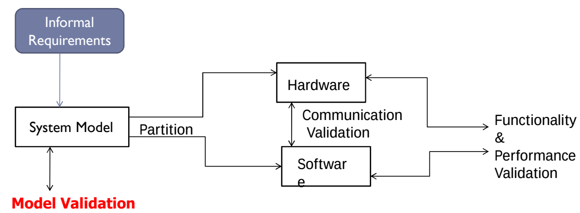
Embedded Systems and Software Validation
Model Validation

Abhik Roychoudhury
National University of Singapore

http://www.comp.nus.edu.sg/~abhik/

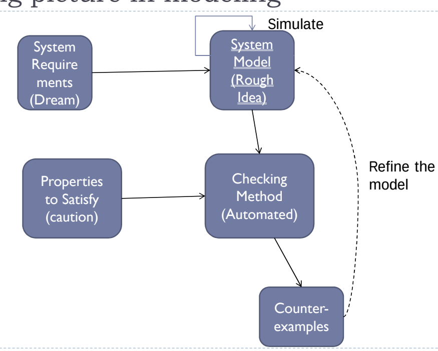
Different kinds of ES Validation



What is a system design model?

- We first clarify the following terms
  - System Architecture: Inter-connection among the system components.
  - System behavior: How the components change state, by communicating among themselves.
- System Design Model = Architecture + Behavior
  - More precise definition later.

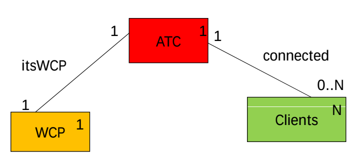
The big picture in modeling



Criteria for a Design Model

- Provides structure as well as behavior for the system components.
- Complete
  - Complete description of system behavior.
- Based on well-established modeling notations.
  - We use UML.
- Preferably executable
  - Can simulate the model, and get a feel for how the constructed system will behave!

Running Example - ATC



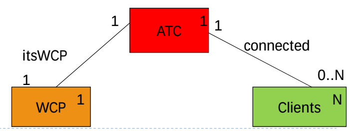
Overall System Structure, Behavior not shown.

On system behavior

- Consider a "scenario"
  - Client1 sends "connect" request to ATC
  - Client2 sends "connect" request to ATC
  - ATC sends weather information to Client1, Client2.
- No need to capture "weather info." in model.
- OK to abstract this info. from the requirements while constructing the model, provided
  - No decisions are made in the system based on weather info.
- Model is "complete" at a certain level of abstraction.

ATC – the example control sys.

- NASA CTAS
  - Automation tools for managing large volume arrival air traffic in large airports.
  - Final Approach Spacing Tool
    - Determine speed and trajectory of incoming aircrafts on their final approach.
    - Master controller updates weather info. to "clients"
      - **controllers using inputs to compute aircraft trajectories.**



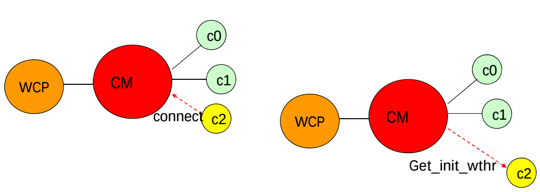
ATC – the example control sys.

- Part of the *Center TRACON Automation System (CTAS)* by NASA
  - manage high volume of arrival air traffic at large airports
  - http://ctas.arc.nasa.gov
- Control weather updating to all weather-aware clients
  - A weather control panel (WCP)
  - Many weather-aware clients
  - A communication manager (CM)

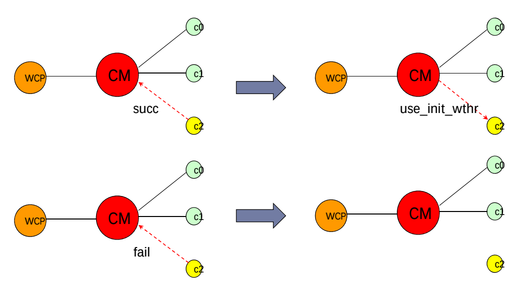
Behavior of ATC example

- Two standard behaviors
  - Client initialization
  - Weather update
- Abstracted Information
  - Weather information types
  - Clients types
  - Internal computation on weather information
- For simplified requirements: textbook Chap 2.3

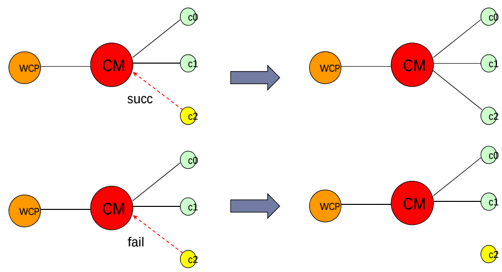
Client Initialization



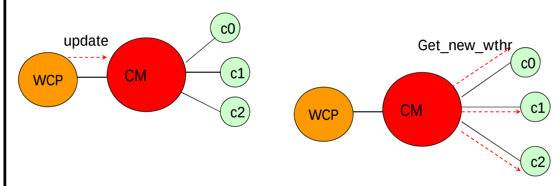
Client - Initialization



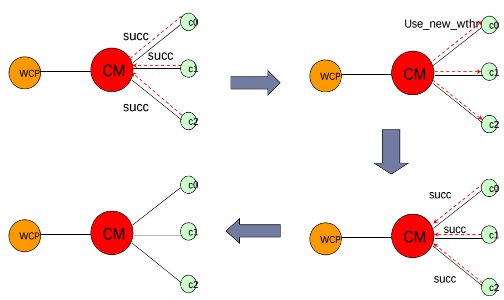
Client - Initialization



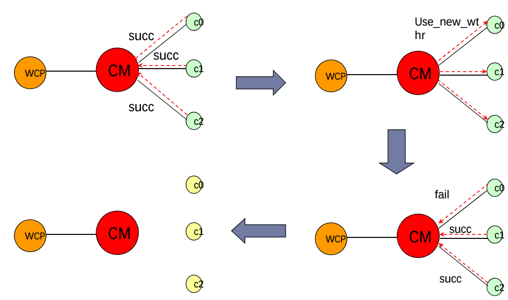
Client – Weather Update



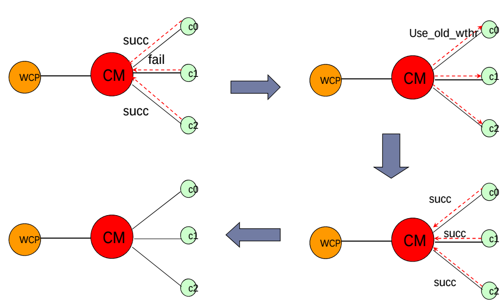
Client Update – Case 1



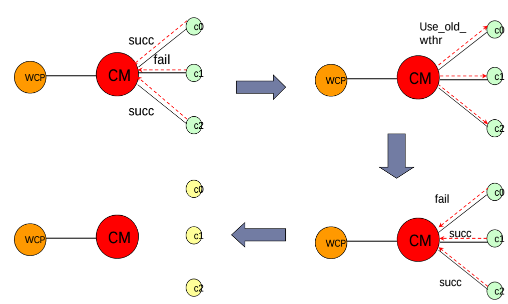
Client Update – Case 2



Client Update – Case 3



Client Update – Case 4



What do the requirements

- … look like ?

A weather update controller is consist of a weather control panel (WCP), a number of weather-aware clients, and a communication manager (ATC) which controls the interactions between the WCP and all connected clients. Initially, the WCP is enabled for manually weather updating, the ATC is at its idle status, and all the clients are disconnected. Two standard behaviors of this system are as follows.

Sample Initialization Requirements

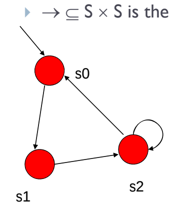
- A disconnected weather-aware client can establish a connection by sending a connecting request to the CM.
- If the ATC's status is idle when the connecting request is received, it will set both its own status and the connecting client's status to preinitializing, and disable the weather control panel so that no manual updates can be made by the user during the process of client initialization.
  - Otherwise (ATC's status is not idle), the ATC will send a message to the client to refuse the connection, and the client remains disconnected.

Organization

- So Far
  - What is a Model?
  - ATC – Running Example
    - Informal Req. at a lab scale.
    - Has subtle deadlock error (see textbook chap 2.3)
- Now, how to model/validate such requirements
    - Modeling Notations
      - Finite State Machines

Finite State Machines

- M = (S, I, $\rightarrow$)
  - S is a finite set of states
  - $I \subseteq S$ is the set of initial states
  - $\rightarrow \subseteq S \times S$ is the transition relation.



S = {s0, s1, s2}

I = {s0}

$\rightarrow$ = {(s0,s1), (s1,s2), (s2,s2), (s2,s0)}

Issues in system modeling …

- … using FSMs
  - Unit step: How much computation does a single transition denote?
  - Hierarchy: How to visualize a FSM model at different levels of details?
  - Concurrency: How to compose the behaviors of concurrently running subsystems (of a large sys.)
    - Each subsystem is modeled as an FSM!

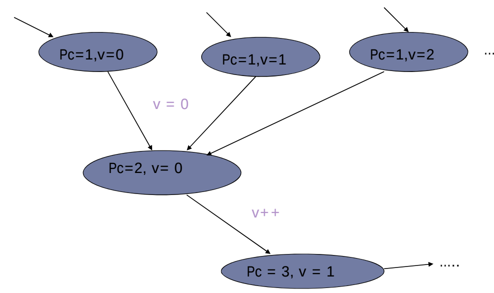
What's in a step?

- For hardware systems
  - A single clock cycle
- For software systems
  - Atomic execution of a "minimal" block of code
    - A statement or an instruction?
    - Depends on the level at which the software system is being modeled as an FSM !

Example

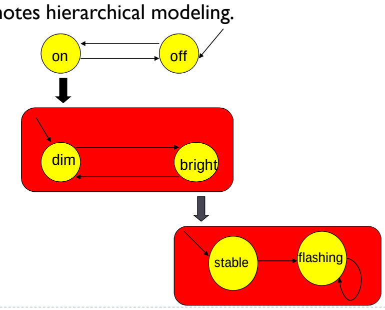
- 1 v = 0;
- 2 v++;
- 3 …
  - What are the states ?
    - (value of pc, value of v)
  - How many initial states are there ?
    - No info, depends on the type of v
- Draw the states and transitions corresponding to this program.

Example



Hierarchy

- Choice of steps at different levels of details also promotes hierarchical modeling.



Basic Concurrent Composition

- $M1 = (S1, I1, \rightarrow_1)$ $M2 = (S2, I2, \rightarrow_2)$
- Define
  - $M1 \times M2 = (S1 \times S2, I1 \times I2, \rightarrow)$
  - Where $(s1,s2) \rightarrow (t1, t2)$ provided
    - $s1 \in S1, t1 \in S1,$
    - $s2 \in S2, t2 \in S2,$
    - $(s1 \rightarrow_1 t1)$ OR $(s2 \rightarrow_2 t2)$
    - Defines control flow of the composed FSM as an arbitrary interleaving of flows from components.
  - *Interleaving of independent flows, what about comm.?*

Communicating FSM

<u>Basic FSM</u>

- $M = (S, I, \rightarrow)$
  - S is a finite set of states
  - $I \subseteq S$ is the set of initial states
  - $\rightarrow \subseteq S \times S$ is the transition relation.

<u>Communicating FSM</u>

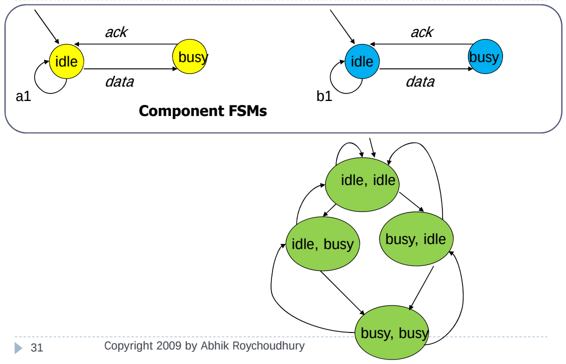
- $M = (S, I, \Sigma, \rightarrow)$
  - S is a finite set of states
  - $I \subseteq S$ is the set of initial states
  - Σ is the set of action names that it takes part in
  - $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.

Communication across FSMs via action names.

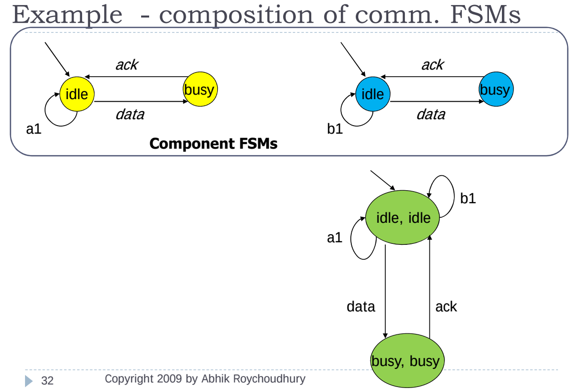
Composition of comm. FSMs

- $M1 = (S1, I1, \Sigma_1, \rightarrow_1)$ $M2 = (S2, I2, \Sigma_2, \rightarrow_2)$
- Define
  - $M1 \times M2 = (S1 \times S2, I1 \times I2, \Sigma_1 \cup \Sigma_2, \rightarrow)$
  - And $(s1,s2) \xrightarrow{a} (t1,t2)$ provided
    - $s1 \in S1, t1 \in S1,$ and
    - $s2 \in S2, t2 \in S2,$ and
    - If $a \in \Sigma_1 \cap \Sigma_2$ we have $(s1 \xrightarrow{a} t1)$ and $(s2 \xrightarrow{a} t2)$
    - If $a \in \Sigma_1 - \Sigma_2$ we have $(s1 \xrightarrow{a} t1)$
    - If $a \in \Sigma_2 - \Sigma_1$ we have $(s2 \xrightarrow{a} t2)$

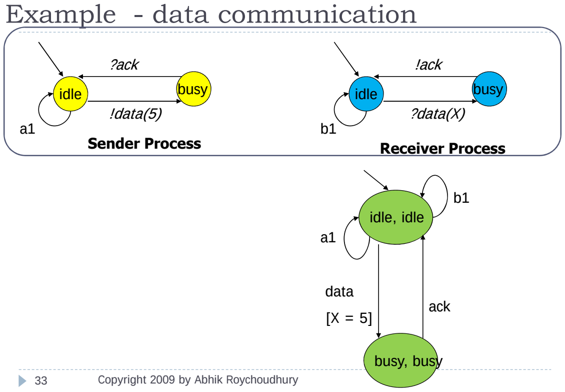
Example - basic composition



Example - composition of comm. FSMs



Example - data communication



Example: Concurrent Program

P0 || P1

- l0: while true do
- l1: wait(turn = 0);
- l2: turn := 1;
- l3: endwhile

- m0: while true do
- m1: wait(turn = 1);
- m2: turn := 0;
- m3: endwhile

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

Example Concurrent Program: States

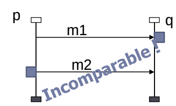
- Global State = (pc0, pc1, turn)
  - pc0 ∈ { l0, l1, l2, l3 }
  - pc1 ∈ { m0, m1, m2, m3 }
  - turn ∈ { 0, 1 }
- Total = 4 * 4 * 2 = 32 possible states
  - Not all of them might be reachable from the initial states.
  - How many are reachable – try it!

Wrap-up of FSMs

- FSMs denote an intra-component style of modeling
  - Given a large system – identify its components
  - Model each component as FSM – M1, M2, M3
  - Overall system modeled as concurrent composition
    - M1 || M2 || M3
- Alternate style of modeling
  - Inter-component style
  - Emphasize communication over computation.
  - Sequence Diagrams are basic snippets for describing communication.

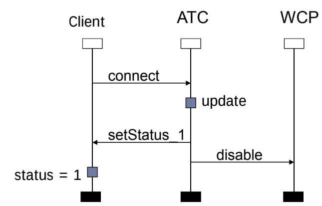
MSC based Models

- MSC = Message Sequence Chart
- Labeled partial order of events
  - Highlights inter-process communications
  - While, FSMs highlight intra-process control flow.



Conventional use of MSCs

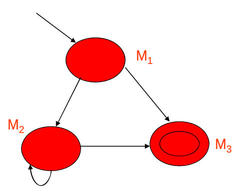
- Describe sample scenarios of system interaction
  - Appears in requirement documents
  - Do not describe "complete" system behavior



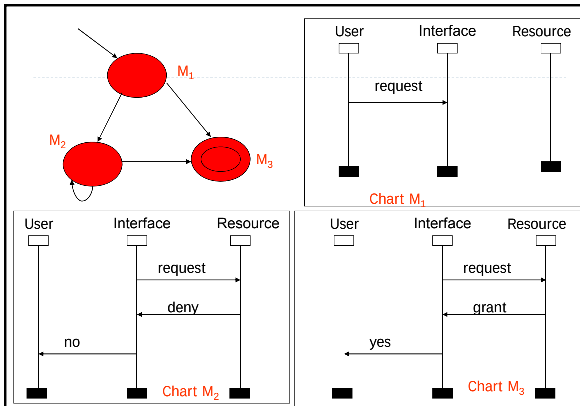
Sample MSC from ATC example

Exercise: Find two incomparable events in this MSC

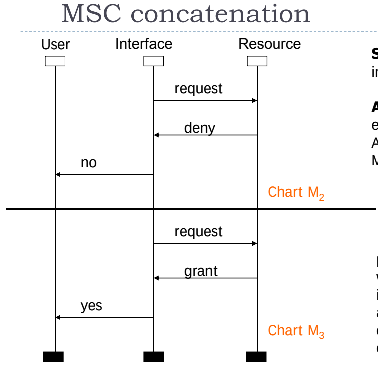
MSC-based design model



Connect MSCs into a graph – Message Sequence Graph (MSG)
Each node of the graph is a MSC.
Need to define the meaning of concatenation of MSCs



MSC concatenation



Synchronous: All events in M2 $\leq$ All events in M3

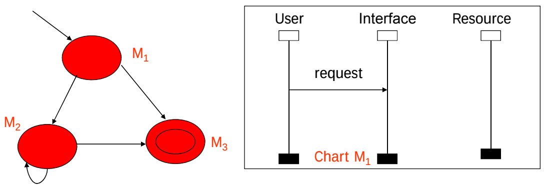
Asynchronous: All events in process p of M2 $\leq$ All events in process p of M3

Interface and Resource processes can finish M3 while User process is still in M2 – provided asynchronous concatenation is considered.

MSC-based design model?

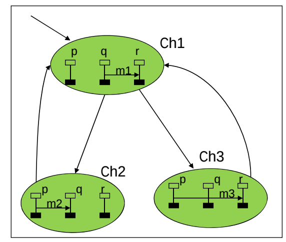
- Complete
  - Complete description of system behavior.
  - MSG achieves this criterion.
- Based on well-established modeling notations.
  - We use UML Sequence Diagrams, which is OK.
- Preferably executable
  - Can simulate the model, and get a feel for how the constructed system will behave!
  - Global simulation of MSG is possible.
  - But not per-process execution !!

Why not executable?



At the end of M1, all the processes agree together to execute either M2 or M3.
One process may go ahead of the others (under asynchronous concatenation).
However, the **decision** of which MSC to execute next must be consistent.
Difficult to generate per-process code to capture this **joint decision**.

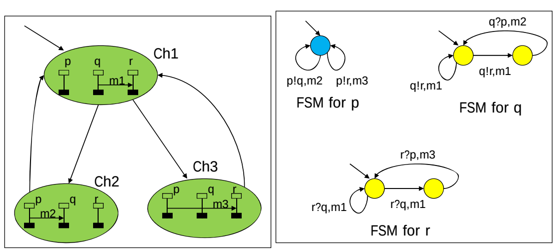
Example MSG



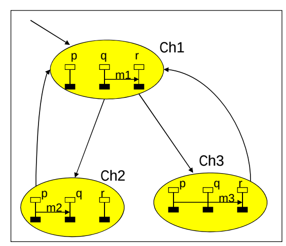
Generates behavior of the form

(Ch1 o (Ch2 + Ch3))$^\omega$

Per-process FSMs

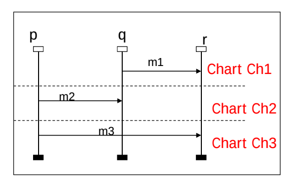


Implied Scenario



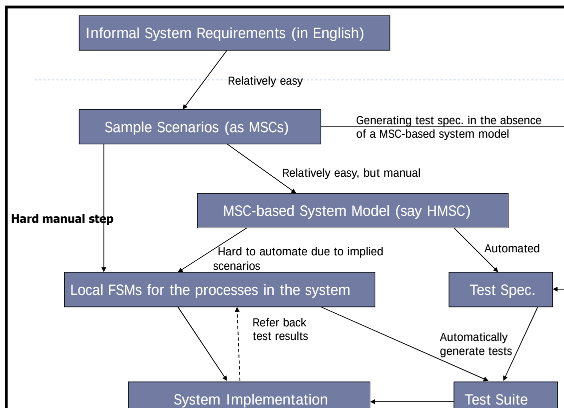
Supposed to generates behavior of the form

(Ch1 o (Ch2 + Ch3))$^\omega$



Putting the notations together

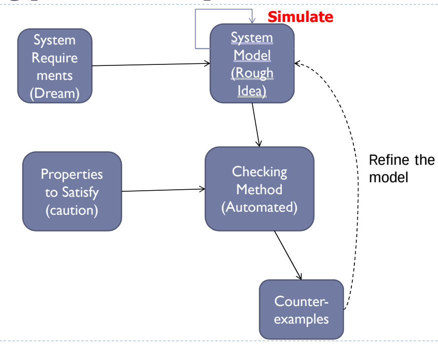
- So, far we have studied 2 notational styles
  - Intra-process style FSM modeling notations
  - Inter-process style MSC-based modeling notation.
- In actual system modeling from English requirements
  - How do they fit together?
  - What roles do they play?
  - Are they both used in parallel?



Organization

- So Far
  - What is a Model?
  - ATC – Running Example
    - Informal Req. at a lab scale.
    - Has subtle deadlock error (see textbook chap 2.3)
  - How to model such requirements
    - Modeling Notations
      - Finite State Machines
      - MSC based models
- Now, how to validate the models
  - Simulations

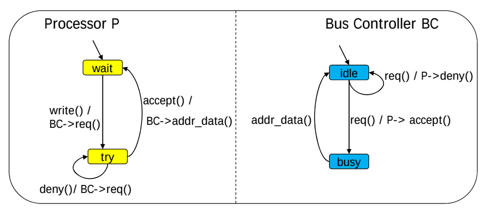
The big picture - recapitulate



FSM Simulations

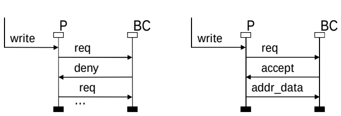
- Monolithic FSM simulation
  - A random walk through the FSM's graph.
- Simulating a composition of FSMs
  - Need to consider the definition of concurrent composition.
  - Keep track of local states of the individual processes.
- Simulating more complex notations
  - UML State Diagrams
  - MSC-based models

Example – State Diagrams



Processor and Bus Controller – what does the example do?

This is what the example does

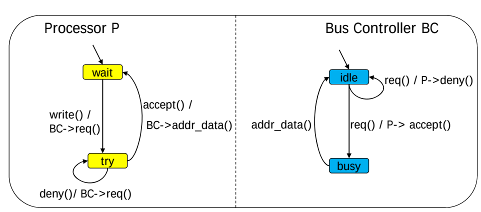


Sample scenarios of the State Diagram shown in the previous slide.

Super-step:
On encountering a write, the sequence of method calls executed is
write, req, (deny, req)*, accept, addr_data

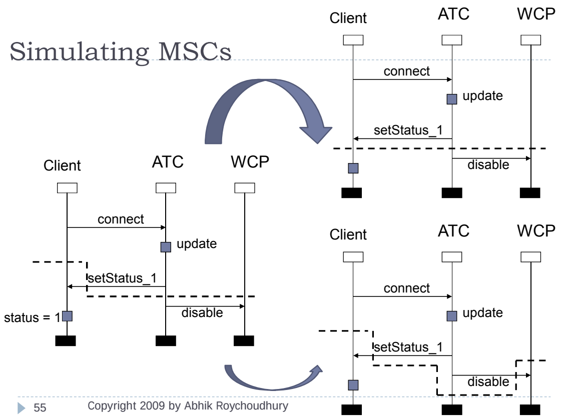
How?

Simulation – State Diagrams



req(), deny(), req(), accept(), addr_data()

Simulating MSCs



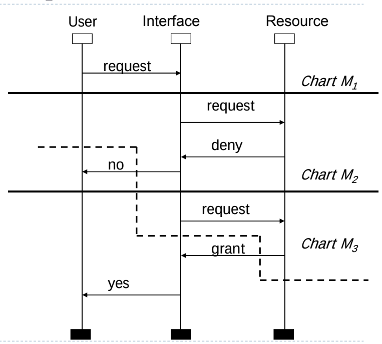
Recap on MSC semantics

- For a sequence of MSCs --- M1, M2
  - **Synchronous concatenation:** All events in M1 $\leq$ All events in M2
  - **Asynchronous concatenation**: All events in process p of M1 $\leq$ All events in process p of M2
- For any msg. m sent from process p to process q
  - **Synchronous message passing**: Send and receive happens in the form of a hand-shake.
  - **Asynchronous message passing**: Sender sends message which is stored in a queue, picked up by receiver later.
- Simulating a sequence of MSCs will need to follow the concatenation & message passing semantics.

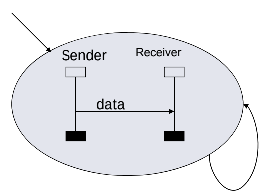
Simulating a sequence of MSCs

Allowed for asynchronous concatenation.

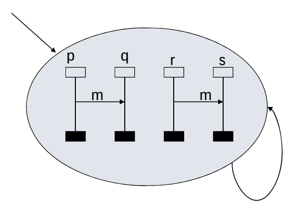
Not allowed for synchronous concatenation.



Simulation requires unbounded memory?



Simulation requires unbounded memory under asynchronous concatenation and asynchronous message passing



Simulation requires unbounded memory under asynchronous concatenation and synchronous / asynchronous message passing.

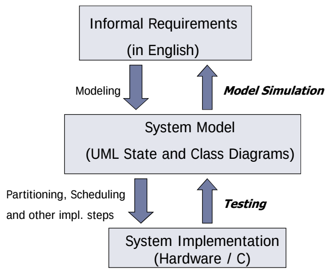
Avoiding unbounded memory

- Spot Exercise:
  - How can we avoid spending unbounded memory while simulating Message Sequence Graphs ?

Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
- How to validate the models
  - So far: Simulations
  - Now: Model-based testing, Model Checking

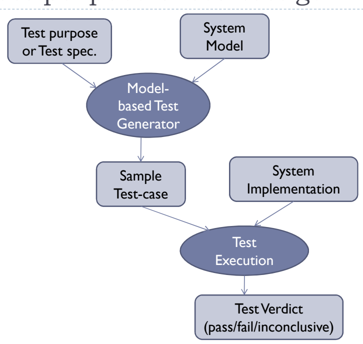
Model-based system development



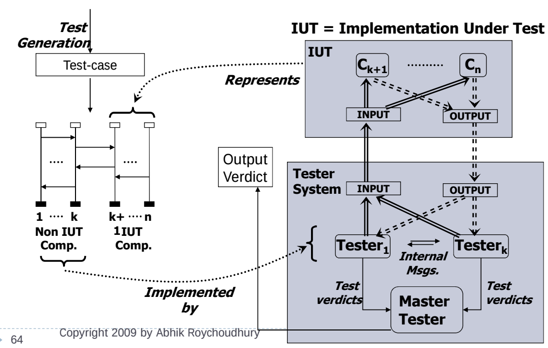
Model-based testing

- Generate test-cases from model, run them on the implementation.
- What are the criteria for generating test cases?
  - Generate a suite of test cases to ensure a structural coverage of the model
    - State coverage, Transition coverage for State Diagrams.
  - Generate test cases from the model based on some test specification
    - How to describe the test specification?
      - *Temporal logic (discussed later)*
    - How to find a test satisfying a test specification?
      - *Model checking (discussed later)*

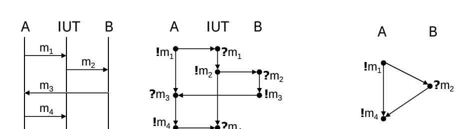
Test-purpose based test gen. & exec.



Test Execution Architecture



Test Execution – (1)

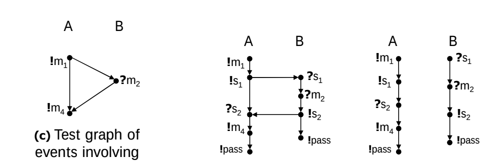


(a) Test-case MSC ***M***

(b) Partial Order of ***M***

(c) Test graph of events involving interaction between tester components and IUT.

Test Execution – (2)

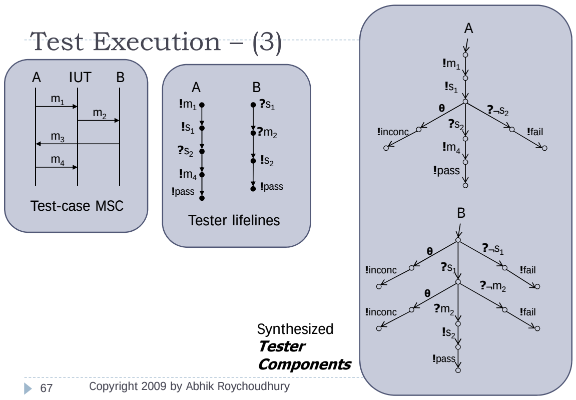


(c) Test graph of events involving interaction between tester components and IUT.

(d) Test graph of ***M***-*Synchronization* events

(e) Local test graphs of tester lifelines ***A*** and ***B***

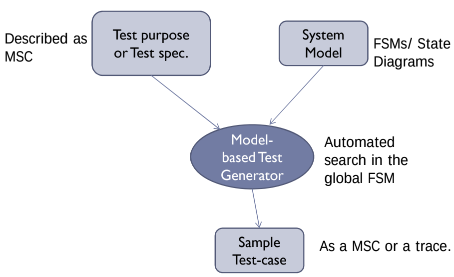
Test Execution – (3)



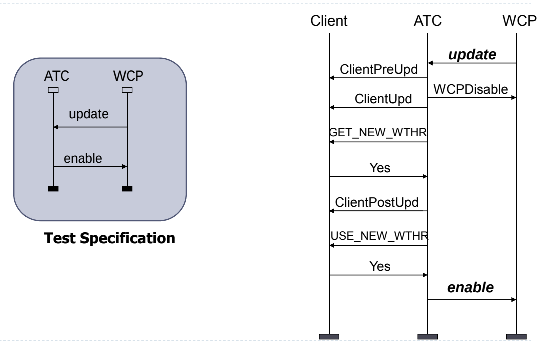
Test Verdicts

- Pass
  - All the tester components convey "Pass" to a Master tester.
- Fail
  - At least one tester component returns fail.
- Inconclusive
  - None of the tester components return fail, and
  - At least one tester component returns inconclusive.

Test-purpose based test generation



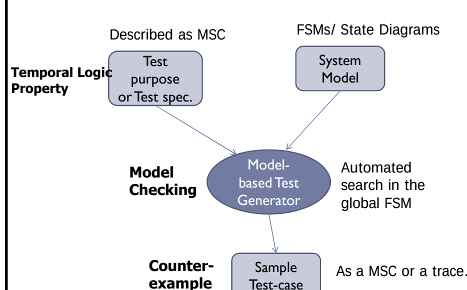
Test spec. & Generated Test



Test spec. & Generated test

- Test spec. is in the form of an MSC M.
- Def. 1
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a contiguous subsequence.
- Def. 2
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a subsequence.
- Which def. did we follow in the previous slide?

Test Generation



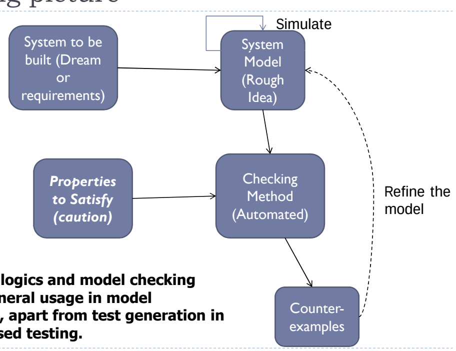
Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
  - How to validate the models
    - Simulations,
    - Model-based testing,
    - Model Checking (discussed now)
      - Temporal logics (the property specification)
      - Checking method
    - Also, model-based testing accomplished by model checking

Temporal Logic

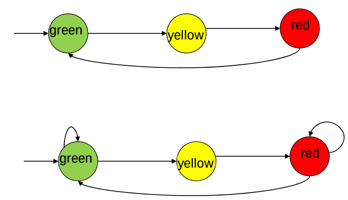
- On ~~June 1 2007~~, I am teaching temporal logics which will be followed by teaching on model checking on ~~June 8, 2007~~
- Teaching of temporal logics occurs ~~1 week~~ before the teaching of model checking.
- Teaching of temporal logics is *always eventually* followed by the teaching of model checking.
- Teaching of temporal logics is *always immediately* followed by the teaching of model checking.

The big picture



Temporal logics and model checking have a general usage in model validation, apart from test generation in model-based testing.

Example System Model



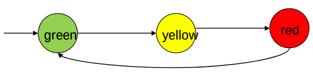
Example properties

- The light is *always* green.
- *Whenever* the light is red, it *eventually* becomes green.
- *Whenever* the light is green, it remains green *until* it becomes yellow.
- …
- Are these properties true for the 2 example models in the previous slide?
  - Let us try the second property for example …

When is a property satisfied?

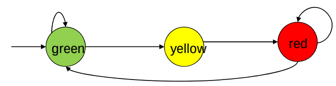
- A property is interpreted on the traces of a system model.
  - Given a trace of the system model x and a property p, we can uniquely determine a yes/no answer to whether x satisfies p.
- A property p is satisfied by a system model M, if all traces of M satisfy p.
- So, given a system model what are its traces?

Traces of a system model



- Only one trace, it has infinite length
  - *(green, yellow, red)* – repeated forever   Written as *(green, yellow, red)* $^{\omega}$

Traces of a system model



- Infinitely many traces, each of infinite length
  - (green)$^{\omega}$ - 1 trace
  - (green)* yellow (red)$^{\omega}$ - infinitely many traces
  - (green)*yellow (red)* (green)$^{\omega}$
  - …
  - (green, yellow, red)$^{\omega}$

Property Specification Language

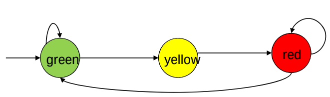
- Properties in our property spec. language will be interpreted over infinite length traces.
  - Finite length traces can be converted into infinite length traces by putting a self-loop at last state.
- A property is satisfied by a system model if all execution traces satisfy the property.
  - In general, we cannot test the property on each exec. trace – infinitely many of them.
  - Model checking is smarter – we discuss it later!
- We formally describe the property spec. lang.

Formally, system model is

- Model for reactive systems
  - M = (S, I, $\rightarrow$, L)
  - S is the set of states
  - S0 $\subseteq$ S is the set of initial states
  - $\rightarrow \subseteq$ S $\times$ S is the transition relation
    - Set of (source-state, destination-state) pairs
  - L: is the labeling function mapping S to 2^{AP}
    - Maps each state s to a subset of AP
    - These are the atomic prop. which are true in s.

Atomic Propositions

- All of our properties will contain atomic props.
  - These atomic props. will appear in the labeling function of the system model you verify.
  - The atomic props. represent some relationships among variables in the design that you verify.
  - Atomic props in the following example
    - **green, yellow, red (marked inside the states with obvious labeling function).**



Linear-time Temporal Logic

- The temporal logic that we study today build on a "static" logic like propositional logic.
  - Used to describ/constrain properties inside states.
- Temporal operators describe properties on execution traces.
  - Used to describe/constrain evolution of states.
- Time is not explicitly mentioned in the formulae
  - Properties describe how the system should evolve over time.
- Does not capture exact timing of events, but rather the relative order of events
- We capture properties of the following form.
  - Whenever event e occurs, eventually event e' must occur.
- We do not capture properties of the following form.
  - At t =2 e occurs followed by e' occurring at t =4.

Notations and Conventions

- An LTL formula φ is interpreted over and infinite sequence of states π = s0,s1, …
  - Use M,π |= φ to denote that formula φ holds in path π of system model M.
- Define semantics of LTL formulae w.r.t. a system model M.
  - **An LTL property φ is true of a system model iff all its traces satisfy φ**
  - **M |=φ iff M,π |= φ for all traces π in system model M**

Notations and Conventions

- M,π |= φ
  - Path $\pi = s_0,s_1,s_2,\ldots$ in model M satisfies property φ
- M,π^k |= φ
  - Path s_k , s_{k+1} , … in model M satisfies property φ
- We now use these notations to define the syntax & semantics of LTL.

LTL - syntax

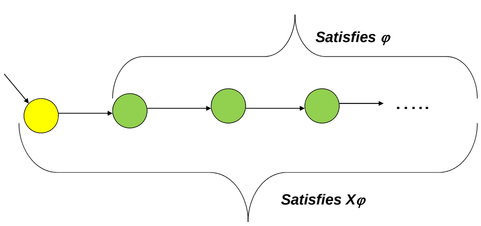
- Propositional Linear-time Temporal logic
- φ = Xφ | Gφ | Fφ | φ U φ | φ R φ | $\neg\varphi$ | $\varphi \wedge \varphi$ | Prop
- Prop is the set of atomic propositions
- Temporal operators
  - X (next – state)
  - F (eventually), G (globally)
  - U (until), R (release)

Semantics of propositional logic

- M,π |= p iff s0 |= p i.e. p $\in$ L(s0) where L is the labeling function of Kripke Structure M
- M, π |= $\neg$ φ iff $\neg$ (M, π |= φ)
- M, π |= φ1 $\wedge$ φ2 iff M, π |= φ1 and M, π |= φ2

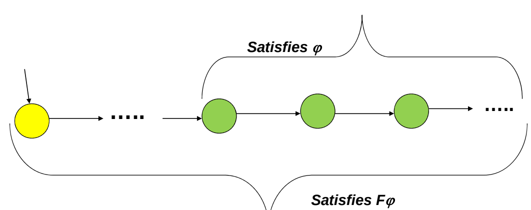
neXt-state operator of LTL

- M,π |= Xφ iff M,π^1 |= φ
  - Path starting from next state satisfies φ



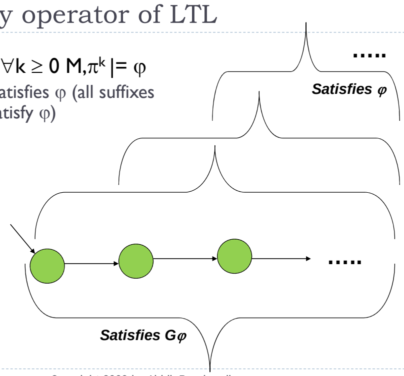
Finally operator of LTL

- M,π |= Fφ iff $\exists k \geq 0$ M,π^k |= φ
  - Path starting from an eventually reached state satisfies φ



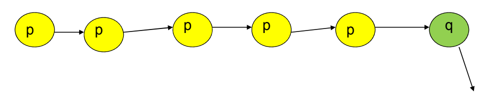
Globally operator of LTL

- $M,\pi \models G\varphi$ iff $\forall k \geq 0\ M,\pi^k \models \varphi$
  - Path **always** satisfies φ (all suffixes of the path satisfy φ)



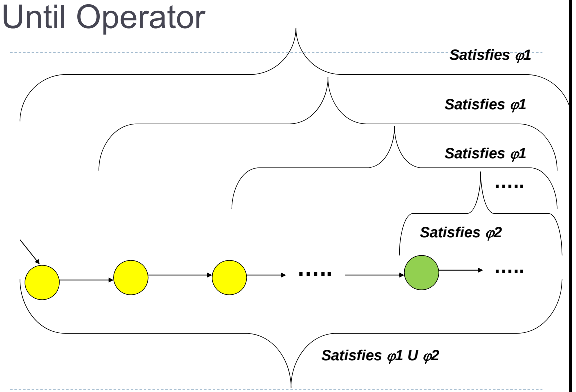
Until operator of LTL

- $M,\pi \models \varphi1\ U\ \varphi2$ iff $\exists k \geq 0$ such that
  - $M,\pi^k \models \varphi2$, and
  - $\forall 0 \leq j < k\ M,\ \pi^j \models \varphi1$



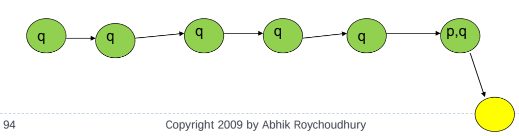
A trace satisfying pU q, where p,q ∈ Prop

Until Operator

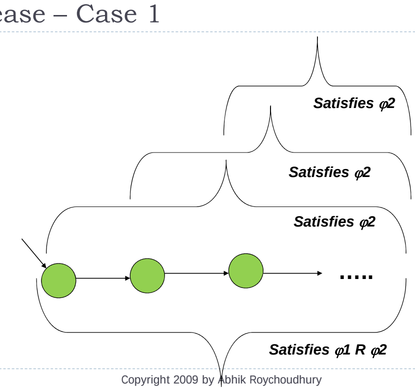


Release operator of LTL

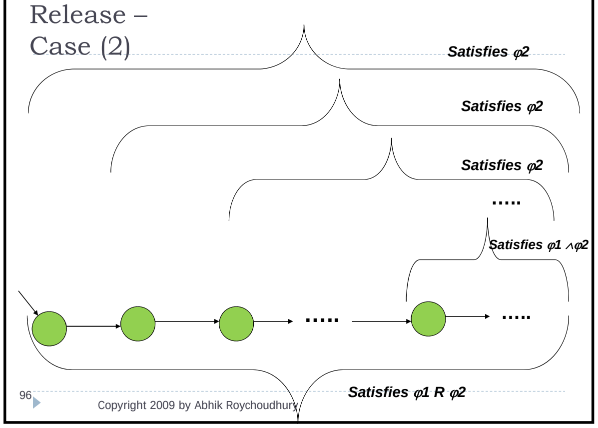
- $M,\pi \models \varphi1\ R\ \varphi2$ iff
  - Either $\forall k \geq 0\ M,\pi^k \models \varphi2$
  - OR both of the following hold
    - $\exists k \geq 0\ M,\pi^k \models \varphi1$
    - $\forall 0 \leq j \leq k\ M,\pi^j \models \varphi2$
- $\varphi1$ releases the req. for $\varphi2$ to hold.



Release – Case 1



Release – Case (2)



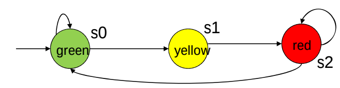
Exercise – (1)

- The light is *always* green.
- *Whenever* the light is red, it *eventually* becomes green.
- *Whenever* the light is green, it remains green *until* it becomes yellow.
- *Whenever* the light is yellow, it becomes red *immediately after*.

- Encode these properties in LTL.

Exercise – (2)

- Check whether the four LTL properties in the previous slide are satisfied by our simple traffic light controller.



LTL Exercise – (3)

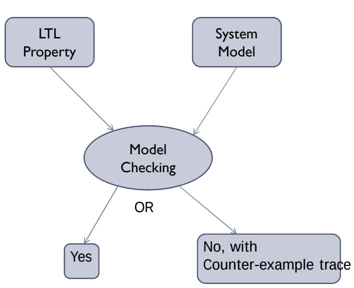
Consider a resource allocation protocol where n processes $P_1,\ldots,P_n$ are contending for exclusive access of a shared resource. Access to the shared resource is controlled by an arbiter process. The atomic proposition req_i is true only when P_i explicitly sends an access request to the arbiter. The atomic proposition gnt_i is true only when the arbiter grants access to P_i. Now suppose that the following LTL formula holds for our resource allocation protocol.

- $G\ (req_i \Rightarrow F\ gnt_i)$

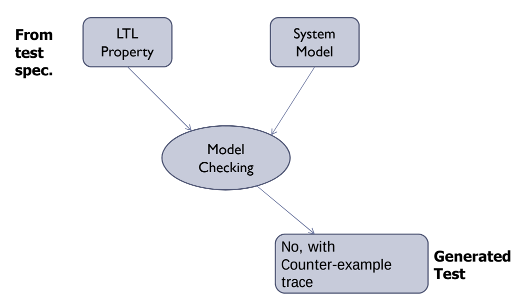
LTL Exercise – (3)

- Explain in English what the property means.
- Is this a desirable property of the protocol ?
- Suppose that the resource allocation protocol has a distributed implementation so that each process is implemented in a different site. Does the LTL property affect the communication overheads among the processes in any way ?

Model Checking



Recap: Model Checking for model-based testing



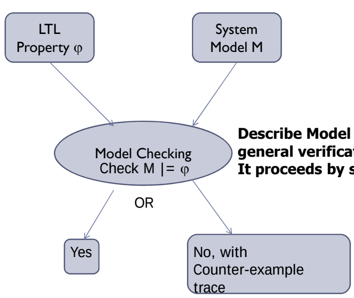
Encoding test specifications

- Def. 1
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a **contiguous** subsequence.
  - Given MSC M,
    - define Lin(M) = set of linearizations of M.
    - For each linearization $\sigma = e_1, e_2, \ldots, e_k$ define
      - Define $prop_\sigma = \mathbf{F}(e_1 \wedge \mathbf{X}(e_2 \wedge \mathbf{X}(\ldots \mathbf{X}(e_k)\ldots)))$
    - Define property φ_M corresponding to M as
      - $\varphi_M = \neg (\vee_{\sigma \in Lin(M)} prop_\sigma)$
- A counter-example to φ_M is a test satisfying M.

Encoding test specifications

- Def. 2
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a subsequence.
  - Given MSC M,
    - define Lin(M) = set of linearizations of M.
    - For each linearization $\sigma = e_1, e_2, \ldots, e_k$ define
      - $n_\sigma = \neg(e_1 \vee e_2 \vee \ldots \vee e_k)$
      - $prop_\sigma = (n_\sigma \mathbf{U} (e_1 \wedge \mathbf{X}(n_\sigma \mathbf{U}(e2 \wedge \mathbf{X}(\ldots \mathbf{X}(n_\sigma \mathbf{U} ek)\ldots))))$
    - Define property φ_M corresponding to M as
      - $\varphi_M = \neg (\vee_{\sigma \in Lin(M)} prop_\sigma)$
- A counter-example to φ_M is a test satisfying M.

Model Checking



Describe Model Checking as a general verification procedure. It proceeds by search.

LTL Model Checking - steps

1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes:Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

Recap: finite-state automata

- $A = (Q, \Sigma, Q_0, \rightarrow, F)$
  - Q is a finite set of states
  - Σ is a finite alphabet
  - $Q_0 \subseteq Q$ is the set of initial states
  - $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation
  - $F \subseteq Q$ is the set of final states.
- What is the Language of such an automaton?

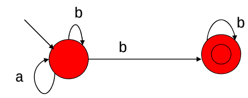
Recap: finite-state automata

- Regular languages:
  - Accept any finite-length string $\sigma \in \Sigma^*$ which ends in a final state.
- ω-regular languages:
  - Accept any infinite-length string $\sigma \in \Sigma^\omega$ which visits a final state infinitely many times.
- Set of strings accepted = *Language* of the automata.

LTL properties to automata

- Given a LTL property p
  - we want to convert p to an automata A_p s.t.
  - Language(A_p) = strings / traces satisfying p
- LTL properties are checked over infinite traces.
  - Given an infinite trace σ and a LTL property p, we can check whether $\sigma \models p$
- To convert LTL properties to finite-state automata, consider automata accepting inf.-length traces.
  - Language(A_p) is ω-regular, not regular.

LTL properties to automata



- Meaning as a regular language
  - $(a+b)^*b^+$
  - All finite length strings ending with b
- Meaning as a ω-regular language
  - All infinite length strings with finitely many a

LTL properties to automata

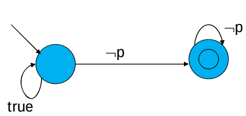
- Given a LTL property φ
  - We convert it to a ω-regular automata A_φ
- Language(A_φ) = $\{\sigma \mid \sigma \in \Sigma^\omega \wedge \sigma \models \varphi\}$
  - Language(A_φ) is defined as per the ω-regular notion of string acceptance. It accepts inf. length strings.
  - All infinite length strings satisfying φ form the language of A_φ
  - Whether an infinite length string satisfies φ (or not) is defined as per LTL semantics.

Recall: LTL Model Checking

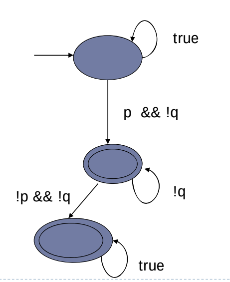
1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes:Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

Example: Verify GFp

- Construct negation of the property
  - $\neg GFp \equiv FG\neg p$
- Construct automata accepting infinite length traces satisfying $FG\neg p$



Example: LTL property to automata

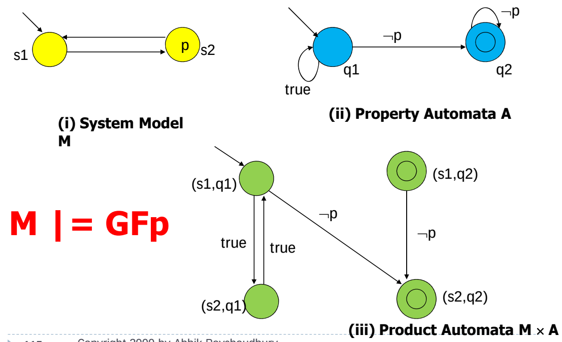


Represents negation of the LTL property

G (p $\Rightarrow$ (p U q))

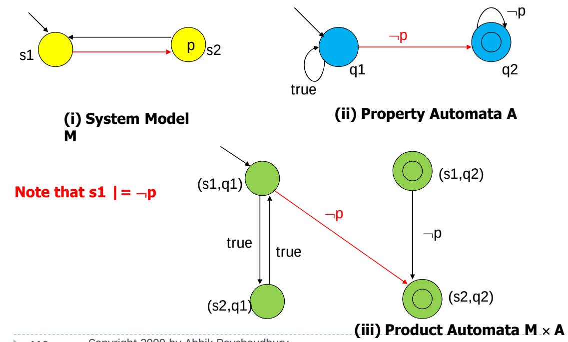
Internally generated by SPIN when the user wants to verify the LTL property.

Product Automata



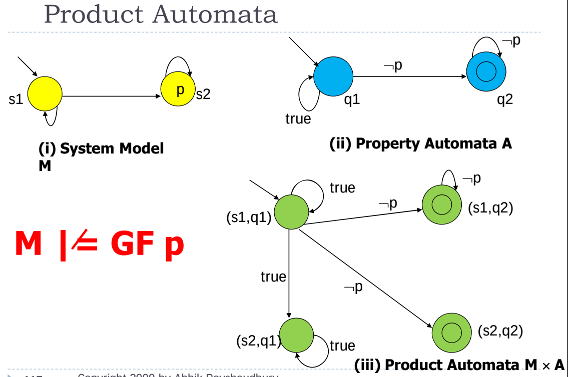
M |= GFp

Product Automata Construction



Note that s1 |= ¬p

Product Automata



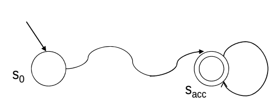
M |≠ GF p

Recall: LTL Model Checking

1. Consider ¬φ. None of the exec. traces of M should satisfy ¬φ.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying ¬φ
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes: Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

Emptiness Check

- Language($M \times A_{\neg\varphi}$) = empty-set?
  - Is there any trace which visits one of the accepting states of the product automata infinitely many times?
  - Look for accepting cycles.



Emptiness Check

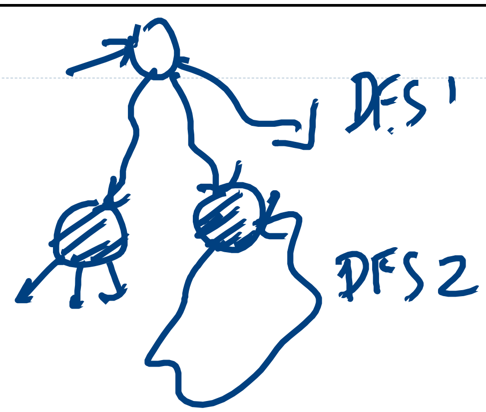
- Perform DFS from initial state until you reach an accepting state s_{acc}
- When you reach s_{acc}, remember s_{acc} in a global var. and start a nested DFS from s_{acc}
  - Stop the nested DFS if you can reach s_{acc}
- If no accepting cycles are found, report yes.
- If accepting cycles are found
  - Concatenate the two DFS stacks and report it as counter-example trace of the LTL property.
- This algo. is implemented in SPIN model checker.

Nested DFS – step 1

- procedure dfs1(s)
  - push s to Stack1
  - add {s} to States1
  - if accepting(s) then
  - States2 := empty; seed := s; dfs2(s)
  - endif
  - for each transition s → s' do
  - if s' ∉ States1 then df1(s')
  - endfor
  - pop s from Stack1
- end

Nested DFS – step 2

- procedure dfs2(s)
  - push s to Stack2
  - add {s} to States2
  - for each transition s → s' do
  - if s' = seed then report acceptance cycle
  - else if s' ∉ States2 then df2(s')
  - endif
  - endfor
  - pop s from Stack2
- end



Organization

- So Far
  - What is a Model?
  - ATC – Running Example
  - How to model such requirements
  - How to validate the models
    - Simulations,
    - Model-based testing,
    - Model Checking
    - Model Checkers
      - SPIN

SPIN

- A tool for modeling complex concurrent and distributed systems.
- Provides:
  - Promela, a protocol meta language
  - A model checker
  - A random simulator for system simulation
  - Promela models can be automatically generated from a safe subset of C.

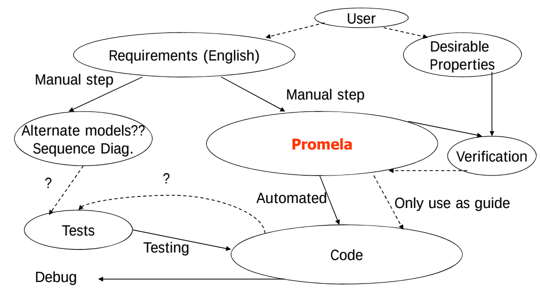
Our Usage

- Learn Promela, a low-level modeling language.
- Use it to model simple concurrent system protocols and interactions.
- Gain experience in verifying such concurrent software using the SPIN model checker.
- Gives a feel (at a small scale)
  - What are hard-to-find errors ?
  - How to find the bug in the code, once model checking has produced a counter-example ?

Another possible usage?

- Write programs in C
  - Or C programs for each process in a distributed sys.
- Generate Promela Code from C automatically.
- Use the model checker of SPIN to search through the model represented by the Promela code (automatic verification).
- But …
  - C → Promela tool relatively new.
  - Promela itself is useful for modeling protocols etc.

Our Usage



Features of Promela

- Concurrency
  - Multiple processes in a system description.
- Asynchronous Composition
  - At any point one of the processes active.
  - Interleaving semantics
- Communication
  - Shared variables
  - Message passing
    - Handshake (synchronous message passing)
    - Buffers (asynchronous message passing)

Features of Promela

- Within a process
  - Non-determinism : supports the situation where all details of a process may not be captured in Promela model.
  - Standard C-like syntax
    - Assignment
    - Switch statement
    - While loop
    - Guarded command
      - Guard and body may not evaluated together, that is, atomically.

Example 0

```
byte state = 0;

proctype A()
{ byte tmp;

    (state==0) -> tmp = state;
     tmp = tmp+1;
     state = tmp;
}

init { run A() ; }
```

state : Global Variable

tmp : Local Variable

(state==0) -> tmp = state is a guarded command (blocked if the guard is false).

Only one process created.

Final value of **state** is 1

But SPIN allows multiple processes to be created.

Example 1

```
byte state = 0;

proctype A()
{ byte tmp;

    (state==0) -> tmp = state;
     tmp = tmp+1;
     state = tmp;
}

init { run A() ; run A(); }
```

We need to define how processes are scheduled.

SPIN's process scheduling

- All processes execute concurrently
- Interleaving semantics
  - At each time step, only one of the "active" processes will execute (non-deterministic choice here)
  - A process is active, if it has been created, and its "next" statement is not blocked.
  - Each statement in each process executed atomically.
  - Within the chosen process, if several statements are enabled, one of them executed non-deterministically.
    - We have not seen such an example yet !

SPIN Execution Semantics

- Select an enabled transition of any thread, and execute it.
- A transition corresponds to one statement in a thread.
  - Handshakes must be executed together.
    - *chan x = [0] of {…};*
    - *x!1* || *x?data*

SPIN Execution Engine

```
while ( (E = executable(s)) != {})
    for some (p,t) ∈ E
    {   s' = apply(t.effect, s);   /* execute the chosen statement */
        if (handshake == 0)
        {       s = s' ;
                p.curstate = t.target
        }
        else{ …
```


SPIN Execution Engine

```
                /* try to complete the handshake */
                E' = executable(s'); /* E' ={} ⇒ s unchanged */
                for some (p', t') ∈ E'
                {     s = apply(t'.effect, s');
                        p.curstate = t.target;
                        p'.curstate = t'.target;
                }
                handshake = 0
        }  /* else */
    } /* for some (p, t) ∈ E */
} /* while ((E = executable(s)) … */
while (stutter) { s = s }
```


Example 2

```
bit flag;                           init {
byte sem;                               atomic{
proctype myprocess(bit i)                   run myprocess(0));
{  (flag != 1) -> flag = 1;                 run myprocess(1));
   sem = sem + 1;                           run observer();
   sem = sem – 1;                       }
   flag = 0;                        }
}
proctype observer() {
    assert( sem != 2 );
}
```

All three processes
Instantiated together

Issues

- Initial values of sem, flag not given
  - All possible init. values used for model checking.
- The system being verified is the asynchronous composition
  - myprocess(0) || myprocess(1)
- The property is the invariant
  - G sem ≠ 2
- Local & global invariants can be specified inside code via assert statements.

assert

- Of the form assert B
  - B is a boolean expression
  - If B then no-op else abort (with error).
- Can be used inside a process (local invariants)
  - proctype P(…) { x = … ; assert(x != 2); ….}
- Or as a separate observer process (global invariants)
  - proctype observer(){ assert(x != 2); }

Example 3

```
bit flags[2];
byte sem, turn;
proctype myprocess(bit id) {
   flags[id] = 1;
   turn = 1 – id;
   flags[1-id] == 0 || turn == id;
   sem++;
   sem--;
   flags[id] = 0;
}

init() {
   atomic{
      run myprocess(0);
      run myprocess(1);
      run observer(); }
}
proctype observer() {
      assert( sem != 2 );
}
```


Issues

- Can you use SPIN to prove mutual exclusion ?
  - What purpose does turn serve ?
- Arrays have been used in this example.
  - Flags is global, but each element is updated by only one process in the protocol
  - Not enforced by the language features.
- Processes could alternatively be started as:
  - active proctype myprocess(…) {
  - Alternative to dynamic creation via run statement

So far, in SPIN

- Process creation and interleaving.
- Process communication via shared variables.
- Standard data structures within a process.
- Assignment, Assert, Guards.
- NOW …
  - Guarded IF and DO statements
  - Channel Communication between processes
  - Model checking of LTL properties

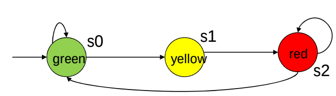
Will this loop terminate?

```
byte count;

      proctype counter()
      {
            do
            :: count = count + 1
            :: count = count - 1
            :: (count == 0) -> break
            od;
      }
```

Enumerate the reasons for non-termination in this example

This loop will not terminate

```
active proctype TrafficLightController() {
      byte color = green;
      do
       :: (color == green) -> color = yellow;
       :: (color == yellow) -> color = red;
       :: (color == red) -> color = green;
       od;
}
```

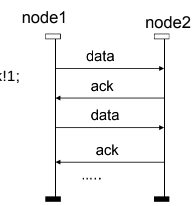


Channels

- SPIN processes can communicate by exchanging messages across channels
- Channels are typed.
- Any channel is a FIFO buffer.
- Handshakes supported when buffer is null.
- chan ch = [2] of bit;
  - A buffer of length 2, each element is a bit.
- Array of channels also possible.
  - Talking to diff. processes via dedicated channels.

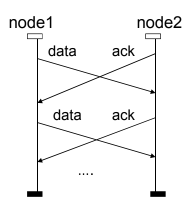
Example with channels

```
chan data, ack = [1] of bit;

        proctype node1() {                proctype
node2() {
        do                                   do
        :: data!1;                           :: ack!1;
        :: ack?1;                            ::
data?1;
        od                                   od
        }                                 }

        init{ atomic{
              run node1(); run node2();
           }
        }
```



Example with channels

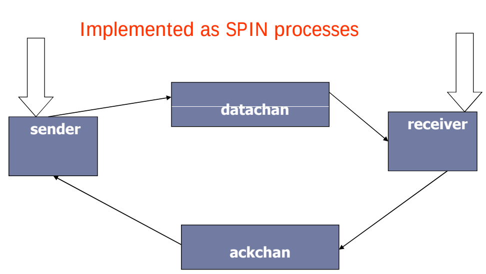
```
chan data, ack = [1] of bit;

        proctype node1() {                proctype
node2() {
        do                                   do
        :: data!1;                           :: ack!1;
        :: ack?1;                            ::
data?1;
        od                                   od
        }                                 }

        init{ atomic{
              run node1(); run node2();
           }
        }
```



More Involved Example

- Alternating Bit Protocol
  - Reliable channel communication between sender and receiver.
  - Exchanging msg and ack.
  - Channels are lossy
  - Attach a bit with each msg/ack.
  - Proceed with next message if the received bit matches your expectation.

ABP Architecture



Sender code

- chan datachan = [2] of { bit };
- chan ackchan = [2] of { bit };

```
active proctype Sender()
{   bit out, in;
    do
     :: datachan!out ->
              ackchan?in;
              if
              :: in == out -> out = 1- out;
              :: else fi
     od
}
```

Receiver code

- chan datachan = [2] of { bit };
- chan ackchan = [2] of { bit };

```
active proctype Receiver()
{   bit in ;
     do
     :: datachan?in -> ackchan!in
     :: timeout -> ackchan!in
     od
}
```


Timeouts

- Special feature of the language
  - Time independent feature.
    - Do not specify a time as if you are programming.
  - True if and only if there are no executable statements in any of the currently active processes.
  - True modeling of deadlocks in concurrent systems (and the resultant recovery).

Model Checking in SPIN

- (P1 || P2 || P3) $|= \varphi$
  - P1, P2, P3 are Promela processes
  - φ is a LTL formula
- Construct a state machine via
  - M, asynchronous composition of processes P1, P2, P3
  - $A_{\neg\varphi}$, representing $\neg\varphi$
- Show that "language" of $M \times A_{\neg\varphi}$ is empty
  - No accepting cycles.
- All these steps have been studied by us !!

Specifying properties in SPIN

- Invariants
  - Local: via assert statement insertion
  - Global: assert statement in a monitor process
- Deadlocks
- Arbitrary Temporal Properties (entered by user)
  - SPIN is a LTL model checker.
- Why Verify, not Test?
  - "I have been fishing all day, I have found a number of fish since the morning, I cannot find any more now, I am pretty sure, there aren't any left!"
  - Bug finding techniques will ensure worse coverage than fishing in a small pond.

Connect system & property in SPIN

- **System model**
  - int x = 100;
  - active proctype A()
  - { do
  -     :: x %2 -> x = 3*x+1
  -     od
  - }
  - active proctype B()
  - { do
  -     :: !(x%2) -> x = x/2
  -     od
  - }
- **Property**
  - GF (x = 1)
- Insert into code
  - #define q (x == 1)
- Now try to verify GF q

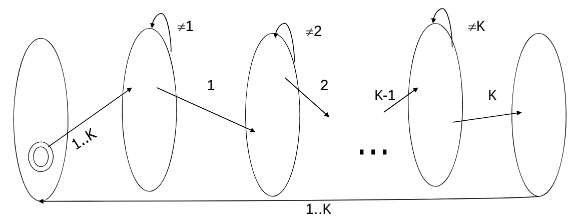
Model Checking in SPIN

- SPIN does not use SCC detection for detecting acceptance cycles (and hence model checking)
- The nested DFS algorithm used in SPIN is more space efficient in practice.
  - SCC detection maintains two integer numbers per node. (*dfs* and *lowlink* numbers)
  - Nested DFS maintains only one integer.
    - This optimization is important due to the huge size of the product graph being traversed on-the-fly by model checker.
- Find acceptance states reachable from initial states (DFS).
- Find all such acceptance states which are reachable from itself (DFS).
- Counter-example evidence (if any) obtained by simply concatenating the two DFS stacks.

Enforcing Fairness

- All LTL counter-examples are acceptance cycles.
- No point in reporting acceptance cycles which arise out of unfair scheduling.
- If there are K active concurrent processes in the Promela model being verified, all of these processes should have transitions within the acceptance cycle found.
- --- Weak fairness requirement
- How to enforce this requirement?
- Linear blow-up in state space.
- Create (K+2) copies of $(P_1 || \ldots || P_K) \times M(\neg\varphi)$
- φ is the property being verified

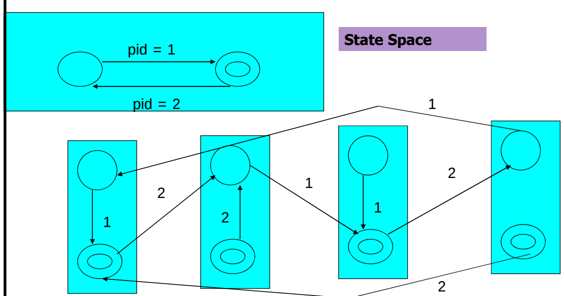
Enforcing Fairness



Also, if state s in i^{th} copy has no outgoing transition by P_i add the transition

s in i^{th} copy $\longrightarrow$ s in $(i+1)^{th}$ copy

Enforcing Fairness



More readings on SPIN

- http://spinroot.com/spin/Man/Manual.html
  - SPIN manual
- The model checker SPIN (Holzmann)
  - IEEE transactions on software engineering, 23(5), 1997.
- http://spinroot.com/spin/Doc/SpinTutorial.pdf
  - SPIN beginner's tutorial (Theo Ruys)
- ``The SPIN model checker: primer and reference manual", by Holzmann (mostly chapters 2,3,7,8)

