Module Folder submitted by Abhik Roychoudhury

CS4271 Critical Systems and their Verification

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

Course Contents:

In this folder, I will present my teaching of

CS 4271: Critical Systems and their Verification

The course is aimed to teach the students an initial idea about formal reasoning and verification. The aim is to study scalable reasoning techniques which can be used for formally validating safety-critical systems. Being part of the Computer Engineering Program, this course focuses on validation of embedded software and systems. Embedded systems are computing systems which are integrated often as part of a larger physical system. Some of the common characteristics of these class of systems include continuous interaction of the computing system with its physical environment, and the computing system implementing a specific dedicated functionality as a combination of hardware and software. Some of the topics / questions covered in this course include the following.

- Complete coverage of the major abstraction levels of embedded systems design.
- Integrates formal techniques for validation for hardware/software with debugging methods for embedded systems;
- Includes practical case studies to answer the questions: does a design meet its requirements, if not, then which parts of the system are responsible for the violation, and once they are identified, then how should the design be suitably modified?

Teaching Style:

The focus of this course is on validation methods to help build system which are reliable in terms of performance and functionality. Of course an important task in the teaching of such modules is to relate the mathematical concepts in validation methods to the actual practice of computer system design. However, this relationship is often brought out in an extreme way (in textbooks or other teaching materials) — *dramatic* historical incidents which led to the *spectacular failure* of computer systems due to the lack of verification are presented to the students. This clearly attracts the students' attention, but this interest becomes difficult to retain, possibly because the historical incidents seem far removed.

Instead, to motivate more mathematical topics like formal verification, I felt that the students need to understand the impact of formal verification in system design; they do not need to be surprised. So, in CS4271, I start with existing practices of verification/validation and how they are intrusive to the design process. I then discuss how a model-based formal approach can help the design cycle, without referring to historic disasters that occurred due to lack of formal verification*. At this point, sometimes the students may question about the veracity of the model-based approach, since it may take effort on the part of the* *designers to construct the models*. This brings the class discussion to an interesting point, since we can then discuss how the models can be automatically extracted from the system implementation – thereby allowing validation methods to be employed automatically without manual involvement of the system designer.

This results in a rather different pedagogical style, where the aim is to discuss the system design cycle with the students rather than surprising the students with the power of formal verification and logic. Once the learners appreciate the differences in the techniques, they develop a more long-lasting interest. This results in a learning process with more active participation from teacher / students — a goal which I always try to achieve in my classes. Furthermore, such classroom discussions have, from time to time, spurred my own research as well as pedagogy, the results of which have later been integrated into teaching. In particular, in the process of teaching CS 4271 I have also over the years derived a more holistic understanding of embedded systems validation, based on which I have now written a textbook to help the teaching of these issues. This is an example of how past teaching experiences can enhance the current teaching practices.

Student Learning Objectives:

The learning objectives of CS 4271 were as follows:

(a) Focused approach: study widely used validation methods (specifically those that are used in the industry by practitioners)
(b) Hands-on approach towards formal verification, with initiation to *validation tools*
(c) Understanding and realizing that embedded system design/validation cannot simply focus on functionality – the designers also need to study other characteristics such as performance
(d) Appreciation and basic understanding of more theoretical topics like temporal logic which have a direct relevance in building the validation methods.

Assessment Plan:

The assessment plan was geared towards a healthy mix of continuous evaluation and examinations. Apart from *one* midterms and a final examination, the students did hands-on work. The hands-on work comes in the form of three different labs or programming assignments.

- Lab1 – Modeling using the Rhapsody tool [here the students do UML modeling and simulation of the UML models]
- Lab2 - Formal verification using the SPIN model checker
- Lab3 - Performance or timing analysis of embedded software

[this lab shows the students how building correct embedded systems goes beyond ensuring correct functionality – such as ensuring correct timing]

The overall division is thus as follows – Midterm 20%, Final 50%, Labs 30% (three labs, each 10%)

In certain offerings of the module the division was slightly altered such as

Midterm 20%, Final 45%, Labs 35%

Student Profile and Student feedback:

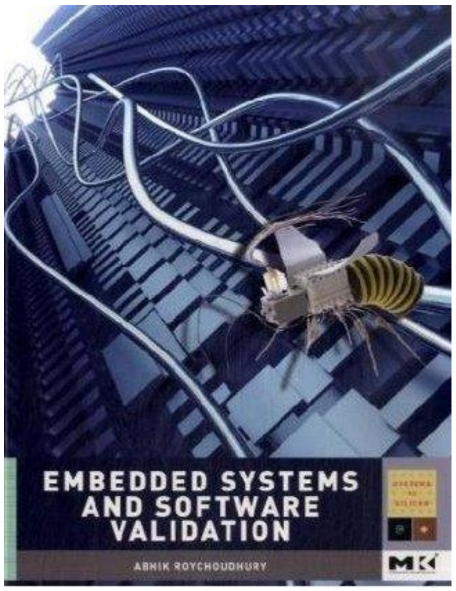
Most of the students are from Computer Engineering program, since CS4271 is offered as an elective in the Computer Engineering program.The enrolment has varied between 20–75. The enrolment, and teaching feedback rating in the last five offerings are as follows.

Academic Year	CS4271 Enrolment	My feedback rating
2012-13	19	4.33
2011-12	35	4.33
2010-11	39	4.07
2009-10	38	4.38
2008-09	74	4.14

Teaching Strategy and Innovative Practices:

As part of my teaching of CS 4271, I have been spurred in terms of both my research as well as pedagogical activities. I will describe how my past teaching has influenced my current pedagogy with some simple examples.

[Textbooks:]



CS 4271 always existed as an elective in the Computer Engineering program. When I took up the teaching of the course, the lack of books which cover the different aspects of system validation was a big problem. Some books would be extremely formal in nature, covering only formal verification methods. Some books would be rather informal and suggest only rules-of-thumb for system validation. Moreover, the different levels of abstraction at which the validation methods would perform were also different in different books. Some books would exclusively look at system level methods, whereas some books were more of a software engineering nature. Hence I felt a real need to bring some amount of structure among the different validation methods, the correctness criteria they target (functionality or performance or other criteria), the formality of the methods, and finally the level of abstraction at which the validation methods will be employed. Around the same time (in 2007), Morgan Kaufmann/Elsevier approached me to publish a book exactly on this topic, as part of its System-on-Silicon series. I authored this textbook, and i*t is now being used in the current offering of CS 4271.* The book is entitled *Embedded Systems and Software Validation, and a webpage of the book appears at* *http://www.comp.nus.edu.sg/~abhik/Book/AbhikESBook09.html*

Apart from being adopted in CS 4271 – the newly published book has also been adopted in several other universities in USA, Europe, South Korea, and New Zealand. The book has also been translated to Chinese at the initiative of Tsignhua University Press, and the Chinese translation has been adopted Details of the authoring of the book, as well as emails from instructors using the book, have been provided with the main dossier. I am attaching the table of contents from the book, along with this module folder.

[Case Studies:]

When I started teaching this module, I found a lack of classroom examples through which I could elaborate some practical experiences in validating (substantial) parts of an embedded system. Around the same time, in the context of some research work in modeling reactive embedded systems I have been studying ARM's AMBA system-on-chip bus protocol. Working with a senior undergraduate student (whose final year project I was supervising), I figured that the AMBA protocol had some subtle corner cases leading to a deadlock which are hard to find using informal manual reasoning. This work was published in DATE, a leading conference in embedded system/design automation. More importantly, it provided with a good hands-on case study which I could provide to my students in the CS 4271 course. It serves to illustrate exactly what role automated formal reasoning techniques can hope to play in practice for debugging subtle design errors. I have therefore, included this as a case study in later offerings of CS 4271.

Things that are now being done differently:

In the past, I conducted a mid-term module feedback using IVLE. Based on the feedback, I made some changes to the course, including abolishing the official consultation hours. The students were more comfortable seeking e-mail appointments, or asking questions in the revision hour so I stayed with this arrangement. This allows the students to come in at times of their convenience.

Appendices in CS4271 module folder:

1. *Lesson Plan:* The module covers validation of embedded systems from both functionality and performance perspective. Practical case studies are used throughout the module. In particular the students are introduced to three concrete tools – one for system modeling and simulation (the first programming assignment or lab), one for formal verification via model checking (the second programming assignment or lab), and one for timing analysis (the third programming assignment or lab).
2. *Sample Lecture notes:* I provide the lecture slides for systems modeling which is covered in the earlier part of the module. Lot of examples are provided to cover the concepts.
3. *Sample Revision hour:* The weekly lecture comprises of 2 hours lecture and 1 hour of revision. Usually the revision hour consists of discussing some exercises to understand the concepts in the lecture. I attach the revision hour corresponding to the lecture on modeling which I have attached.
4. *Sample Lab:* I am attaching a sample lab assignment on system modeling. It gives a small fragment of the informal requirements in a real control system – it is a small and simplified fragment of a flight control software which has been deployed in large airports such as the Dallas Fort Worth Airport. The students need to model the informal requirements in terms of formal models and then find subtle errors (such as deadlocks) in the informal requirements.
5. *Sample midterm examination, with solutions*
6. *Sample final examination, with solutions*
7. *Table of Contents and Preface of the text-book authored by me. The text-book authoring was triggered by the need felt from earlier offerings of the module.*

CS 4271 Lesson Plan

CS 4271- Critical Systems and their Verification Lesson Plan for 2011-12 Semester 2

General Information: Instructor, Teaching Assistant, Textbook.

Assessment: Labs 35%, Midterm 25%, Final Exam 40%

DATES	***Lecture Summary***	***Readings***	***Resources***
Week 1	**General introduction** of embedded systems and the need for validation General discussion on the statechart notation will be conducted in the later part of the lecture. **First Lab Posted (using Rhapsody for Statechart modeling): Description, Tool guide, Tool usage notes**	Textbook Chapter 1 The following papers may also be usefulhttp://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/Statecharts.pdf http://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/OOStatecharts.pdf	First part of lecture Second part of lecture Showing Rhapsody tool to students
Week 2	**Model Validation** - Discussion of Modeling notations	Chapter 2.3 - 2.5 of textbook	Lecture Notes
Week 3,	**Model validation** - Model based testing, test specifications	Chapter 2.6,2.7 of textbook	Lecture Notes

Week 4	**Model validation** -Model checking	Chapter 2.8 of textbook	Lecture Notes (LTL) Lecture notes(MC)
Week 5	**SPIN Model Checker** **Lab 1 is due - no extensions** **Lab2 (on SPIN) is given out.**	Chapter 2 (all) READING: http://spinroot.com/spin/whatispin.html contains a wealth of pointers on SPIN.	Lecture Notes
Week 6	**SMV Model Checker** The modeling language of this checker is in the style of hardware. Time permitting, a real-life case study using SMV is also discussed.	READING: http://www.kenmcmil.com/smv.html http://www.cs.cmu.edu/~modelcheck/smv.html	Lecture Notes
Recess (19 - 27 Feb) **Lab 2 is due - no extensions**			
Week 7,	**Midterm Examination** **Lab 3 (on SMV) is also given out in this week.**	--	Midterm solutions to be posted
Week 8	**Performance validation** - Worst-case Execution time analysis of embedded software. In this lecture, we introduce the problem of timing analysis and why it is important for embedded system design	Chapter 4.1, 4.2	Lecture Notes

Week 9	**Performance validation** - WCET analysis methods via Timing Schema and Integer Linear Programming. Some in-class exercises on timing analysis will be discussed towards the end of this lecture. These exercises will be given out in class, and posted in IVLE after the class.	As in past week	Lecture Notes.
Week 10	**Performance validation (System level)**: We wrap up our discussion on performance validation with discussion on scheduling methods such as Rate monotonic Scheduling (RMS) and Earliest Deadline First (EDF). System support for time-predictable execution will also be studied. **Lab 3 (on SMV) is due - no extensions.** **Lab 4 (timing analysis using Chronos) is given out**	Chapter 4.3 - 4.5	Lecture Notes
Week 11	**Software Testing and Debugging** - we focus on the age-old practice of software testing and how find the error root-cause when a test case fails (i.e. debugging).	Chapter 5.1	Lecture Notes
Week 12	**Software Testing and Debugging** - We conclude our discussions by focusing on directed testing via symbolic execution. Last year's question paper was also discussed as revision exercise in class, and the solutions are posted.	Chapter 5.1	Lecture Notes

Week 13	**Communication Validation**: In this final lecture, we discuss validation of inter-component communication among embedded system components. **Lab 4 (timing analysis using Chronos) is due - no extensions**	Chapter 3	Lecture notes

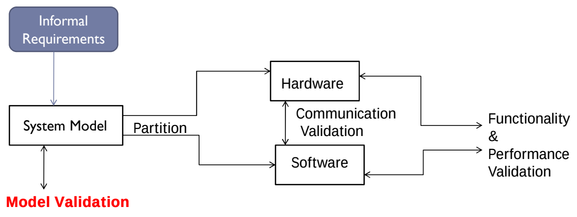
CS 4271 Sample Lecture Notes (on System Modeling)

Embedded Systems and Software Validation
Model Validation

Abhik Roychoudhury
National University of Singapore

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

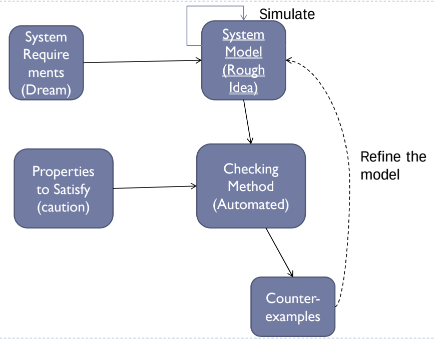
Different kinds of ES Validation



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.

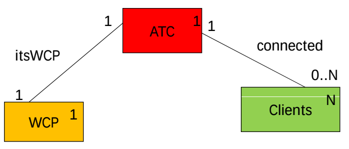
The big picture in modeling


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

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



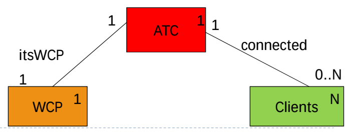
Overall System Structure, Behavior not shown.

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.**



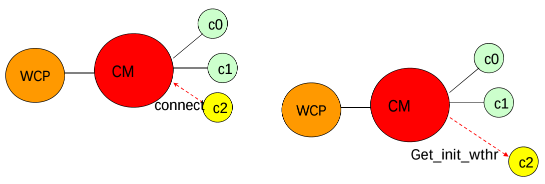
ATC – the example control sys.

- 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)

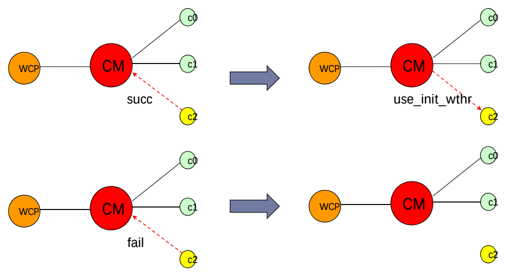
Behavior of ATC example

- 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

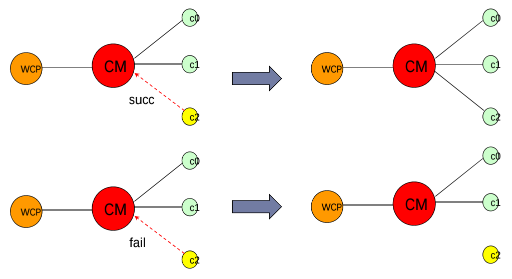
Client Initialization



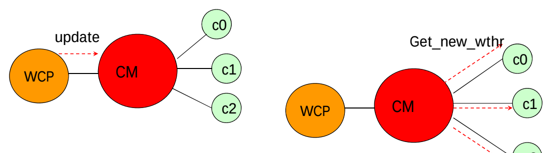
Client - Initialization



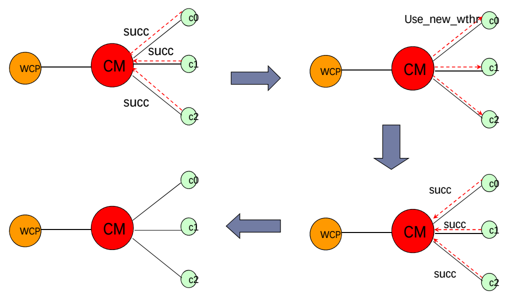
Client Initialization



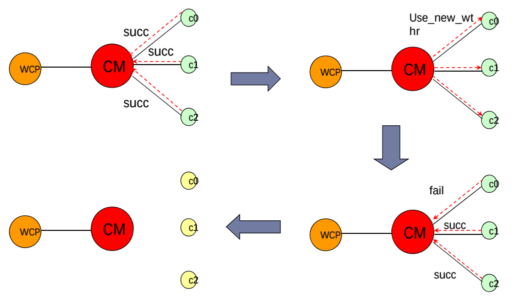
Client – Weather Update



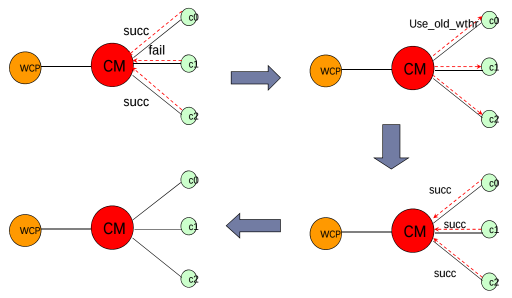
Client Update – Case 1



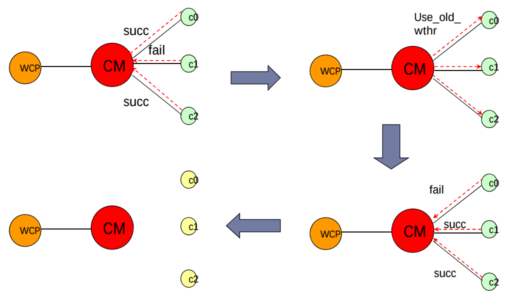
Client Update – Case 2



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

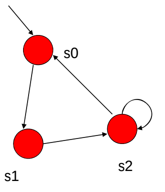
- A disconnected weather-aware client can establish a connection by sending a connecting request to the CM.
- If the ATC's status is idle when the connecting request is received, it will set both its own status and the connecting client's status to preinitializing, and disable the weather control panel 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!

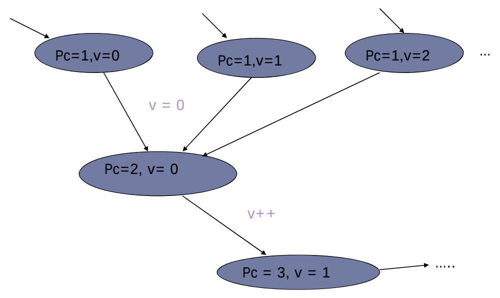
What's in a step?

- 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

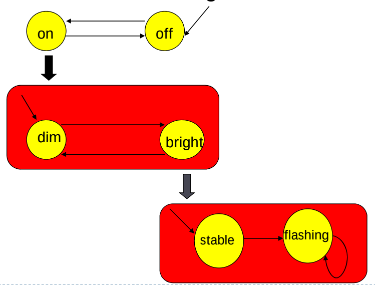
- 1 v = 0;
- 2 v++;
- 3 …
  - What are the states ?
    - (value of pc, value of v)
  - How many initial states are there ?
    - No info, depends on the type of v
- Draw the states and transitions corresponding to this program.

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 × M2 = (S1×S2, I1×I2, →)
  - Where (s1,s2) → (t1, t2) provided
    - s1 ∈ S1, t1 ∈ S1,
    - s2 ∈ S2, t2 ∈ 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, →)
  - S is a finite set of states
  - I ⊆ S is the set of initial states
  - → ⊆ S × S is the transition relation.

<u>Communicating FSM</u>

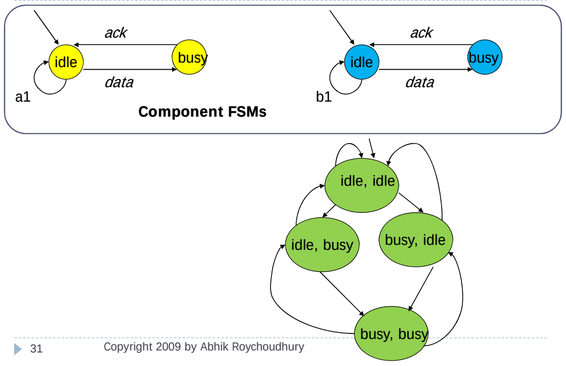
- M = (S, I, Σ, →)
  - S is a finite set of states
  - I ⊆ S is the set of initial states
  - Σ is the set of action names that it takes part in
  - → ⊆ S × Σ × S is the transition relation.

Communication across FSMs via action names.

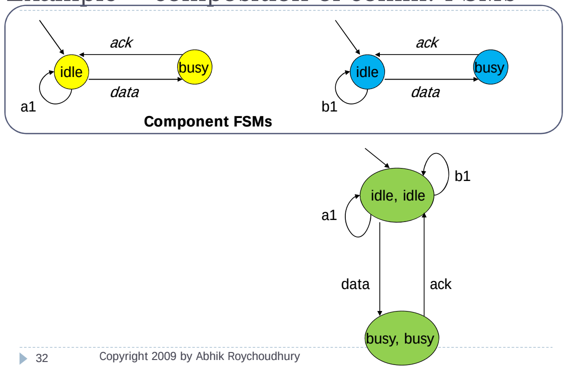
Composition of comm. FSMs

- M1 = (S1, I1, Σ_1 , $\rightarrow_1$) M2 = (S2, I2, Σ_2 , $\rightarrow_2$)
- Define
  - M1 × M2 = (S1×S2, I1×I2, $\Sigma_1 \cup \Sigma_2$, →)
  - And (s1,s2) $\xrightarrow{a}$ (t1,t2) provided
    - s1 ∈ S1, t1 ∈ S1, and
    - s2 ∈ S2, t2 ∈ S2, and
    - If a ∈ $\Sigma_1 \cap \Sigma_2$ we have (s1 $\xrightarrow{a}$ t1) and (s2 $\xrightarrow{a}$ t2)
    - If a ∈ $\Sigma_1 - \Sigma_2$ we have (s1 $\xrightarrow{a}$t1)
    - If a ∈ $\Sigma_2 - \Sigma_1$ we have (s2 $\xrightarrow{a}$ t2)

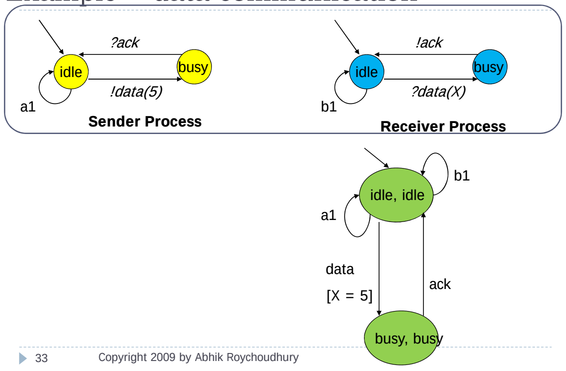
Example - basic composition



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

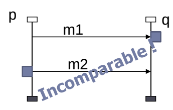
- Global State = (pc0, pc1, turn)
  - pc0 ∈ { l0, l1, l2, l3 }
  - pc1 ∈ { m0, m1, m2, m3 }
  - turn ∈ { 0, 1 }
- Total = 4 * 4 * 2 = 32 possible states
  - Not all of them might be reachable from the initial states.
  - How many are reachable – try it!

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.

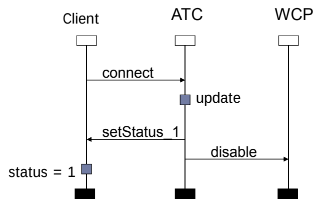
MSC based Models

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



Conventional use of MSCs

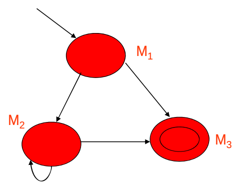
- Describe sample scenarios of system interaction
  - Appears in requirement documents
  - Do not describe "complete" system behavior



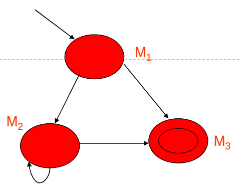
Sample MSC from ATC example

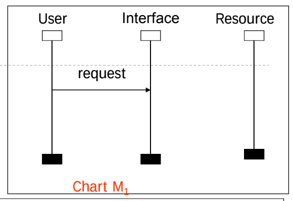
Exercise: Find two incomparable events in this MSC

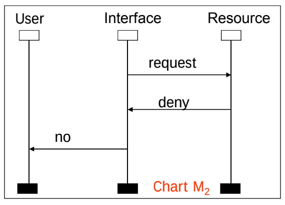
MSC-based design model

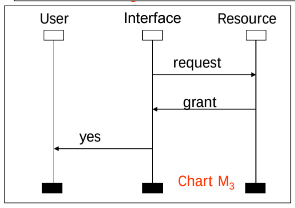


Connect MSCs into a graph – Message Sequence Graph (MSG)
Each node of the graph is a MSC.
Need to define the meaning of concatenation of MSCs

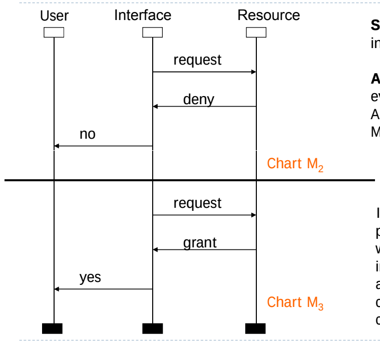








MSC concatenation



Synchronous: All events in M2 ≤ All events in M3

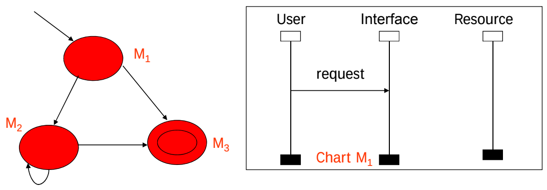
Asynchronous: All events in process p of M2 ≤ All events in process p of M3

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

MSC-based design model?

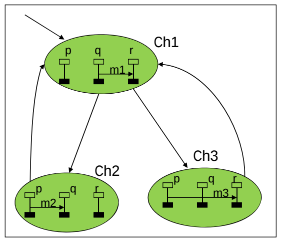
- Complete
  - Complete description of system behavior.
  - MSG achieves this criterion.
- Based on well-established modeling notations.
  - We use UML Sequence Diagrams, which is OK.
- Preferably executable
  - Can simulate the model, and get a feel for how the constructed system will behave!
  - Global 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**.

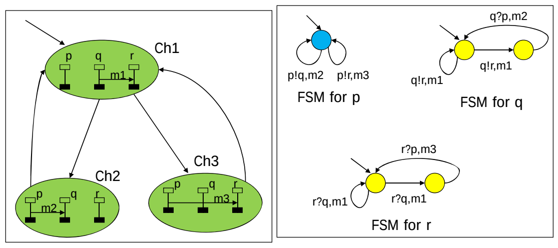
Example MSG



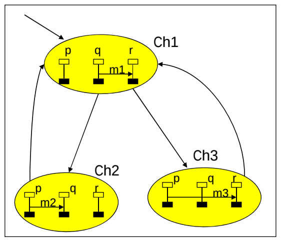
Generates behavior of the form

$(Ch1 \circ (Ch2 + Ch3))^{\omega}$

Per-process FSMs

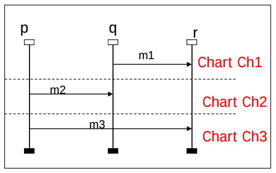


Implied Scenario



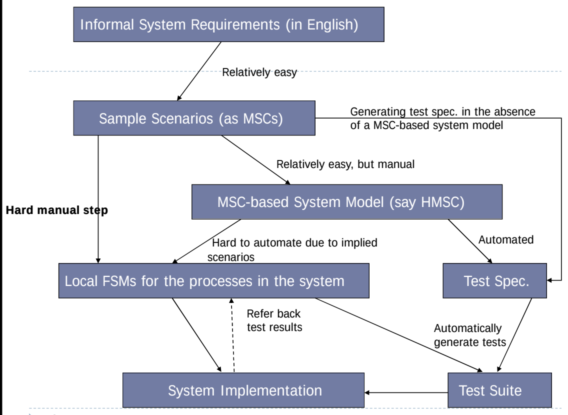
Supposed to generates behavior of the form

$(Ch1 \circ (Ch2 + Ch3))^{\omega}$



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?


Informal System Requirements (in English)
Relatively easy
Sample Scenarios (as MSCs)
Generating test spec. in the absence of a MSC-based system model
Relatively easy, but manual
Hard manual step
MSC-based System Model (say HMSC)
Hard to automate due to implied scenarios
Automated
Local FSMs for the processes in the system
Test Spec.
Refer back test results
Automatically generate tests
System Implementation
Test Suite

CS 4271 Sample Revision Hour (on System Modeling)

Embedded Systems and Software Validation

Review Questions and Revision Exercises
For Lecture on System Modeling

Review Concepts

- Platform model vs. System Model
- System Architecture vs. System Behavior
- Criteria for a design model
- State vs. Transition in the FSM model
- Limitations of the basic FSM Model
- Concept of the Unit Step
  - Basic FSM
  - Hierarchical FSM
  - Concurrent FSM
- Hierarchical vs. Concurrent FSM

Review Concepts

- For a system having m FSMs each having k states, what is the number of states in the global FSM?
- Why is MSC a labeled partial order and not a total order of events?
- Incomparable events in a MSC
- Semantics of Synchronous and Asynchronous Concatenation of MSCs

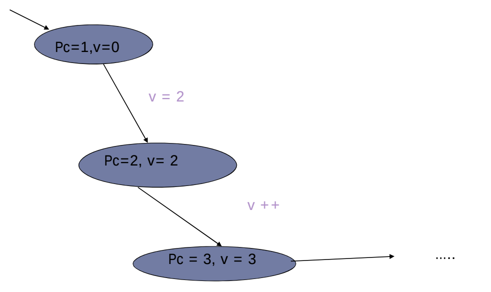
Exercise1

Refer to the code fragment below:

- static int v;
- v = 2;
- v++;
- …..

- What are the program states ?
- How many initial states are there ?
- Draw the states and transitions corresponding to this program.

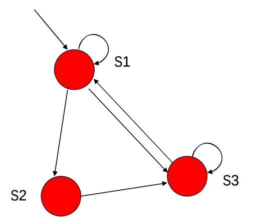
Exercise1: Solution



Exercise2: FSM Modeling: LRT

- A LRT runs between three stations on the North-South Line (S1, S2, S3). The LRT starts from the station S1 and goes to S3 with or without an intermediate stop at S2 depending on whether there is any passenger or not. From S3, the LRT comes back to S1 directly. Design a FSM to portray the LRT behavior.

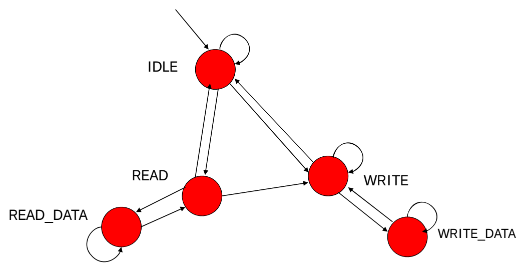
Exercise2: Solution



Exercise3: FSM Modeling: Transaction Controller

- Design a FSM to model the behavior of an abstract transaction controller. The controller is IDLE as long as a READ or a WRITE instruction does not begin on the bus. On encountering a READ instruction, the controller moves to the READ state, sets a transaction variable as ongoing, and waits for the peripheral to provide the READ data. Once READ is finished, it updates the transaction status to finished and moves back to IDLE state. On encountering a WRITE instruction, the controller has a similar behavior, except for the fact that data is written to the peripheral once the peripheral is ready.

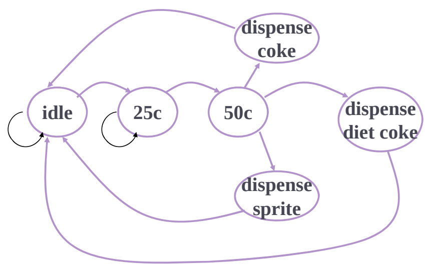
Exercise3: Solution



Exercise4:

- Design a FSM for an automatic vending machine:
  - When turned on, the machine waits for money
  - When a quarter is deposited, the machine waits for another quarter
  - When a second quarter is deposited, the machine waits for a selection
  - When the user presses “COKE,” a coke is dispensed
  - When the user presses either “SPRITE” or “DIET COKE,” a Sprite or a diet Coke is dispensed
  - When the user takes the bottle, the machine waits again

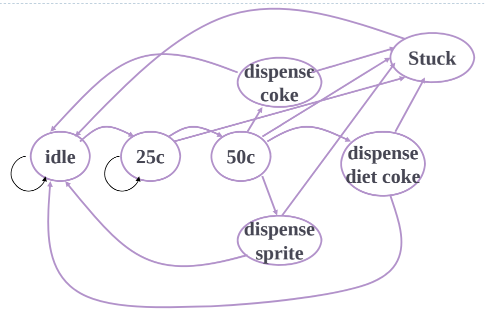
Exercise 4: Solution



Exercise4a:

- Bottles can get stuck in the machine
  - An automatic indicator will notify the system when a bottle is stuck
  - When this occurs, the machine will not accept any money or issue any bottles until the bottle is cleared
  - When the bottle is cleared, the machine will wait for money again
- State machine changes
  - How many new states are required?
  - How many new transitions?

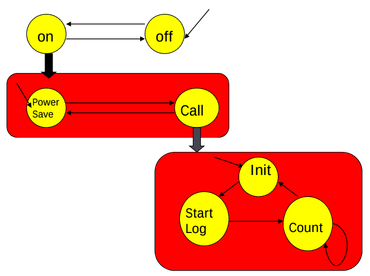
Exercise 4a: Solution



Exercise5: Hierarchical FSM Modeling

- A basic mobile phone (even without the SMS function) has two operating modes: ON and OFF. When in ON mode, the mobile may be in two modes, one when it is connected to a call, and the other in Power-Save mode. When in ON mode and connected to a call, it has two actions: (a) Log the call detail (and initialize the call counter), and (b) Count the number of seconds of the current call. On termination of a call, the mobile phone moves back to the Power-Save mode after a few seconds of inactivity.

Exercise 5: Solution



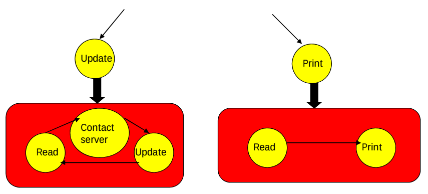
Exercise6: ATM Machine

- An ATM machine can allow an user to carry out one of 3 transactions: Money deposit / Money Withdrawal / Account Balance Enquiry for his savings account. For all the transactions, the user is asked to enter his ATM PIN and on verification of the PIN, the transaction proceeds. By default, the ATM machine is in a IDLE state, and becomes active when a card is entered. In the Money deposit mode, the user is asked to enter the cash or the cheque. In the cash mode, the ATM counts the money and updates the account balance immediately. In the Money withdrawal mode, the ATM may or may not be able to process the transaction depending on whether sufficient cash is present. In all the cases, after a transaction completes, the user is asked whether he wishes to proceed with another transaction. In case he does, he is required to enter his PIN. Otherwise, the card is popped out of the ATM. Design a FSM for the ATM.

Exercise7: Card Processor

- A wireless visiting card processor at a conference booth has two concurrent modes of action: On reading a new visiting card, it performs the following actions simultaneously: (a) Contacts the server to update the contact details read in, and (b) Prints the name and phone number of the card holder on a small print sheet to enter into a raffle. Design a Concurrent FSM view for this operation.

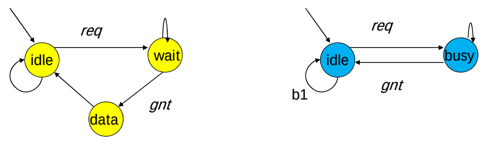
Exercise 7: Solution



Exercise8: A Simple Communication

- Consider a simple master device communicating with an arbiter. To start a transfer, it sends a request signal req to the arbiter. On receiving a request from a master, the arbiter grants by a gnt signal in the next cycle if the transaction buffer is clear, and moves back to its IDLE state; else, it makes the master wait by delaying the send of the gnt signal. The master continues to wait till it receives the gnt. On receiving the gnt, it moves to the DATA state and upon finishing data transfer, it moves back to IDLE state. Present a compositional view of the concurrent FSMs for the arbiter and the master devices.

Solution



Exercise9: A Bus protocol

- We consider a simple bus protocol that supports master device, slave devices and an arbiter. The bus has 64-bit multiplexed address and data lines. During a transfer, address and data are multiplexed over these lines, with address and data appearing alternately in address and data cycles respectively. A master device is IDLE when it does not intend to perform a transfer. When the master intends to perform a transfer, it raises its request signal req and waits for the arbiter to grant using the gnt signal. On getting a gnt, the master floats the address on the bus and waits for the next cycle onwards for the slave to give the rdy signal. We refer to this phase as the ADDRESS cycle. On receiving the rdy signal from the slave, the master enters the DATA cycle and does the following:
  - In case of a write transfer, it floats the data on the bus
  - In case of a read transfer, it expects the slave to produce data on the bus

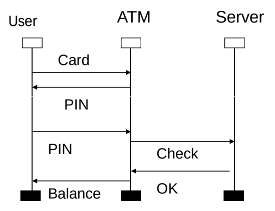
Exercise9: A Simple Bus protocol

- After each DATA cycle, the master may start another DATA cycle by floating the address on the bus and moving to the ADDRESS cycle. The master may also return to the IDLE state by lowering the req line which signals to the arbiter that the master wishes to terminate. Design a FSM for the Master interface as:
- (a) A single communicating FSM
- (b) A set of two concurrent communicating FSMs, one for the normal operation and the second to model the DATA ADDRESS cycles.

Exercise10: ATM access

- Consider the scenario of an user accessing an ATM machine. Describe a successful access scenario (card entry / PIN request / PIN entry / PIN authentication / Account Balance request / Balance detail) using a set of messages on a MSC. Can you identify any incomparable events in the MSC?

MSC



Exercise11: Basic MSC Modeling

Consider a simple master device communicating with an arbiter to get access to a shared resource. To start a transfer, it sends a request signal req to the arbiter. On receiving a request from a master, the arbiter grants by a gnt signal if the transaction slave is ready, else, it makes the master wait by sending the deny signal. The master continues to request till it receives the gnt. Describe the above protocol using a set of MSCs. Also construct the MSG.

Exercise12: MSC Modeling

- Design a MSC showing the steps involved when a user wishes to check his SOC Email.

Exercise13: MSC Modeling

- Design a MSC showing the steps involved when a user wishes to check his SOC Email.

CS 4271 Sample Lab Assignment (on System Modeling)

Programming assignment 1 (Lab1) for CS 4271 on System Modeling

Learning objectives:

- **Mapping of informal system requirements to formal models.**
- **Use of model simulation to detect errors, and discussion of possible fixes to subtle corner cases detected.**
- **Most importantly, a feel for how subtle errors can creep into seemingly correct system descriptions owing to unforeseen interactions among concurrently executing system components.**

Description of the control system used in the assignment:

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

Client initialization

1. A disconnected weather-aware client can establish a connection by sending a connecting request to the CM.

2. If the CM'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 (CM's status is not idle), the CM will send a message to the client to refuse the connection, and the client remains disconnected.

3. When the CM is pre-initializing, it will send a message to instruct the newly connected client to get the new weather information, and then set both its own status and the client's status to initializing.

4. If the client reports success for getting the new weather, the CM will send another message to inform the client to use the weather information, and then set both its own status and the client's status to post-initializing. Otherwise, if getting new weather fails, the CM will disconnect the client and set its own status back to idle.

5. If the client reports success for using the new weather, this initialization process is completed. the CM will set both its own status and the client's status to idle, and re-enable the WCP so that manual weather update is allowed again. Otherwise, if using new weather fails, the CM will disconnect the client, re-enable the WCP, and set its own status back to idle.

Weather update

1. User can manually update new weather information only when the WCP is enabled. By clicking the update button on the WCP, a update message is sent to the CM.

2. When the CM is idle and receives update request from the WCP, it will set its own status and all the connected weather-aware clients' status to pre-updating, and disable the WCP from any further updating requests before the completion of current update.

3. When CM's status is pre-updating, it will send messages to instruct all connected clients to get the new weather information, and then set its own status and the clients' status to updating.

4. If all the clients report success for getting the new weather, the CM will send messages to inform the clients to use the new weather information, and then set its own status and the clients’ status to post-updating. Otherwise, if any of the connected clients reports failure for getting the new weather, the CM will send messages to all clients to use their old weather information, and then set its own status and the clients’ status to post-reverting.

5. When CM’s status is post-updating, if all the clients report success for using the new weather, the updating is completed. The CM will set its own status and the clients’ status to idle, and re-enable the WCP. Otherwise, if any of the connected clients reports failure for using the new weather, the CM will disconnect all connected clients, re-enable the WCP, and set its own status back to idle.

6. When CM’s status is post-reverting, if all the clients report success for using the old weather, the reverting is completed. The CM will set its own status and the clients’ status to idle, and re-enable the WCP.
Otherwise, if any of the connected clients reports failure for using the old weather, the CM will disconnected all connected clients, re-enable the WCP, and set its own status back to idle.

Questions

1. (6 marks) Use Rhapsody in C to model the weather-update controller system according to the specification given above. Your model should contain the class diagram (consist of WCP) and statechart for all classes. Your class diagram should contain the three classes-the WCP, the CM, and clients, as well as corresponding multiplicities and associations. Please state clearly in the report any assumptions you make during modeling of the system.

2. (2 marks) Suppose a system has three weather-aware clients - c1, c2 and c3. Generate an animated sequence diagram named ”q2Sequence” for the following scenario.
(a) c1 connects to the CM successfully.
(b) c2 connects to the CM successfully.
(c) After that, a weather update request is sent to the WCP.
(d) Both c1 and c2 report success in getting new weather information.
(e) At meantime, c3 tries to connect to CM. The connection request fails
due to the weather update is in progress.
(f) c1 successful uses the new weather information, but c2 fails, which
causes both of them disconnected from the CM.

3. (2 marks) For the above system, produce a sequence of events that leads the system into a state such that:
• c1 is connected to the system; c2 and c3 are disconnected.
• the CM is in the idle state.
• and the WCP is disabled from manual update.
Generate an animated sequence diagram named ”q3Sequence” corresponding to this scenario. Note that in this state, even when the CM is idle, no weather update can be made, which is obviously not desirable. Find the bug in the specification that causes such a situation and discuss how to fix it in your report.

CS 4271 Sample Midterm Examination

CS 4271 Critical Systems and their Verification

AY 2010-11 Midterm Examination, Dated 2nd March 2011.

Time: 1 hr 30 minutes

Instructions to Candidates

1. **This is an open-book examination.**
2. **Answer all questions.**
3. **Answer to any question should be written in the space allotted below the question. It is alright to use the blank pages on the other side.**
4. **It is perfectly alright to use pencils, as long as I can read what you wrote.**
5. **Please write only your matriculation number below.**

MATRICULATION NUMBER:

Question A: 4 marks

Question B: 4 marks

Question C: 6 marks

Question D: 4 marks

Question E: 2 marks

Question F: 5 marks

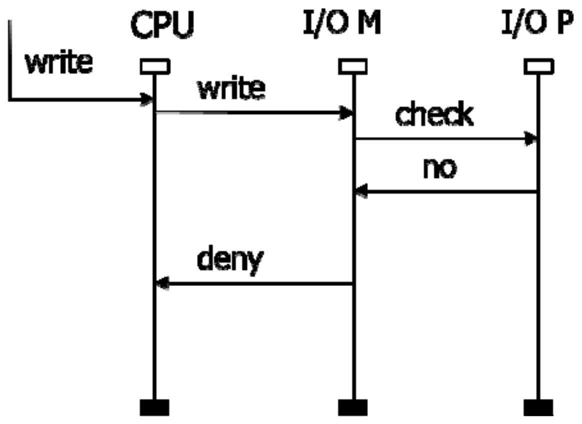
TOTAL: 25 marks

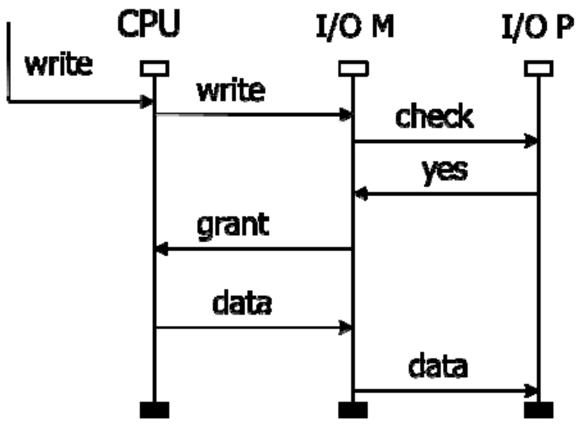
Question A [4 marks]

Consider a situation that occurs in a programmed IO environment. The CPU issues a WRITE command to the I/O module which then checks if the peripheral is free. The request may either be accepted or rejected depending on whether the peripheral is free or not. If the request is accepted, the CPU sends the data to the I/O Module to write to the peripheral. If the peripheral is not free, the I/O module makes the CPU wait till the peripheral is free.

Show sample interactions of the above system using Sequence Diagrams.

Answer:

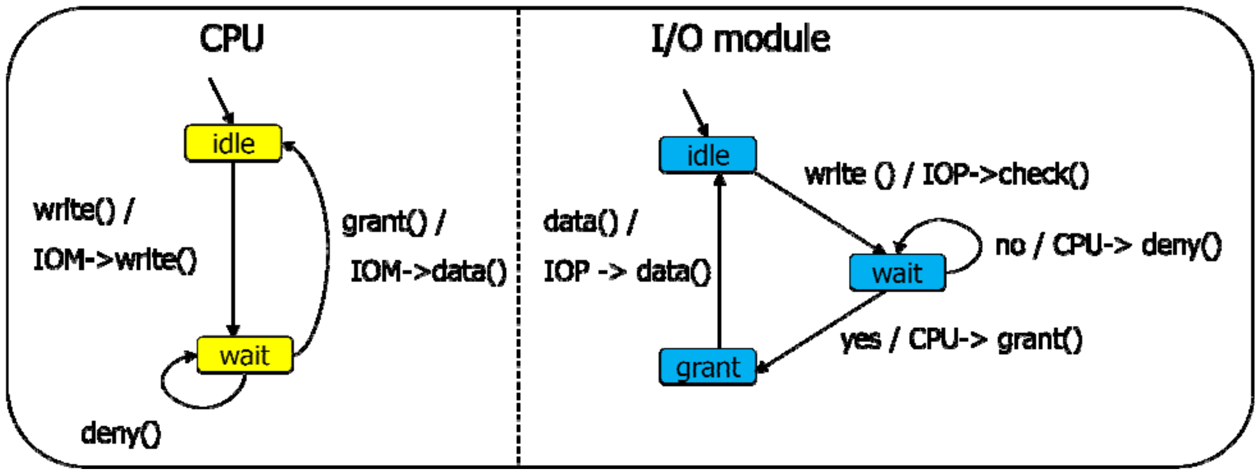




Question B. [4 marks]

Present an intra-component view of CPU and I/O module as a UML State Diagram.

Answer:



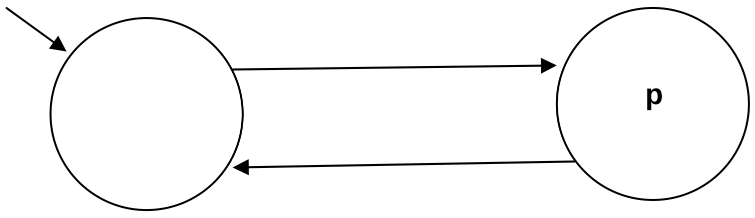
Question C. [6 marks]

Show that the following pairs of temporal logic formula are not equivalent. You should construct an example system model which satisfies one of them but not the other. You may assume that p,q are atomic propositions.

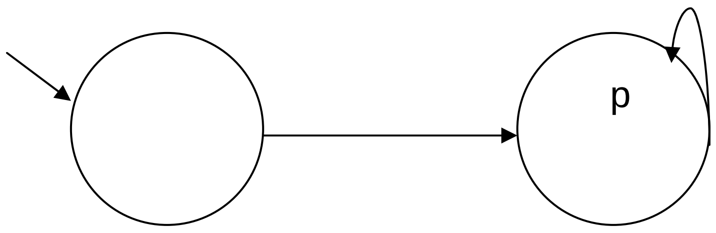
- i) FG p and GF p
- ii) (true U p) and (false U p)
- Iii) (p U q) and (q R p)

Answer:

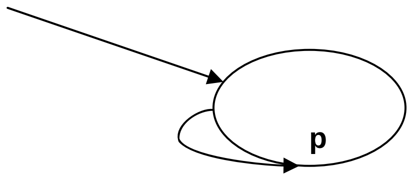
(i) Satisfies GFp but does not satisfy FGp



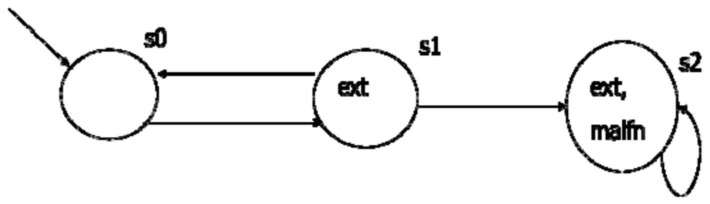
(ii) (true U p) is the LTL formula Fp and (false U p) is equivalent to the formula p. The following state machine satisfies (true U p), but does not satisfy (false U p).



(iii) qRp is true even if q never holds. But pUq cannot be true if q never holds. So, the following system model satisfies qRp but does not satisfy pUq, since it is possible to loop at the initial state and thereby q never holds.



Question D [4 marks]



Consider the above state machine which represents a simplified model of a spring controller. There are two atomic propositions *ext* and *malfn*. The proposition *ext* is true whenever the spring is extended, and the proposition *malfn* is true when the spring is malfunctioning. The true/false valuations of the propositions in the individual states of the state machine are shown.

i) Formally state the property "Whenever a spring is extended, it eventually malfunctions".

ii) Is the property true for the spring model shown above. Explain your answer.

Answer:

i) G(ext $\Rightarrow$ F malfn)

ii) The traces of the spring model are $(s0\ s1)^{\omega}$ and $(s0\ s1)^{*}\ (s2)^{\omega}$

Among these, $(s0\ s1)^{\omega}$ does not satisfy the property in question.

Hence the spring model does not satisfy the property.

Question E [2 marks]

Consider a traffic light controller which initially shows green light. It senses traffic data every second. Thus, starting from time=0, the traffic is sensed at time = 1 sec, 2 sec, and so on. If there is no traffic movement, the light turns from green to yellow. If there is traffic movement, the light stays green, but it can stay green for a maximum of 3 seconds at a stretch. The light always stays yellow for exactly one second after which it turns red. Once the light is red, again traffic is sensed every second. If there is traffic, then the light becomes green after staying red for a minimum of 2 seconds. If there is no traffic, the light eventually becomes green, after staying red for 3 seconds.

Suppose you were trying to model the above controller as a state machine. Is it possible for you to model the exact requirements as stated above as a state machine? *What aspects of the requirements can you model and which aspects you cannot? Explain your answer.*

Answer:

The exact timing (say 1 second) corresponding to each transition cannot be modeled as a state machine, since the state machine model does not capture such information.

Question F [5 marks]

Consider the following Promela specification of a simple producer and consumer. Assume that both of them access a buffer which serves as a shared data structure.

```
toktype = {P, C};
toktype turn = P;

active proctype producer()
do
:: (turn == P) -> /*produce 1 item*/; turn = C;
od
}

active proctype consumer() {
do
    :: (turn == C) -> /* consume 1 item */; turn = P;
od
}
```

If there is one producer process and one consumer process, will an interleaved execution violate mutual exclusion of access to the buffer? To answer this question, *construct the global state transition system after assigning labels to the control locations of the processes.*

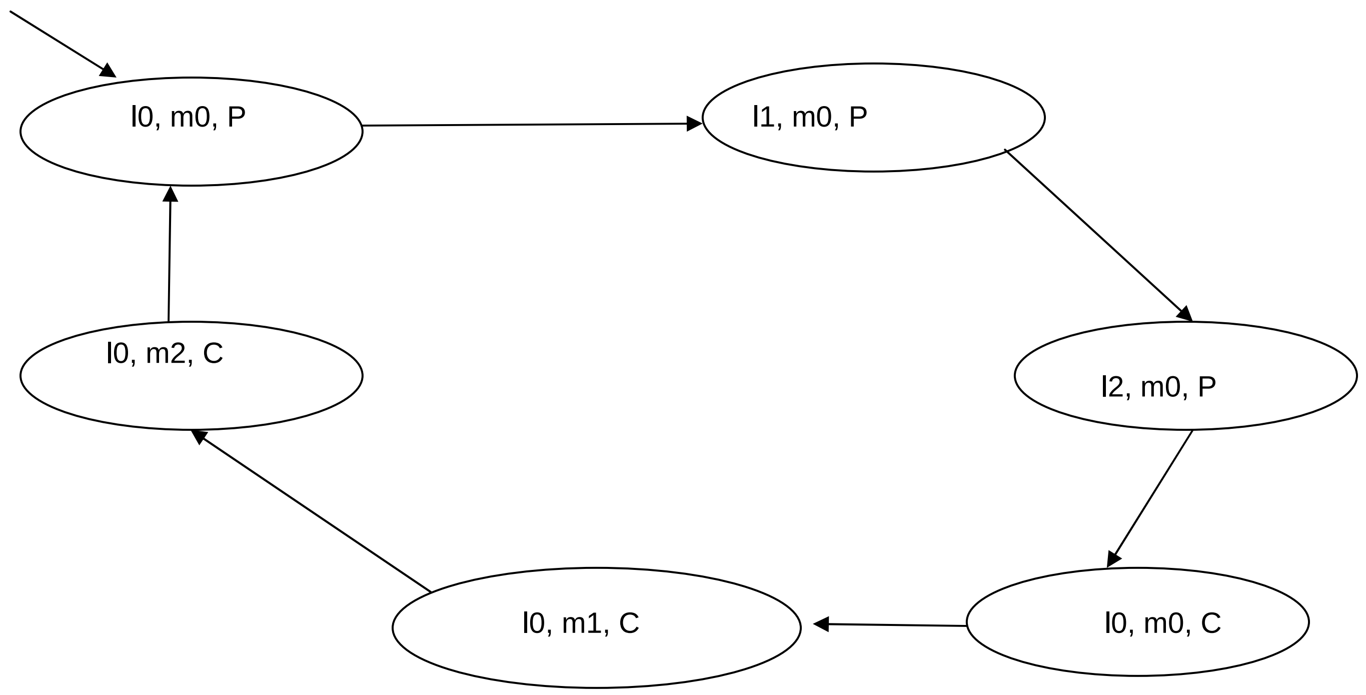
Answer:

The critical section is not shown in the code, but we do not need to assign control labels inside the critical section since it will not be subjected to interleaving. It suffices to consider the following labels {l0,l1,l2,m0,m1,m2}.

```
toktype = {P, C};
toktype turn = P;

active proctype producer()
l0 do
:: (turn == P) -> l1 /*produce 1 item*/ ; l2 turn = C;
od
}

active proctype consumer() {
m0 do
    :: (turn == C) -> m1 /* consume 1 item */; m2 turn = C ;
od
}
```



There is also a self-loop in each state which is not shown.

Mutual exclusion can be seen to be preserved by visual inspection since no reachable state is of the form (l1,m1,_)

-END OF PAPER-

CS 4271 Sample Final Examination

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 2 AY2010/2011

CS4271 – CRITICAL SYSTEMS AND THEIR VERIFICATION

Apr/May 2011 **Time Allowed: 2 Hours**

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **FOUR (4)** questions and comprises **NINE (9)** printed pages, including this page.
2. Answer **ALL** questions within the space in this booklet
3. This is an Open Book examination.
4. Please write your Matriculation Number below.

MATRICULATION NO: ___________________________________

This portion is for examiner's use only

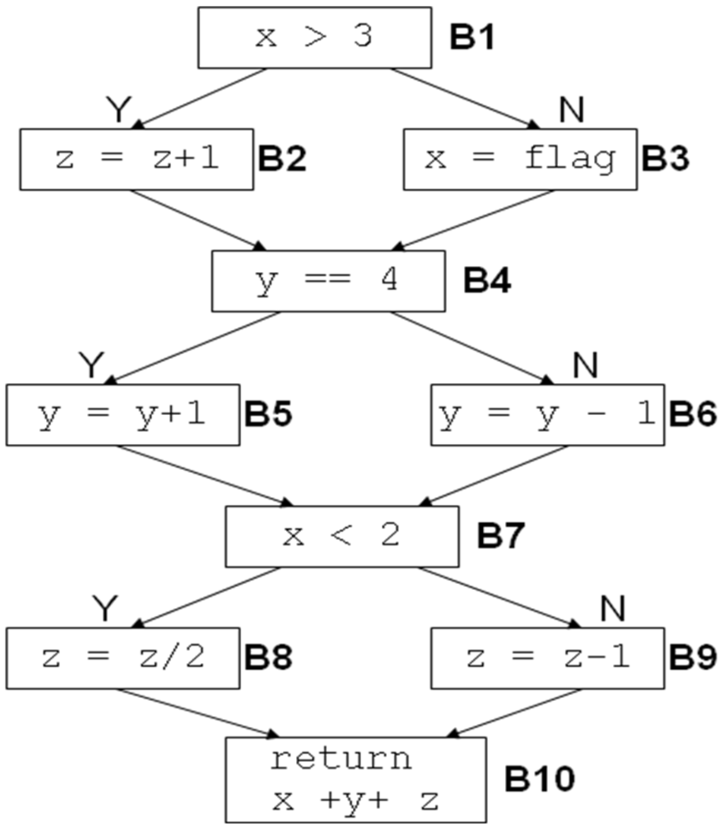
Question	Marks	Remarks
Q1	15	
Q2	15	
Q3	9	
Q4	11	
Total	50	

Question 1 [6 + 4 + 5 = 15 marks]

A. Consider the following program. Draw its control flow graph. How many paths are there? How many of these paths are feasible and how many are infeasible? Give detailed justifications for your answer.
B. Use the timing schema based Worst-case Execution Time (WCET) calculation method which does not use any infeasible path information. Assume that each basic statement takes 1 time unit.
C. Use the Integer Linear Programming based WCET calculation method which takes into account infeasible path information. Encode as many infeasible path constraints as possible so that the estimated WCET is as tight as possible. How much does the WCET estimate change due to infeasible path information?

```
int P1( int flag,x,y,z ) {
       if( x > 3 ) z = z + 1; else x = flag;
       if( y == 4 ) y = y + 1; else y= y - 1;
       if( x < 2 ) z = z / 2; else z = z - 1;
       return x+y+z;
}
```

Answer: A. The control flow graph is follows. There are eight paths.



Infeasible paths are introduced due to occurrence of $[x>3]_{yes}$ and $[x<2]_{yes}$.
This introduces two infeasible paths, the remaining six paths are feasible.

```
int P1( int flag,x,y,z ) {
       if( x > 3 ) z = z + 1; else x = flag;
       if( y == 4 ) y = y + 1; else y= y - 1;
       if( x < 2 ) z = z / 2; else z = z - 1;
       return x+y+z;
}
```

B. Time(pgm) = Time(if1) + Time(if2) + Time(if3) + Time(return)

Time(if1) = Time(x>3) + max(Time(z++), Time(x=flag)) = 1 + 1 = 2 time units

Similarly for Time(if2), Time(if3)

So, Time(pgm) = 2 + 2 + 2 +1 = 7 time units.

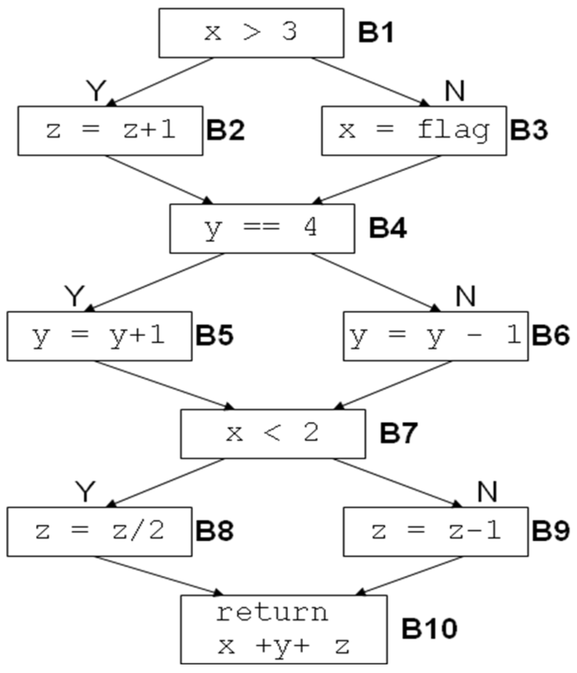
C. The simplified ILP equations are as follows

Maximize Time = N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8 + N9 + N10

1 = N1 = N2 + N3 = N4 = N5 + N6 = N7 = N8 + N9 = N10 = 1

The only infeasible path information can be encoded as N2 + N8 <= 1

Irrespective of the constraint we get Time = 7 units.



Question 2 [6 + 3 + 6 = 15 marks]

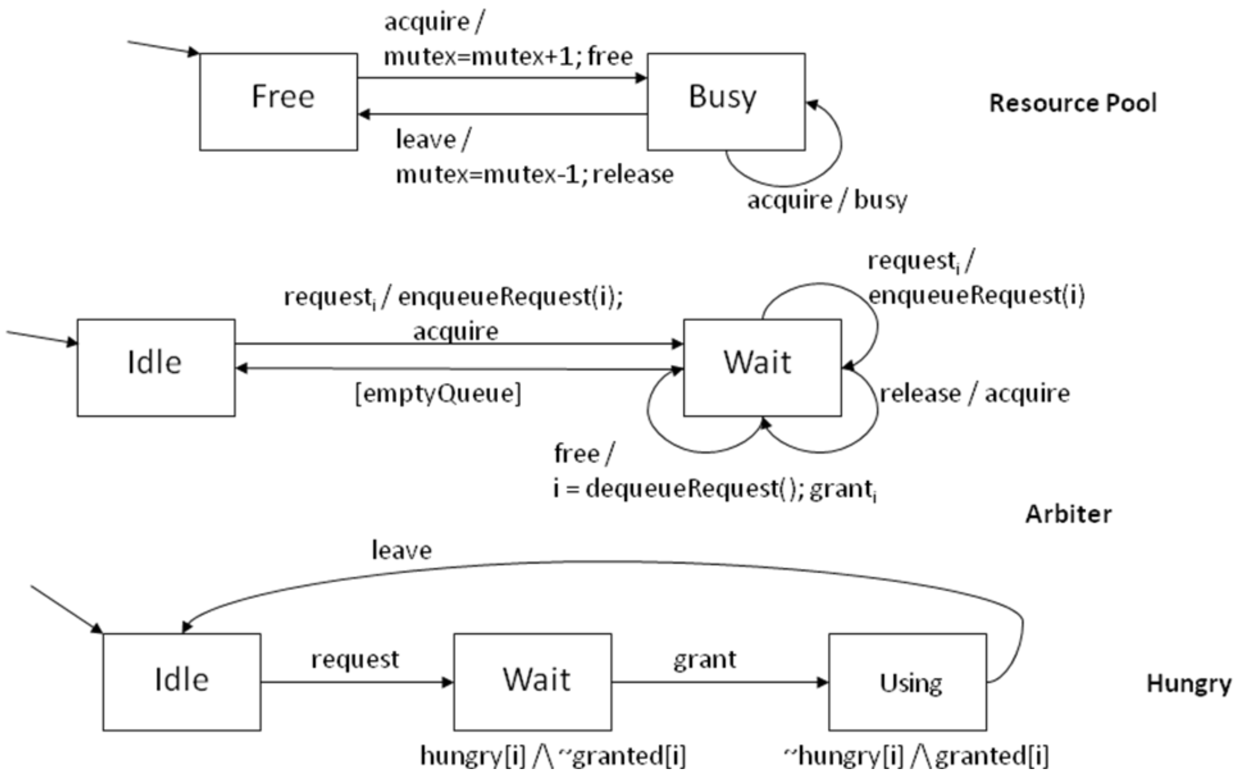
Consider an arbitration protocol where multiple hungry processes request for access to one of a pool of resources via an arbiter. Thus, the hungry processes request the arbiter for access to the resource pool. Once they get access they access a specific resource in the pool. Modeling such a system, we should have three classes of objects – *Hungry*, *Resource* and *Arbiter*.

A. Elaborate the design, by giving the UML State Diagrams of each class. Your State Diagram design must satisfy the following criteria --- (a) at most one hungry process must access the resource pool at any time, (b) if there are one or more hungry processes requesting access to the resource pool, the arbiter should have allowed at least one of them, (c) any hungry process requesting access to resource pool should eventually get access.

B. Clearly state what parts of your State Diagram are ensuring each of these three properties. If you make any assumptions for ensuring these properties, you should clearly state all your assumptions.

C. State the three informal properties in part A formally in Linear Time Temporal Logic (LTL). You should clearly state the meaning of any atomic proposition you use in the LTL properties, and the true/false valuations of these atomic propositions in the states of your State Diagram.

Answer:


acquire / mutex=mutex+1; free
Free
Busy
leave / mutex=mutex-1; release
acquire / busy
Resource Pool
request$_i$ / enqueueRequest(i); acquire
Idle
Wait
[emptyQueue]
request$_i$ / enqueueRequest(i)
release / acquire
free / i = dequeueRequest(); grant$_i$
Arbiter
leave
Idle
request
Wait
grant
Using
hungry[i] /\ ~granted[i]
~hungry[i] /\ granted[i]
Hungry

B. Assumption: Each transition in the state diagram is atomic.

Mutual exclusion is maintained by “mutex” variable inside Resource class. Since the busy signal is returned as long as the Resource is occupied by a hungry process, mutex value never goes more than 1.

Fairness is maintained by a FIFO queue. It stores all the incoming requests to access the resource pool in FIFO order. Once the resource pool becomes free, the front element of the queue is granted access of the resource pool. Therefore, the system is starvation free.

C.

- **At most one hungry process must access the resource pool:**

G(mutex <= 1),

mutex is the number of processes accessing the resource pool

- **If there are one or more hungry processes requesting access to the resource pool, the arbiter should have allowed at least one of them**

G((hungry[1] \/ .... \/ hungry[n])->((hungry[1] /\ Fgranted[1]) \/.....\/ (hungry[n] /\ Fgranted[n])))

- **Any hungry process requesting access to resource pool should eventually get access**

G (hungry[1]->Fgranted[1] /\ hungry[2]->Fgranted[2] /\ ..... /\ hungry[n]->Fgranted[n])

hungry[i] = true if and only if process i has requested access for resource pool.
granted[i] = true if and only if process i has granted access of the resource pool.

Question 3 [3 + (3 + 3) = 9 marks]

A. Suppose several periodic processes are running on a processor (with a cache) which employs a pre-emptive scheduling policy. Thus, the processes share the processor cache. Now, when a process is pre-empted by another process, the pre-empting process pollutes the processor cache which can affect the execution time of the pre-empted process once it resumes execution. In class, when we discussed pre-emptive scheduling policies like RMS, EDF we always assumed a zero context switch overhead. However, in reality, it is not so, and the cache is one reason contributing to a non-zero context switch overhead. Devise an analysis method to estimate the number of cache misses which may be introduced by a single pre-emption. You may want to style your analysis on the cache modeling for WCET analysis we discussed in class.

Answer:

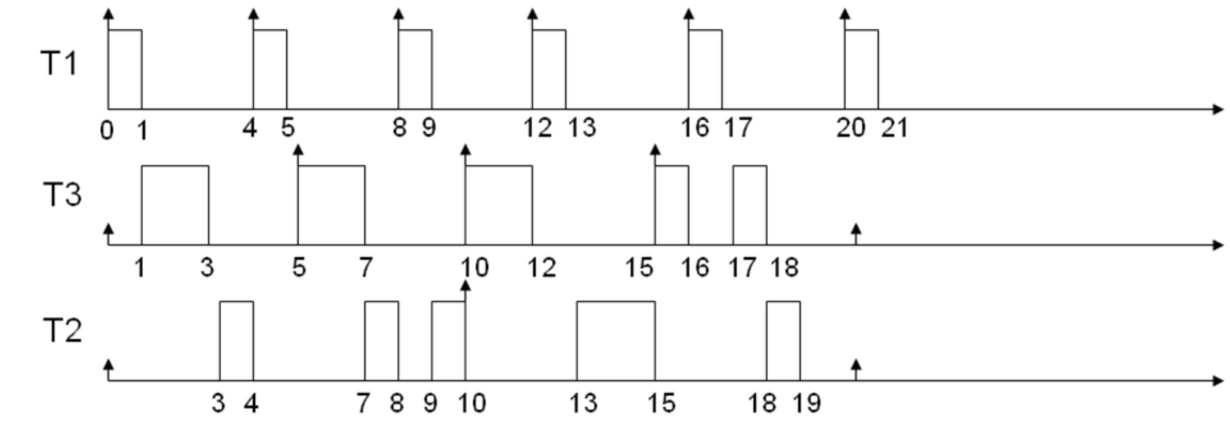
We need to find out the set of cache blocks used after the preemption point. These set of cache blocks, if removed by the preempting task, may create additional delay to the preempted task (due to the cache block reload penalty). A backward fixed point analysis can be carried out to find the set of used cache blocks after the preemption point. These set of cache blocks imposes an upper bound on the number of cache misses after a single preemption.

B. Is the following task set schedulable under the RMS and EDF scheduling policies respectively?

T1 = (1, 4, 4)   T2 = (3, 10, 10)   T3 = (2, 5, 5)

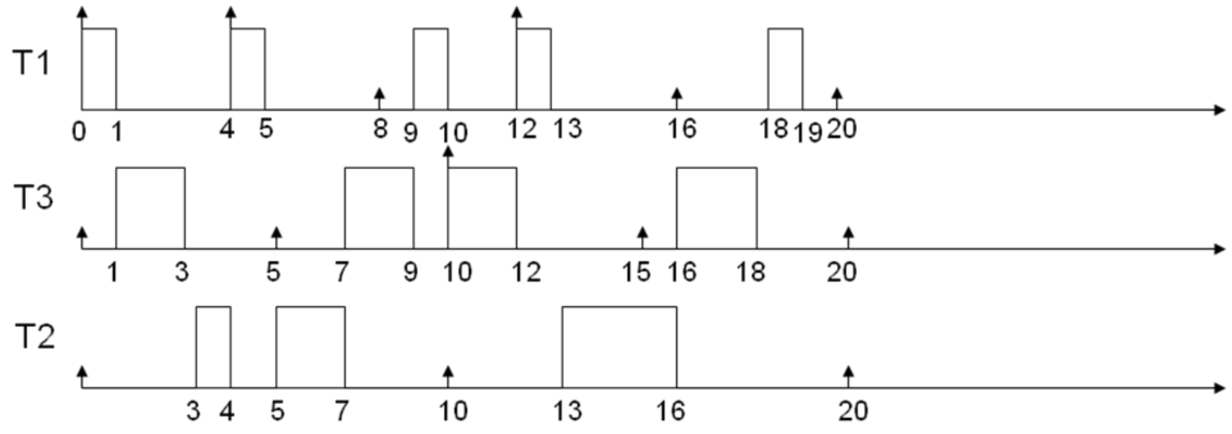
Each task is as a triple (WCET, period, deadline) where deadline = period.

RMS Scheduling:



All tasks meet their deadlines. Therefore, all tasks are schedulable under RMS.

EDF Scheduling:



All tasks meet their deadline. Therefore, all tasks are schedulable in EDF also.

Question 4 [(2 + 4) + 5 = 11 marks]

A. In class we discussed the usage of symbolic execution for test generation. As discussed, we can use symbolic execution to compute a *path condition* – a logical formula which is true for all inputs following a given path. Consider the following program which computes the greatest common divisor of two numbers x and y.

```
scanf("%d", &x); scanf("%d", &y);
while (x != y){ if (x >y) { x = x-y;} else {y = y - x;} }
printf("%d\n", x);
```

What is the path traced by the input (x == 8, y == 6)?
Compute the ***path condition*** of this path.

Answer: The path traced by the input (x == 8, y == 6) is as follows

```
Scanf x; scanf y;
If (x > y)   // yes
    x = x – y    // x is now 8-6 = 2, y is 6
if  (x > y)   // no
    y = y – x   // y is now 6-2=4, x is 2
if  (x > y)   // no
    y = y – x   // y is now  4-2 = 2, x is 2
Exit loop  [x !=y] is false
```

The path condition is

x > y /\ x – y <= y /\ x – y <= y- (x – y) /\ (x – y) == y – (x – y) – (x – y)

x > y /\ x <= 2y /\ 2x <= 3y /\ 3x == 4y

B. Consider a function *triangle* which takes three integers x,y,z which are the lengths of triangle sides and then calculates whether the triangle is equilateral (that is x == y == z), isosceles (exactly two sides are equal), or scalene (otherwise).

Suppose your professor has generated the test-suite for such a function by some means. How would you evaluate whether the professor's test-suite is good enough? Well you can start by asking some questions, to the professor like the following.

- *Do you have a test case for an equilateral triangle?*
- *Do you have a test case for an isosceles triangle?*
- *Do you have at least three test cases for isosceles triangles, where all permutations are considered? (e.g. (x,x,y), (x,y,x), (y,x,x))*
- *Do you have a test case with one side zero?*

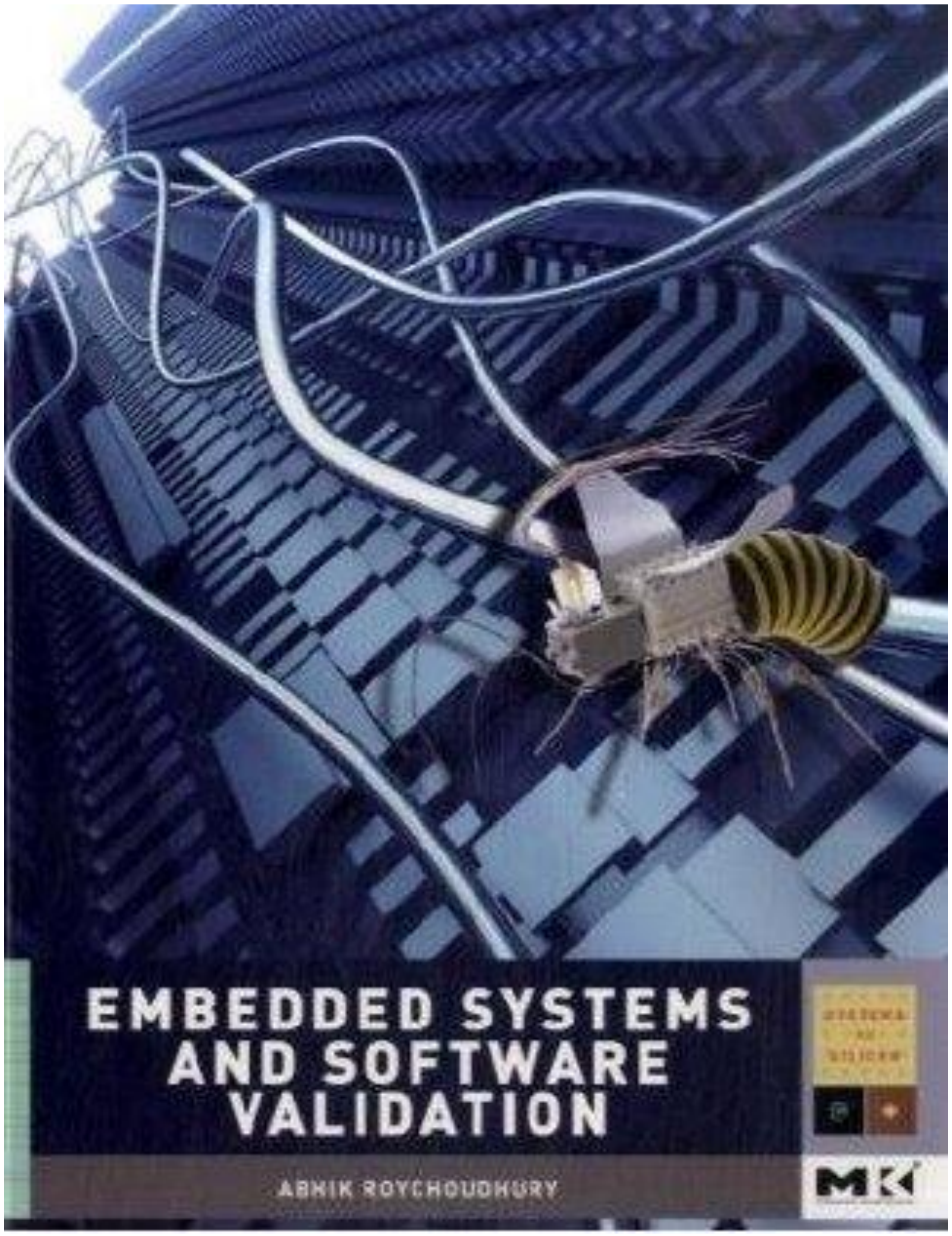
Now, I could go on and on – but I need your help here ☺ Tell some more questions that you should ask to evaluate whether the test-suite generated by the professor is good enough. The more questions you can ask which point to possible corner cases – the more marks you will get. **If you ask ten more questions (over and above the four questions given above), you will get full marks. Good Luck ☺**

Answer:

There can be many other questions such as the following ten questions.

- Do you have the test case (0,0,0)?
- Do you have a test case with negative values?
- Do you have a test case where the sum of two sides equals the third one?
- Do you have at least three test cases for such non-triangles, where all permutations of sides are considered?
- Do you have a test case where the sum of the two smaller inputs is greater than the third one?
- Do you have at least three such test cases, considering all permutations?
- Do you have test cases with very large integers (exceeding MAXINT) ?
- Do you have a test case with non-integer values but numbers?
- Do you have a test case with non-numbers e.g. strings, characters …
- Do you have a test case where 2 or 4 inputs are provided?

CS 4271 Sample Pages of Text-book authored by me



Contents

Preface

This book attempts cover the issues in validation of embedded software and systems. There are lot of books on "embedded software and systems" as a web search with the appropriate search terms will reveal. So, **why this book?**

There are several ways to answer the question. The first, most direct, answer is that the current books mostly deal with the programming and/or co-design of embedded systems. Validation is often discussed almost as an after-thought. In this book, we treat validation as a first class citizen in the design process, weaving it into the design process itself.

The focus of our book is on validation, but from a embedded software and systems perspective. The methods we have covered (testing/model-checking) can also be covered from a completely general perspective, focusing only on the techniques, rather than how they fit into the system design process. But we have not done so. Even though the focus of the book is on validation methods, we clearly show how it fits into system design. As an example, we present and discuss the model checking method twice in two different ways — once at the level of system model (Chapter 2) and again at the level of system implementation (Chapter 5).

Finally, being rooted in embedded software and systems — the focus of our book is not restricted to functionality validation. We have covered at least two other aspects — debugging of performance and communication behavior. As a result, this book contains analysis methods which are rarely found in a single book — testing (informal validation), model checking (formal validation), worst-case execution time analysis (static analysis for program performance), schedulability analysis (system level performance analysis) and so on — all blended under one cover, with the goal of reliable embedded system design.

As for the chapters of the book, Chapter 1 gives a general introduction to the issues in embedded system validation. Differences between functionality and performance validation are discussed at a general level.

Chapter 2 discusses model-level validation. It starts with a generic discussions on system structure and behavior and zooms into behavioral modeling notations such as Finite-state machines (FSMs) and Message Sequence Charts (MSCs). Simulation, testing and formal verification of these models are discussed. We discuss model-based testing, where test cases generated from the model are tried out on the system implementation. We also discuss property verification, and the well-known model checking method. The chapter wraps with a nice hands-on discussion on practical validation tools such as SPIN and SMV. Thus, this chapter corresponds to *model-level debugging.*

Chapter 3 discusses the issues in resolving communication incompatibilities between embedded system components. We discuss different strategies for resolving such incompatibilities, such as endowing the components with appropriate interfaces, and/or constructing a centralized communication protocol converter. Thus, this chapter corresponds to *communication debugging.*

Chapter 4 discusses system level performance validation. We start with software timing analysis, in particular Worst-case Execution Time (WCET) analysis. This is followed by the estimation of time spent due to different interferences in a program execution — from the external environment, or due to other executing programs on same/different processing elements. Suitable analysis methods to estimate the time due to such interferences are discussed. We then discuss mechanisms to combat execution time unpredictability via system level support. In particular, we discuss compiler controlled memories or scratchpad memories. The chapter concludes with a discussion on time predictability issues in emerging applications. Thus, this chapter corresponds to *performance debugging.*

Chapter 5 discusses functionality debugging of embedded software. We discuss both formal and informal approaches, with almost equal emphasis on testing and formal verification. The first half of the chapter involves validation methods built on testing or dynamic analysis. The second half of the chapter concentrates on formal verification, in particular, software model checking. The chapter concludes with a discussion on combining formal verification with testing. Thus, this chapter corresponds to *software debugging.*

Apart from some debugging/validation methods being common to Chapters 2 and 5, the readers may try to read the chapters independently. A senior undergraduate or graduate course on this topic may however read the chapters in sequence, that is, chapters 2, 3, 4, 5.