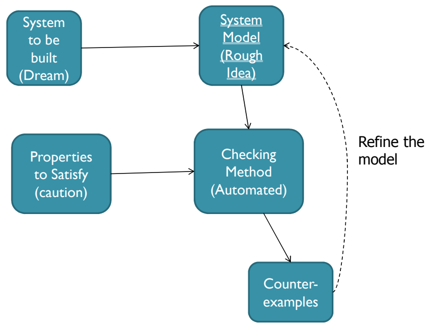
Transition Systems as Models

Abhik Roychoudhury
CS 4271 Lecture Notes
School of Computing, NUS

Warm up – the big picture


System to be built (Dream)
System Model (Rough Idea)
Properties to Satisfy (caution)
Checking Method (Automated)
Counter-examples
Refine the model

Today's lecture

- System Models
  - What kind of models we use?
    - Transition Systems.
  - How do we characterize their behaviors?
    - Execution Traces.
    - … (more later)
  - How the model relates to the real system?
    - Associate transition systems with programs/circuits.

The purpose

- A transition system is supposed to serve as a model of system behavior.
  - Model is required for analysis and verification of program/system.
  - Once the model is derived, our technique focuses on space and time efficient search of the model (for verification).

What sort of a model ?

- Will capture evolution of the program/ circuit with the passage of time.
- Will contain information about internal values which are essential for establishing correctness
  - We consider functional correctness
- Will leave out low-level details e.g. the data values exchanged in a communication protocol

- Key point to remember
  - Focus on how the system state changes with system execution.

Transition system

- M = (S, S0, R)
  - S = Set of states (may not be finite)
  - S0 = Set of initial states
  - $R \subseteq S \times S$ is the transition relation
- Model M can then be subjected to verification.
- Need to associate states and transitions with the text of a program.

Why states/transitions are important ?

- We are looking at reactive systems
  - Never-ending evolution over time.
- Snapshots of evolution captured via states
  - Stop the evolution at any time and peek into it.
- One atomic step of evolution captured by transition

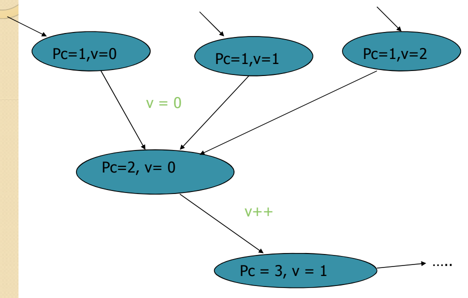
The association

- Associating states/transitions with pgm.
  - Program Variables
    - Program Counter (pc), Data variables
  - Program State
    - Valuation of program variables
  - Transition
    - Moving from one state to another by executing a program statement.

Example

- 1 v = 0;
- 2 v++;
- 3 …
  - What are the states ?
    - (value of pc, value of v)
  - How many initial states are there ?
    - No info, depends on the type of v
- Draw the states and transitions corresponding to this program.

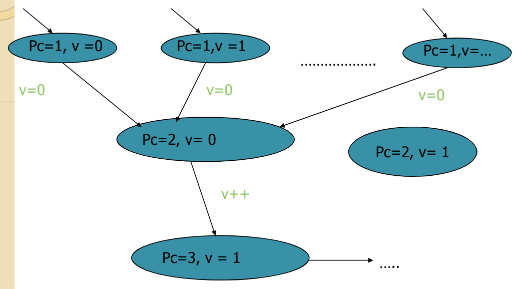
Example - Continued



Too many states ?

- Defined by possible variable valuations
  - Determined by variable types
    - integer, float … : this a problem !
  - State space explosion is the central problem in model checking.
  - Not all of the states might however not be reachable from the initial state(s)
    - Do not appear in the model (i.e. if we were to draw it !)

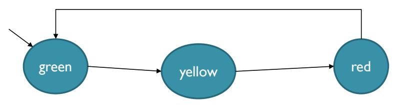
Example - continued



Propositions

- We are trying to generate a high-level model of system behavior for validation
- With each state, we capture the truth/falsehood w.r.t. atomic propositions
  - [v == 0 ?]
  - [u == v ?]
- Where do we get these propositions ?
  - Depends on what properties we want to verify !
  - Related to variables in the program being verified.
  - We assume the set of all AP is given.

Transition System with Propositions



Atomic propositions = {green, yellow, red}
--- Instead of having a "color" variable

In each state, only the atomic propositions which are true are shown.

Kripke Structures

- An unifying formalism based on transition systems
  - State transition graph (S, S0, R, L)
    - S is a (finite) set of states
    - $S0 \subseteq S$ is the set of initial states
    - $R \subseteq S \times S$ is the state transition relation such that every state has at least one successor
    - And …

Kripke Structures

  - $L : S \rightarrow 2^{AP}$ is a labeling function which assigns a set of atomic propositions to a state
    - L allows us to describe the truth/falsehood of a proposition in the various system states.
    - The propositions refer to valuations of state variables.
- Kripke structures are powerful enough to model behaviors of
  - sequential as well as concurrent systems,
  - software as well as hardware.

Obtaining Kripke Structures

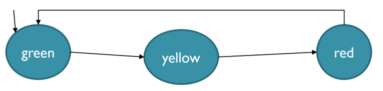
- Obtaining Kripke Structure from a concurrent program directly is laborious.
- A model checking tool allows you to input the program in its modeling language, and then it extracts the Kripke Structure.
- You model the sequential threads separately, and specify a model of concurrency
  - e.g. asynchronous with shared variable communication

Obtaining Kripke Structures

- Later in today's lecture [Revision Hour]
  - Choose a toy programming language IMP
  - Define valuations for every operator (arithmetic, boolean etc)
  - Define state changes for every statement type.
- Directly defining states and transitions for programs written in IMP without going through first-order logic
  - Gives a flavor of the laborious task involved in defining an execution model for programs written in a standard PL.

BTW, what is behavior ?

- What do we mean when we say Kripke structures model "system behavior" ?
  - For now, consider the infinite traces of the Kripke structure as representing behavior.
    - We consider Kripke structures where each state has at least one successor state.
    - Represents an ongoing evolution of states in a reactive system.
    - Non-determinism in evolution due to concurrency.



Behavior is given by the trace

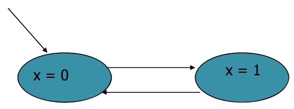
$(g, y, r)^{\omega}$

Now…

- How do Kripke Structures correspond to behavior of
  - Sequential programs
  - Concurrent programs
  - Hardware circuits

Sequential program - Example

```
Initially x is 0
while true do
      x := 1 – x;
endwhile
```



Exercise: We did not model the program counter in this example, Can you try to do so?

Concurrent Program

- A set of programs running independently, communicating from time to time, thereby performing a common task.
- Flavors of Concurrency
  - Synchronous execution
    - Common in hardware circuits.
  - Asynchronous/interleaved execution
    - Communication via shared variables
    - Message passing communication

Concurrent Program - Example

P0 || P1

P0	P1
l0: while true do	m0: while true do
l1: wait(turn = 0);	m1: wait(turn = 1);
l2: turn := 1;	m2: turn := 0;
l3: endwhile	m3: endwhile

Models a crude protocol for entry/exit to critical section without modeling the critical section itself.

States

- Global State = (pc0, pc1, turn)
  - pc0 $\in$ { l0, l1, l2, l3 }
  - pc1 $\in$ { m0, m1, m2, m3 }
  - turn $\in$ { 0, 1 }
- Total = 4 * 4 * 2 = 32 possible states
  - Not all of them might be reachable from the initial states.

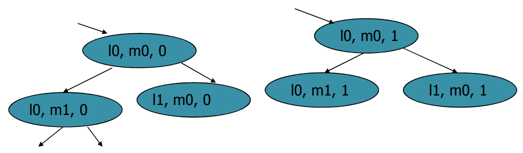
Propositions

- Define propositions to capture information about variable valuations. Possible propositions in our ex.
  - pc0 = l0, pc0 = l1, pc0 = l2, pc0 = l3
  - pc1 = m0, pc1 = m1, pc1 = m2, pc1 = m3
  - turn = 0, turn = 1
- Clearly the proposition pc0 = l0 is true in any state
  - (pc0 = l0, pc1 = ?, turn = ??)
- Clarifies the labeling function L in Kripke Structures.

System Properties

- Propositions can be used to define interesting properties
  - It is never the case that pc0 = l2 and pc1 = m2
- The above property defines mutually exclusive access to the critical section.
- We will study a logic for describing such properties in the next class.

State Transition Graph



There are two initial states.

Finite number of possible states, so the construction (of the Kripke Structure) must terminate.

Verification of the mutual exclusion property can be achieved by inspection (of the constructed Kripke Structure) in this simple example.

Control and data variables

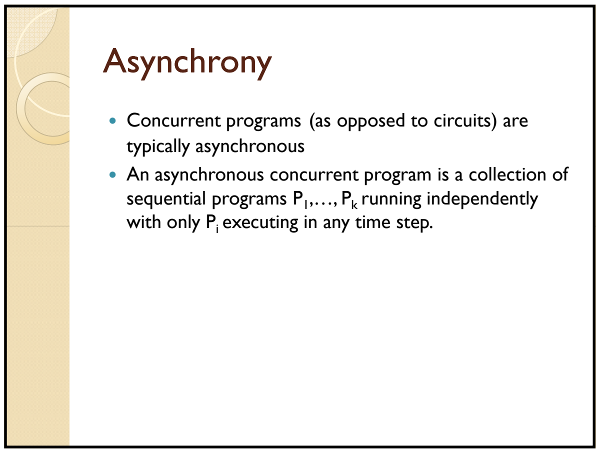
- State = valuation of control and data vars.
- In our example
  - *pc0, pc1* are control variables.
  - *turn* is a shared data variable.
- To generate a finite state transition system
  - Data variables must have finite types, and
  - Finitely many control locations

Data variables

- Data variables often do not have finite types
  - integer, …
  - Usually abstracted into a finite type.
  - An integer variable can be abstracted to {-,0,+}
    - Just store the information about the sign of the variable.
  - ***Caution: Coming up with these abstractions is a whole new problem which we are not discussing in this course.***

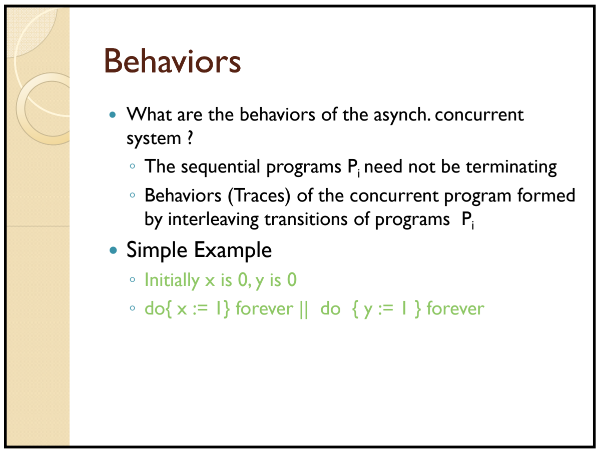
Control Locations

- # of control locations of a program is always finite ?
  - NO, because your program may be a concurrent program with unboundedly many processes or threads (parameterized system).
  - Can employ control abstractions (such as symmetry reduction)
  - ***We will not consider these issues in CS 4271.***
- Let us revise and recapitulate
  - the use of Kripke structures for modeling behaviors of asynchronous concurrent systems



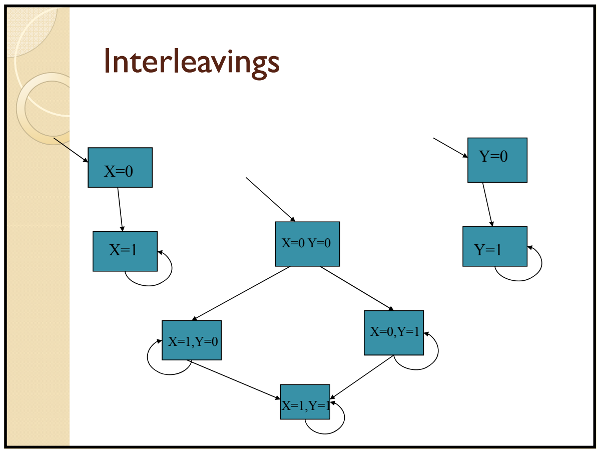
Asynchrony

- Concurrent programs (as opposed to circuits) are typically asynchronous
- An asynchronous concurrent program is a collection of sequential programs $P_1,\ldots,P_k$ running independently with only P_i executing in any time step.

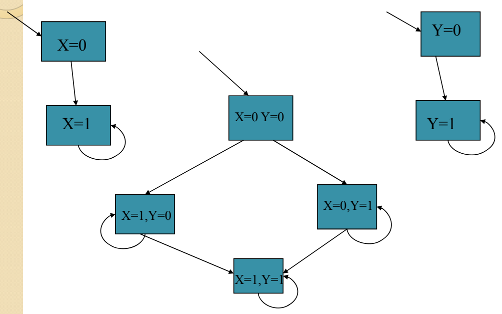


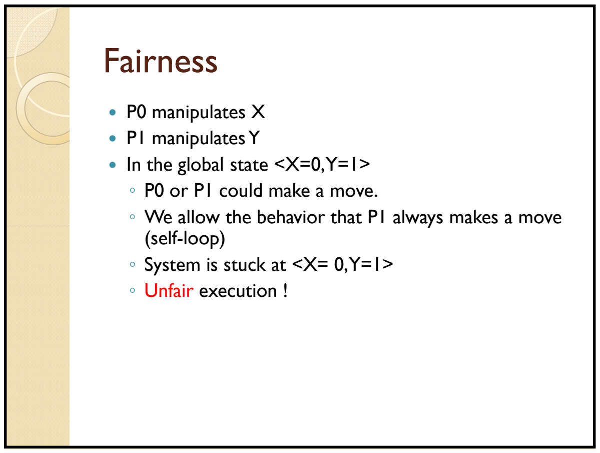
Behaviors

- What are the behaviors of the asynch. concurrent system ?
  - The sequential programs P_i need not be terminating
  - Behaviors (Traces) of the concurrent program formed by interleaving transitions of programs P_i
- Simple Example
  - Initially x is 0, y is 0
  - do{ x := 1} forever || do { y := 1 } forever



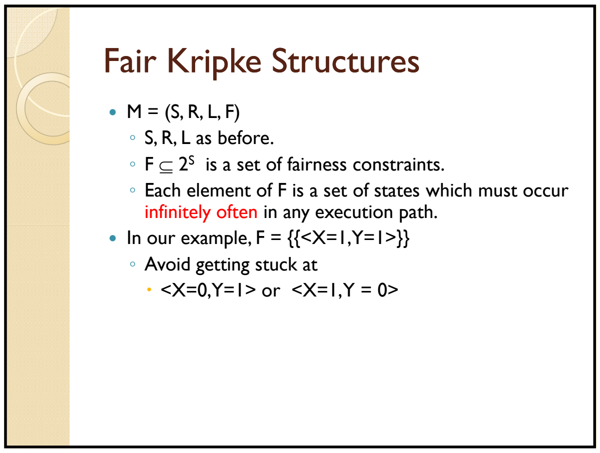
Interleavings





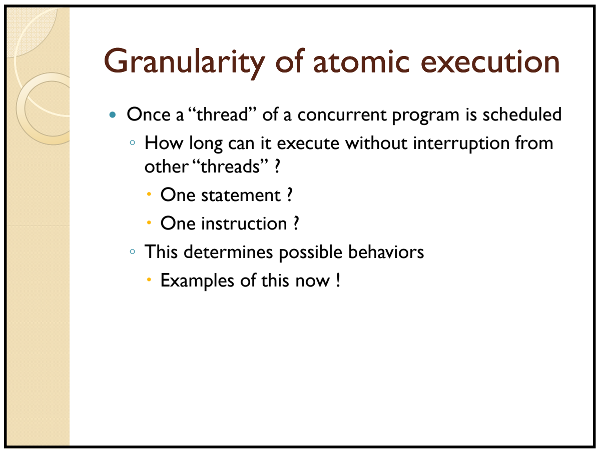
Fairness

- P0 manipulates X
- P1 manipulates Y
- In the global state <X=0,Y=1>
  - P0 or P1 could make a move.
  - We allow the behavior that P1 always makes a move (self-loop)
  - System is stuck at <X= 0,Y=1>
  - Unfair execution !



Fair Kripke Structures

- M = (S, R, L, F)
  - S, R, L as before.
  - $F \subseteq 2^S$ is a set of fairness constraints.
  - Each element of F is a set of states which must occur infinitely often in any execution path.
- In our example, F = {{<X=1,Y=1>}}
  - Avoid getting stuck at
    - <X=0,Y=1> or <X=1,Y = 0>



Granularity of atomic execution

- Once a "thread" of a concurrent program is scheduled
  - How long can it execute without interruption from other "threads" ?
    - One statement ?
    - One instruction ?
  - This determines possible behaviors
    - Examples of this now !

One statement at a time

- Consider an asynchronous composition of two processes i.e. in
- any time step only one of them makes a move. These processes communicate via a single shared variable x. Both processes are executing the following infinite loop:
  - ***while true do x := x + x***
- Every time one of the processes is scheduled, it atomically
- executes x := x + x and then again another process is scheduled. The initial value of x is 1.
- What will be the values of x reached during system execution and why ?

One instruction at a time

- Suppose the infinite loop is compiled by a naïve compiler as follows. The sequence of instructions
- executed by process A and process B are shown.
- The processes are running asynchronously, and each time a process is scheduled, only its next instruction is executed
- atomically. Initially x = 1.

A version of this problem is originally credited to the Thread Game by Prof. J. Strother Moore (University of Texas).

One instruction at a time

- L1: $reg_A^1 = x$
- $reg_A^2 = x$
- $reg_A^3 = reg_A^1 + reg_A^2$
- $x = reg_A^3$
- go to L1

- L2: $reg_B^1 = x$
- $reg_B^2 = x$
- $reg_B^3 = reg_B^1 + reg_B^2$
- $x = reg_B^3$
- go to L2

Can you construct a schedule to show that x can get the value of 3 ?

Process Communication in Asynchronous Execution

- Shared Variables
  - Mutual exclusion example discussed earlier
    - Programs communicate by setting and resetting the shared variable turn
- Message Passing
  - Typically, asynch. Msg. passing = Each program has its own FIFO queue(s) for receiving messages.
    - Also can be Synchronous message passing = Handshake between sender and receiver.

States/Transitions for Message passing

- Global state should take into account message queue contents.
- As before, in each time step
  - Only one program makes a move
- If the queues grow unboundedly, then what about the number of possible states, need to be careful !
- Here is a program to look at
  - do { 1 ! 2 (msg) } forever || do { 2 ? 1 (msg) } forever
    - 1 !2 --- 1 send to
- ***We do not discuss/use message passing in cs4271.***

Now …

- How do Kripke Structures correspond to behavior of
  - Sequential programs
  - Concurrent programs (Asynchronous)
  - Hardware circuits (Synchronous)

States and Transitions

- Each component makes a move at every step.
- Digital circuits are most often synchronous.
  - Common clock driving the system
- Possible contents of all the flip-flops denote the set of possible states.
- On every clock pulse
  - The content of each flip-flop potentially changes.
  - This defines the transition relation.

States and Transitions

- Assume n flip-flops in the circuit
- Define $V = \{v_1, \ldots, v_n\}$
  - Boolean variables representing the value stored in each flip-flop
  - Any set of states captured by a boolean formula on V
- To define transitions, define n fresh variables
  - $V' = \{v'_1, \ldots, v'_n\}$
  - Next – state variables

States and Transitions

- Transitions represented by a relation $R(V,V') \subseteq V \times V'$
  - $(s, s') \in R(V,V')$ iff $s \rightarrow s'$
  - $R(V,V') = R_1(V,V') \wedge R_2(V,V') \wedge \ldots R_n(V,V')$
  - $R_i(V,V')$ captures the change in state variable v_i, that is, flip-flop i with each clock pulse.
  - Define $Ri(V,V') = (v'_i \Leftrightarrow f_i(V))$ where $f_i(V)$ is a boolean function defining the value of flip-flop i in next clock cycle.
  - Given a synchronous circuit, we need to define $f_i(V)$ for each flip-flop i.

A synchronous mod 8 counter

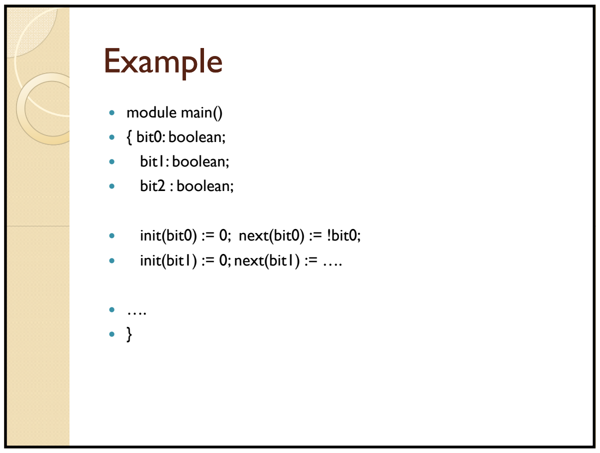
- V = {v2,v1,v0} where v0 is the least significant bit
- The transitions can be enumerated as
  - $000 \rightarrow 001 \rightarrow 010 \rightarrow \ldots$
- For the transition relation, define how each of the three bits get changed from one clock cycle to next
  - $v0' = \neg v0$ (least significant bit)
  - $v1' = v0 \oplus v1$
  - $v2' = (v0 \wedge v1) \oplus v2$ (most significant bit)

So, once again …

- Each bit of the counter can be thought of as an independent communicating process.
  - A flip-flop with its contents as the state of the process
- Here all the processes change state in every time unit
  - A clock cycle is a time unit here.
  - With each cycle, each bit can potentially change state
- Compare this with the asynchronous composition of processes where in each time unit one process is scheduled to execute.
  - Refer to Mutual exclusion example discussed in this lecture.

SMV input language

- SMV model checker's language is close to the transition system encoding for synch. systems.
  - Allows only finite domain variables
  - For each variable v
    - Can specify initial value (otherwise any value from the domain of v)
    - Define v', value of v in next clock cycle
    - No concept of a program counter.
- SMV is the tool we will use in CS4271.

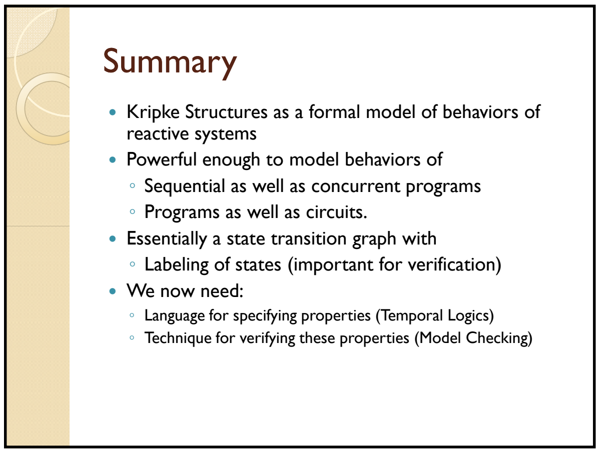


Example

- module main()
- { bit0: boolean;
-   bit1: boolean;
-   bit2 : boolean;

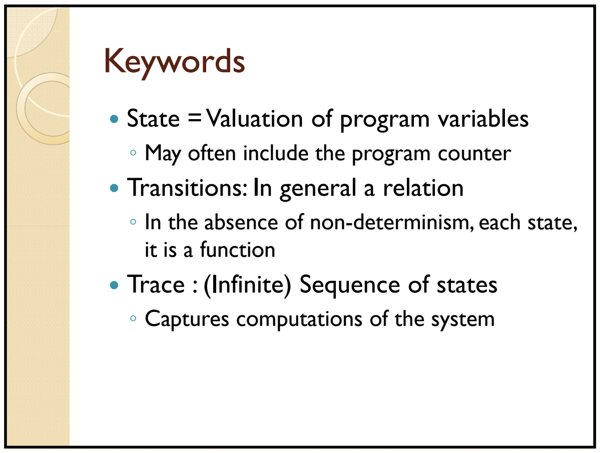
-   init(bit0) := 0; next(bit0) := !bit0;
-   init(bit1) := 0; next(bit1) := ….

- ….
- }



Summary

- Kripke Structures as a formal model of behaviors of reactive systems
- Powerful enough to model behaviors of
  - Sequential as well as concurrent programs
  - Programs as well as circuits.
- Essentially a state transition graph with
  - Labeling of states (important for verification)
- We now need:
  - Language for specifying properties (Temporal Logics)
  - Technique for verifying these properties (Model Checking)



Keywords

- State = Valuation of program variables
  - May often include the program counter
- Transitions: In general a relation
  - In the absence of non-determinism, each state, it is a function
- Trace : (Infinite) Sequence of states
  - Captures computations of the system