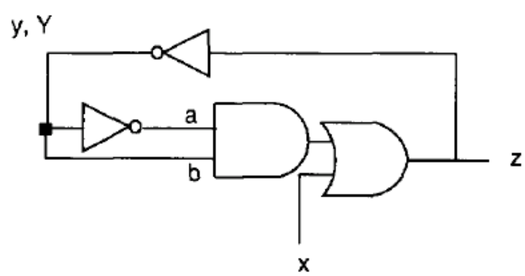
Analyzing cyclic circuits using the constructive fixpoint approach: other examples



Constructive semantics (ternary logic) vs. classic logic

We could interpret our circuits also in classic logic: only 0, 1 (true, false). No “unknown” (⊥) value.

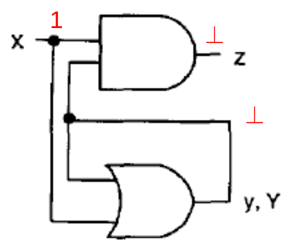
But there are bad circuits with unique fixpoints in classic logic:



Logically, the output of the AND gate is 0. But if *x=0*, and the inverters have delay, then this circuit will oscillate.

Applying the procedure

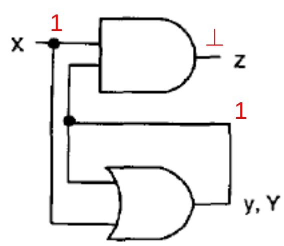
Initialize with input values and “unknown” on other nodes.



[Malik, Trans. on CAD, 1994]

Applying the procedure

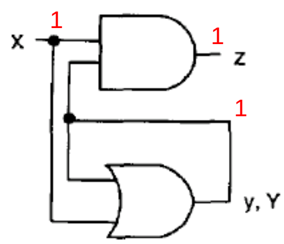
Evaluate lower gate.



[Malik, Trans. on CAD, 1994]

Applying the procedure

Evaluate gates in arbitrary order until nothing changes.

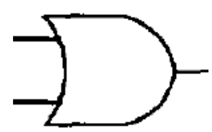


Does this mean the circuit is valid ("constructive")?

Not necessarily: must try all other possible inputs

At this point, all nodes are known => fixpoint reached.

Implicitly using Parallel (Non-Strict) Or



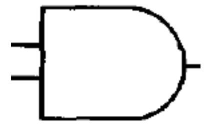
The non-strict or (often called the "parallel or") can produce a known output even if the input is not completely know. Here is a table showing the output as a function of two inputs:

input 1 (columns); input 2 (rows)

	⊥	F	T
⊥	⊥	⊥	T
F	⊥	F	T
T	T	T	T

Extending gates in "ternary" (constructive) logic

Implicitly using
Parallel (Non-Strict) And



The non-strict and (often called the “parallel and”) can produce a known output even if the input is not completely know. Here is a table showing the output as a function of two inputs:

input 1

input 2	⊥	F	T
⊥	⊥	F	⊥
F	F	F	F
T	⊥	F	T

Applying the procedure with input 0 on our circuit and a variant of it

Initialize with input values and “unknown” on other nodes.

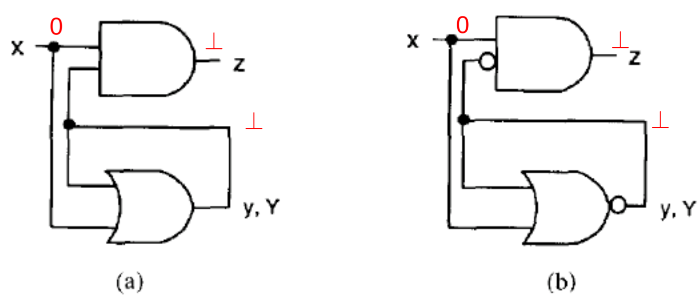


Fig. 4. Cyclic combinational circuits with sequential parts.

[Malik, Trans. on CAD, 1994]

Applying the procedure with input 0 on our circuit and a variant of it

Evaluate gates in arbitrary order until no progress is made.

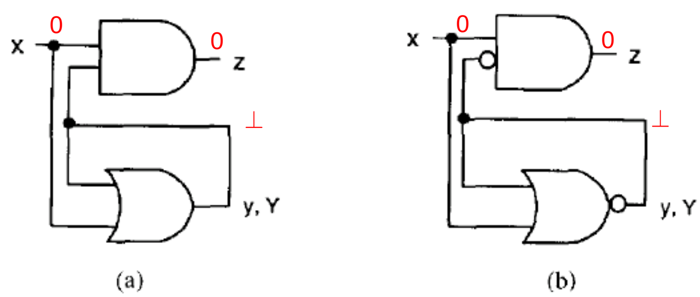


Fig. 4. Cyclic combinational circuits with sequential parts.

Unknown nodes remain. Constructive semantics rejects these circuits.

Key Property

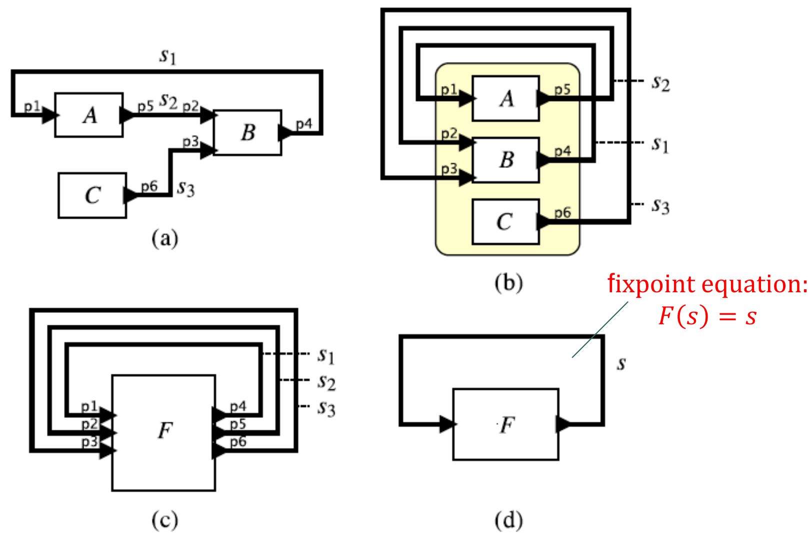
The procedure always converges to a unique solution for all nodes.

The solution may contain unknown values! (⊥)

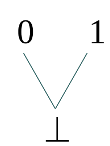
That solution is the least fixed point of a monotonic function on a complete partial order (CPO).

The Kleene fixed-point theorem assures that such a least fixed point exists, is unique, and can be found via this procedure.

A circuit (with inputs known) with m nodes can be represented as a function $F{:}\{0,1,\perp\}^m \rightarrow \{0,1,\perp\}^m$



Our CPO (Complete Partially Ordered Set)

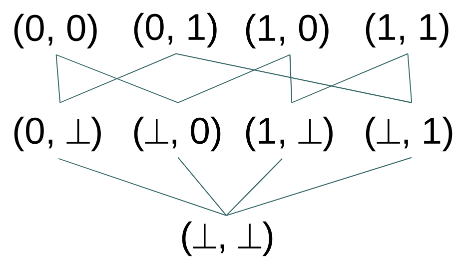


Hasse diagram

This means:

$\perp \leq 0$ and $\perp \leq 1$

Product CPO on Pairs



Hasse diagram

This means:

$(\perp,0) \leq (1,0)$ $\qquad$ $(\perp,1) \leq (1,1)$ $\qquad \ldots$

This generalizes to arbitrary m-tuples.

Height is $m + 1$

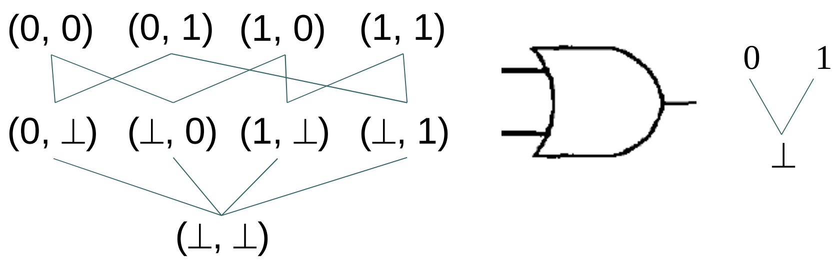
Monotonic (Order Preserving) Functions

Let $(A, \leq)$ and $(B, \leq)$ be posets.

A function $f : A \to B$ is called *monotonic* if

$$a \leq a' \quad \Rightarrow \quad f(a) \leq f(a')$$

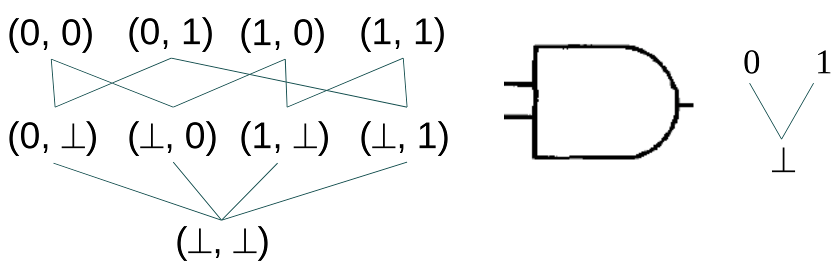
Parallel Or is Monotonic on our CPO



input 1

input 2	⊥	F	T
⊥	⊥	⊥	T
F	⊥	F	T
T	T	T	T

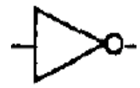
Parallel And is also Monotonic



input 1

input 2	⊥	F	T
⊥	⊥	F	⊥
F	F	F	F
T	⊥	F	T

What about logical NOT ?

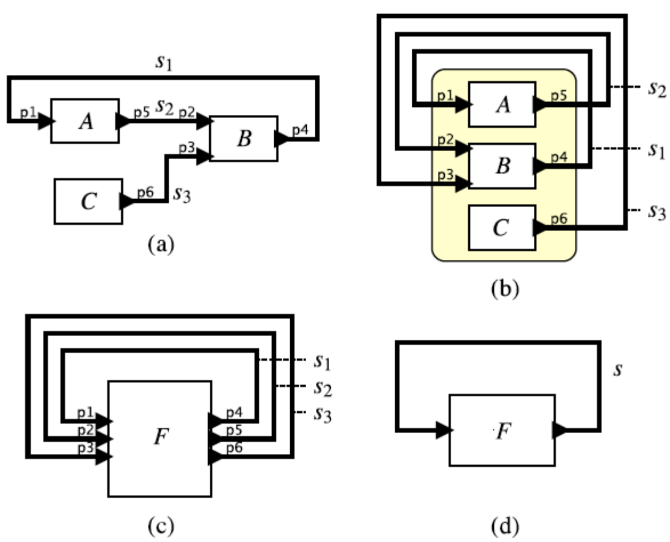


What does the extended truth table (with “uknown”) for NOT look like?

Is NOT monotonic?

Composition of monotonic functions is monotonic

=> $F{:}\{0,1,\perp\}^m \to \{0,1,\perp\}^m$ is monotonic



Fundamental Algorithms for System Modeling, Analysis, and Optimization

Stavros Tripakis
UC Berkeley
EECS 144/244
Fall 2016

Synchronous Composition – Part 2

Thanks to Edward A. Lee for several slides

Defining the semantics of synchronous feedback

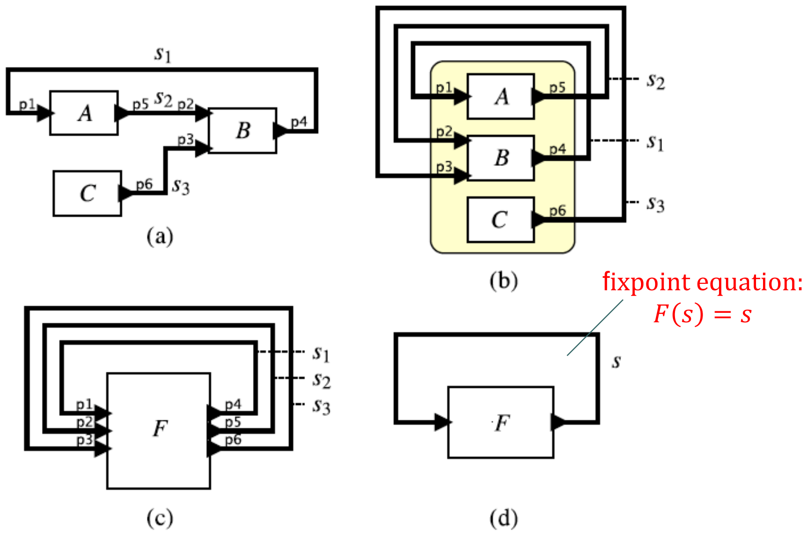
Two basic approaches:

Non-deterministic semantics: used for verification (e.g., tools like NuSMV)

Deterministic semantics: used for programming – two approaches to ensure determinism:

1. **Strict approach**: Forbid instantaneous feedback
2. **Non-strict approach**: Constructive fixpoint semantics

A closed circuit (inputs known) with m nodes = a monotonic function $F\colon\{0,1,\perp\}^m \to \{0,1,\perp\}^m$



Partial orders: basics

Partial Orders

A *partial order* on the set A is a binary relation $\leq$ that is, for all $a, b, c \in A$,

- reflexive: $a \leq a$
- antisymmetric: $a \leq b$ and $b \leq a \Rightarrow a = b$
- transitive: $a \leq b$ and $b \leq c \Rightarrow a \leq c$

A *partially ordered set* (*poset*) is a set A and a binary relation $\leq$, written $(A, \leq)$.

Total Orders

Elements a and b of a poset $(A, \leq)$ are *comparable* if either $a \leq b$ or $b \leq a$. Otherwise they are *incomparable*.

A poset $(A, \leq)$ is *totally ordered* if every pair of elements is comparable.

Totally ordered sets are also called *linearly ordered sets* and *chains*.

Examples

1. $0 < 1$
2. $1 < \infty$
3. child < parent
4. child > parent
5. 11,000/3,501 is a better approximation to π than 22/7
6. integer n is a divisor of integer m.
7. Set A is a subset of set B.

Which of these are partial orders? Total orders?
Which are the corresponding posets?

Fixed Point Theorem
(a variant of the Kleene fixed-point theorem)

Let $(A, \leq)$ be the CPO $\{0,1,\perp\}^m$ (on m-tuples)
Let $f : A \to A$ be a monotonic function
Let $C = \{ f^n(\perp), n \in \{1, \ldots , m\} \}$

$\vee C = f^m(\perp)$ is the *least* fixed point of f

Intuition: The least fixed point of a monotonic function is obtained by applying the function first to unknown, then to the result, then to that result, etc.
Bounded by the height of the CPO, m.

Join (Least Upper Bound)

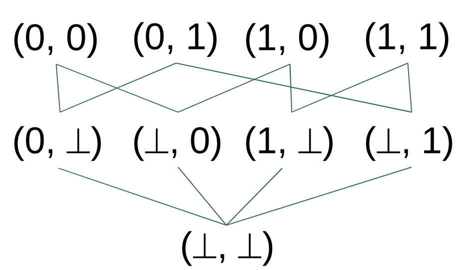
An *upper bound* of a subset $B \subseteq A$ of a poset $(A, \leq)$ is an element $a \in A$ such that for all $b \in B$ we have $b \leq a$.

A *least upper bound* (LUB) or *join* of B is an upper bound a such that for all other upper bounds a' we have $a \leq a'$.

The *join* of B is written $\vee B$.

When the join of B exists, then B is said to be *joinable*.

Least Upper Bound – Examples



Hasse diagram

Does the upper bound exist for

$\{(0,\perp), (\perp, 0)\}$?

$\{(0,\perp), (\perp, 1)\}$?

Does the upper bound exist for: 0<1<2<… ?

Proof of Theorem (part 1: $\vee C$ is a fixed point)

Note that C is a chain in a finite poset:

$\perp \leq f(\perp)$

$f(\perp) \leq f^2(\perp)$ by monotonicity

…

$f^{m-1}(\perp) \leq f^m(\perp)$

Since the longest chain in the poset has length $m + 1$, this sequence has to stop increasing and settle to a fixed point: $f^{k-1}(\perp) = f^k(\perp)$ for some k.

Moreover, $f^k(\perp) = \vee C$. Hence, $\vee C$ is a fixed point of f.

Proof of Theorem (part 2: $\vee C$ is the least fixed point)

Let a be another fixed point: $f(a) = a$

Show that $\vee C$ is the least fixed point: $\vee C \leq a$

Since f is monotonic:

$\perp \leq a$

$f(\perp) \leq f(a) = a$

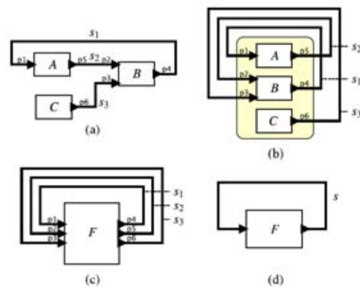
…

$f^m(\perp) \leq f^m(a) = a$

So a is an upper bound of the chain C, hence $\vee C \leq a$.

Brute Force Application of the Theorem

- Start with signals "unknown" at all nodes of the circuit.
- Evaluate components (gates) in arbitrary order repeatedly until no further progress is made.
- If the result has all signals "known," then declare it to be the constructive solution.
- Otherwise, reject model as non-constructive (buggy).



Does evaluation order matter?

In the circuit below, evaluation order matters:

- 1, 2, 3, 4: requires three passes to converge.
- 3, 1, 4, 2: requires one pass to converge.

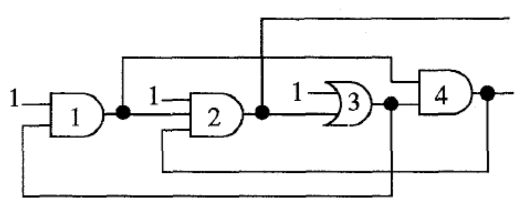


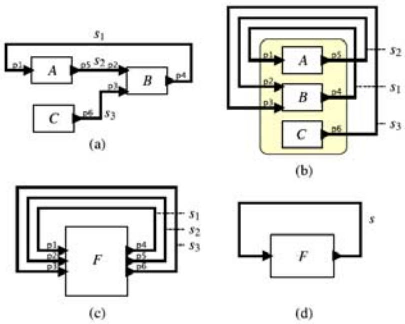
Figure 3: Number of gate evaluations depends on evaluation order.

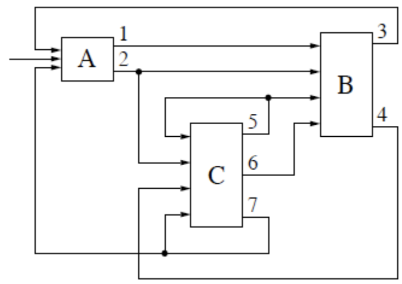
There exists research to optimize this [see paper by Edwards-Lee]

[Shiple, Berry, and Touati, 1996]

What if inputs are unknown?

We will extend the solution to **open** systems.





From closed … … to open systems

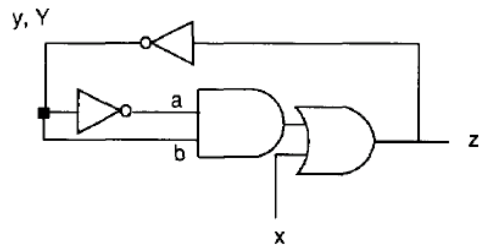
We want to <u>avoid</u> the **brute-force** method of checking all possible inputs.

Instead: use **symbolic execution**

Main idea: instead of iterating over **values**, iterate over **functions**:

- Start with unknown *function of the inputs* at all nodes except inputs.
- Update the functions in arbitrary order repeatedly until no further progress is made.
- If the result has all functions known, then declare the circuit constructive.

Symbolic execution



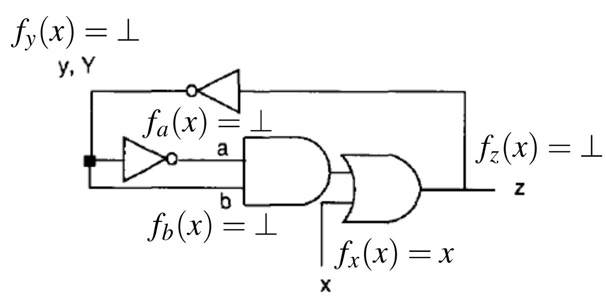
Assume a single binary input (for now). For each node a in the circuit, define a function from the input to the node value:

$$f_a : \{0,1\} \rightarrow \{\perp, 0, 1\}$$

These give the outputs as a function of x only.

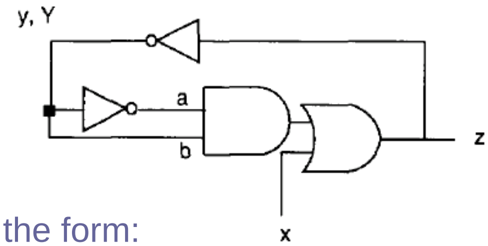
Symbolic execution strategy

Start with all nodes except inputs being given by the unknown function:



Then update these functions iteratively until convergence. But how to update the functions?

First: how to represent the functions



Represent each function of the form:

$$f_a : \{0,1\} \rightarrow \{\perp, 0, 1\}$$

using **two** *characteristic functions* of the form:

$$f_a^0 : \{0,1\} \rightarrow \{0,1\}$$
$$f_a^1 : \{0,1\} \rightarrow \{0,1\}$$

where

$$f_a^0(x) = \begin{cases} 1 & \text{if } f_a(x) = 0 \\ 0 & \text{otherwise} \end{cases} \qquad f_a^1(x) = \begin{cases} 1 & \text{if } f_a(x) = 1 \\ 0 & \text{otherwise} \end{cases}$$

How can these be represented in practice?

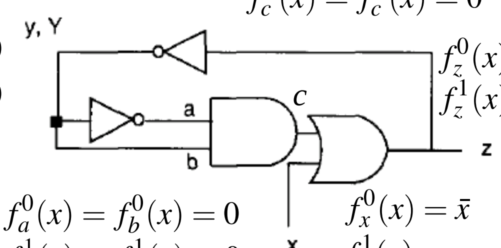
Using **BDDs**!

Symbolic execution strategy using characteristic functions

Start with all nodes except inputs being given by the unknown function:

$$f_c^0(x) = f_c^1(x) = 0$$

$$f_y^0(x) = 0 \qquad f_z^0(x) = 0$$
$$f_y^1(x) = 0 \qquad f_z^1(x) = 0$$

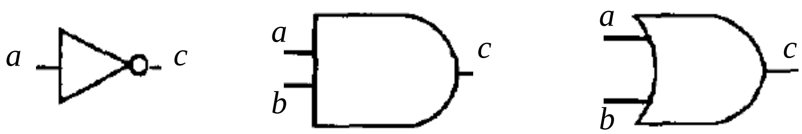


$$f_a^0(x) = f_b^0(x) = 0 \qquad f_x^0(x) = \bar{x}$$
$$f_a^1(x) = f_b^1(x) = 0 \qquad f_x^1(x) = x$$

Then update these functions iteratively until convergence. But how to update the functions?

Operating on characteristic functions

Gates relate characteristic functions of the outputs with those of the inputs:



$$f_c^0(x) = f_a^1(x) \qquad f_c^0(x) = f_a^0(x) + f_b^0(x) \qquad f_c^0(x) = f_a^0(x) \cdot f_b^0(x)$$

$$f_c^1(x) = f_a^0(x) \qquad f_c^1(x) = f_a^1(x) \cdot f_b^1(x) \qquad f_c^1(x) = f_a^1(x) + f_b^1(x)$$

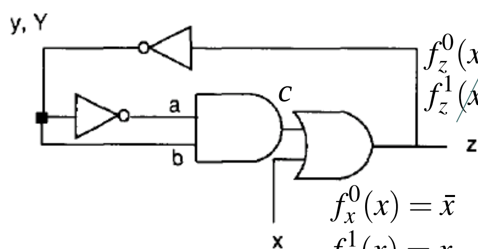
Symbolic execution strategy using characteristic functions

$$f_z^0(x) = f_c^0(x) \cdot f_x^0(x) = 0 \cdot \bar{x} = 0$$

$$f_z^1(x) = f_c^1(x) + f_x^1(x) = 0 + x = x$$

Update nodes in arbitrary order:

$$f_y^0(x) = 0$$
$$f_y^1(x) = 0$$



$$f_z^0(x) = 0$$
$$f_z^1(x) = 0$$

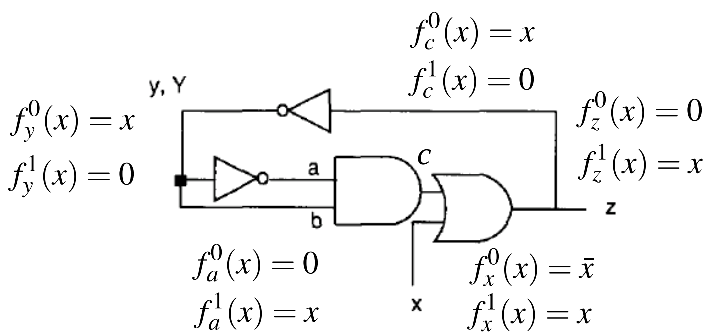
$$f_y^0(x) = f_z^1(x) = x$$
$$f_y^1(x) = f_z^1(x) = 0$$

$$f_x^0(x) = \bar{x}$$
$$f_x^1(x) = x$$

etc.

Convergence

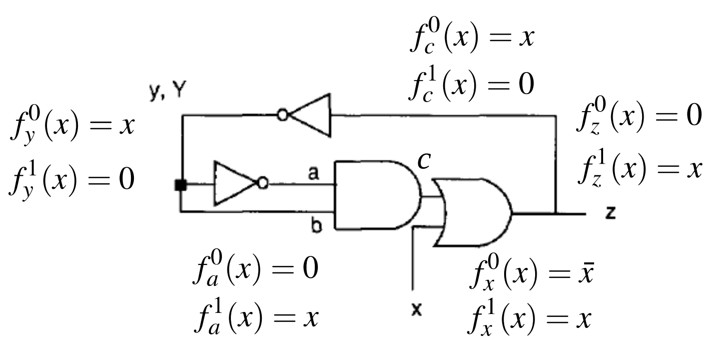
Quickly converge to these characteristic functions:



How do we know whether the circuit is constructive?

Checking whether circuit is constructive

Quickly converge to these characteristic functions:



Circuit is constructive iff at all nodes *a* we have for all *x*

$$f_a^0(x) + f_a^1(x) = 1$$

i.e. the value is known! (Checking this is a SAT problem)

Does the procedure always converge? Is the answer unique?

Consider a poset {0, 1} where 0 < 1.

This induces a poset on the set of functions of form:

$$f_a^i : \{0,1\} \to \{0,1\} \qquad \text{How?}$$

This poset has a bottom element: the function

$$f_\perp^i(x) = 0$$

This poset is finite, with structure much like the flat order. The Kleene fixed-point theorem applies. Extends easily to tuples of functions.

Gate operations on characteristic functions are monotonic functions!

These are monotonic in the sense that if you know more about the inputs, then you learn more about the outputs:

$$(f_a^0, f_a^1) \le (g_a^0, g_a^1) \Rightarrow (f_c^0, f_c^1) \le (g_c^0, g_c^1)$$

$$((f_a^0, f_b^0), (f_a^1, f_b^1)) \le ((g_a^0, g_b^0), (g_a^1, g_b^1)) \Rightarrow (f_c^0, f_c^1) \le (g_c^0, g_c^1)$$

$$f_c^0(x) = f_a^1(x) \qquad f_c^0(x) = f_a^0(x) + f_b^0(x) \qquad f_c^0(x) = f_a^0(x) \cdot f_b^0(x)$$

$$f_c^1(x) = f_a^0(x) \qquad f_c^1(x) = f_a^1(x) \cdot f_b^1(x) \qquad f_c^1(x) = f_a^1(x) + f_b^1(x)$$

Extension to sequential circuits: circuits with state

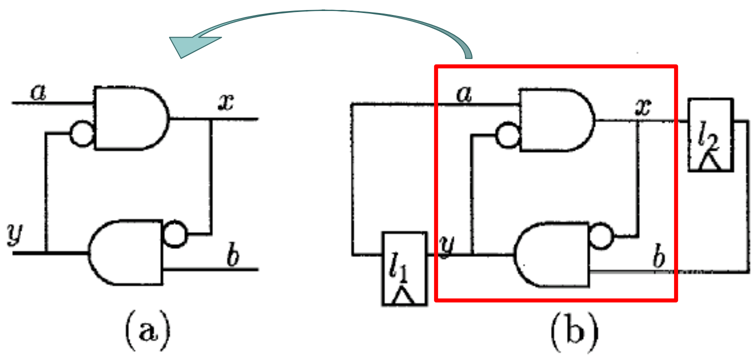


Figure 1: Circuits are well-behaved unless $a = b = 1$.

First need to find which inputs are problematic, if any.

Then need to determine whether those inputs can occur (reachability analysis on a state machine)

[Shiple, Berry, and Touati, DATE, 1996]

Asynchronous Composition

See 11-asynchronous.pdf

Bibliography

Malik, S. (1994). Analysis of cyclic combinational circuits. IEEE Trans. Computer-Aided Design, 13(7):950–956.

Edwards, S. and Lee, E. (July 2003). The semantics and execution of a synchronous block-diagram language. Science of Computer Programming, 48:21–42(22).

Shiple, T., Berry, G., and Touati, H. (1996). Constructive analysis of cyclic circuits. In European Design and Test Conference (EDTC'96). IEEE Computer Society.

Berry, G. (1999). The Constructive Semantics of Pure Esterel.

Davey, B. A. and Priestley, H. A. (2002). Introduction to Lattices and Order. Cambridge University Press, 2nd edition.