MTCPS: Cheat sheesh

Synchronous Model **3**
- Parallel Composition 3
  - Inputs 3
  - States 3
  - Initial states 3
  - Requirements for Parallel composition 4
  - Component compatibility 4
  - Example 4
- Textual description 5
  - From text to model 5
  - From model to textual description 5
- Execution/transitions format example 6
- Describe a component as an extended-state machine 6
- Mealy machine 6

Safety Requirements **7**
- Reachable states 7
  - Example 8
- Inductive Invariant 8
  - Example 11
- Image Computation 12
- Safety Monitor 12
  - Example 12
- Properties exercise example 12

Asynchronous Model **14**
- Create Asynchronous process example 15
- Asynchronous process composition 16
  - Example 17
- Mutual exclusion protocol 18
- Fairness assumptions 18
  - Weak vs Strong Fairness 18
  - Exercise example 20

Transition systems **22**
- Timed model/Automata 22
  - Fisher’s Protocol for Mutual Exclusion 23
  - Timing Analysis example 24
- Symbolic Transition System 24
  - Describe the transitions system example 1 25
  - Computing image example 1 25
  - Describe the transitions system example 2 26

Computing image example 2 26

Real-Time Scheduling **27**
The Cyclic Executive Approach 28
Task Scheduling Architectures 28
Fixed Priority Scheduling (FPS) 29
How to assign priorities for FPS 29
Earliest Deadline First (EDF) 30

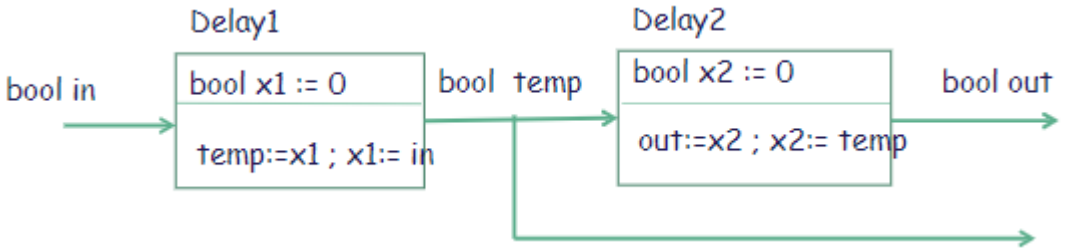
Uppaal **30**
Reachable states 30
Verifier 31
Cheat sheet 31

MatLab **31**
(A, B, C, D) Notation 31
Cheat sheet 32
Functions list/explanations 32
place() 32
plot() 32
ss() 32
step() 32
ode45() 32
eig() 33
inv() 33
syms() 33
sym() 33
expm() 33
rank() 33
zeroes() 33
Problem examples 33
Find Eigenvalues 33
Create gain matrix 33
Compute numerical gain matrix solution 34
Solve differential equation with ode45 34

Definitions **34**

Synchronous Model

Parallel Composition



Inputs

- Input variables of Delay1 || Delay2 is {in}
  - Even though temp is input of Delay2, it is not an input of product
- If C1 has input vars I1 and C2 has input vars I2 then the product C1 || C2 has input vars (I1 U I2) \ (O1 U O2)
  - A variable is an input of the product if it is an input of one of the components, and not an output of the other

States

- State variables of Delay1 || Delay2 : {x1, x2}
- If C1 has state vars S1 and C2 has state vars S2 then the product has state vars (S1 U S2)
  - A state of the product is a pair (s1, s2), where s1 is a state of C1 and s2 is a state of C2
  - If C1 has n1 states and C2 has n2 states then the product has (n1 x n2) states

Initial states

- The initialization code Init for Delay1 || Delay2 is "x1:=0;x2:=0"
  - Initial state is (0,0)
- If C1 has initialization Init1 and C2 has initialization Init2 then the product C1 || C2 has initialization Init1; Init2
  - Order does not matter
  - [Init] is the product of sets [Init1] x [Init2]

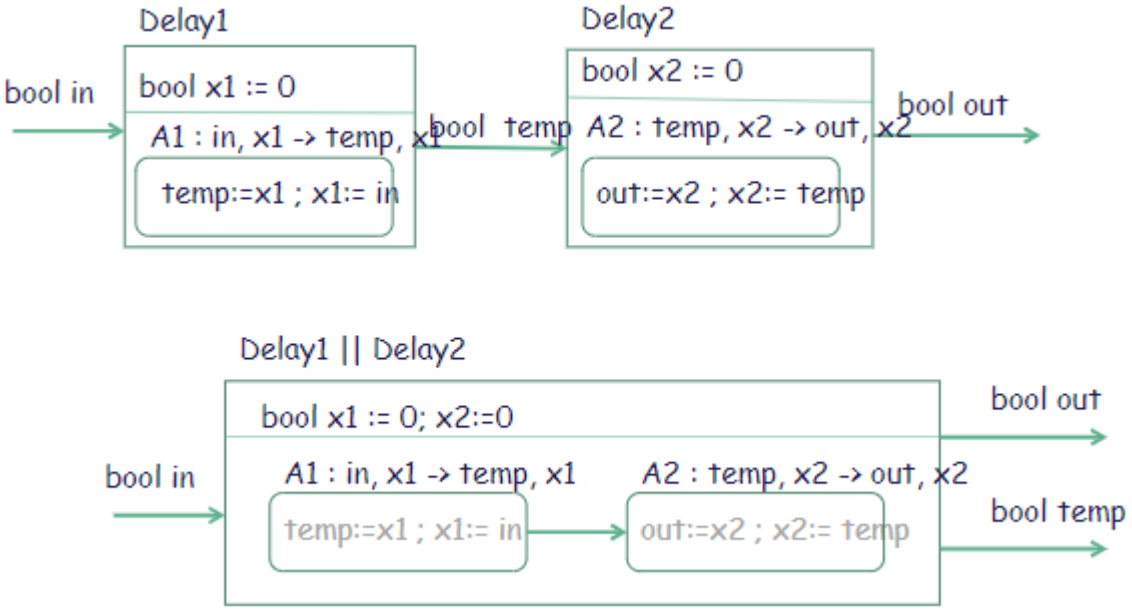
Requirements for Parallel composition

… slides 2

Component compatibility

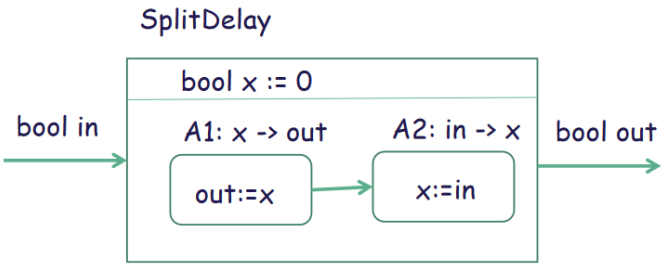
- Given:
  - Component C1 with input vars I1, output vars O1, and awaits-dependency relation $>_1$
  - Component C2 with input vars I2, output vars O2, and awaits-dependency relation $>_2$
- The components C1 and C2 are compatible if
  - No common outputs: sets O1 and O2 are disjoint
  - The relation $(>_1 \cup >_2)$ of combined await-dependencies is acyclic
- Parallel Composition is allowed only for compatible components

Example



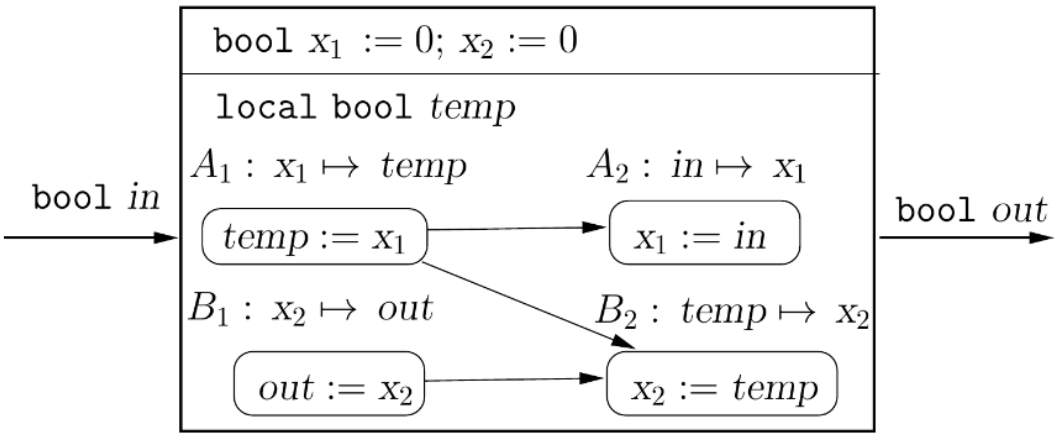
Textual description

From text to model



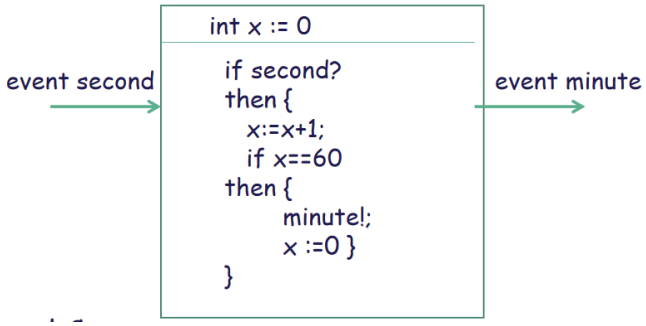
$(\texttt{SplitDelay}[\texttt{out} \mapsto \texttt{temp}] \| \texttt{SplitDelay}[\texttt{in} \mapsto \texttt{temp}]) \setminus \{\texttt{temp}\}$

Example:
↦ = Renameing
\ = Hiding
|| = Parallel composition



From model to textual description

SecondToMinute:



Create a SecondToHour component, using the above component:

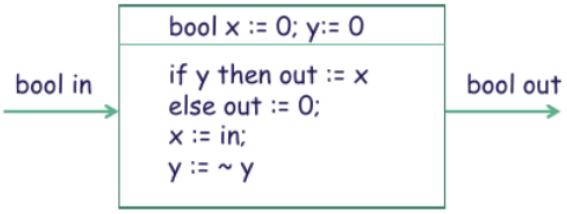
$(\texttt{SecondToMinute} \| \texttt{SecondToMinute}[\texttt{minute} \mapsto \texttt{hour}][\texttt{second} \mapsto \texttt{minute}]) \setminus \{\texttt{minute}\}.$

Execution/transitions format example

Lets assume ordering on state variables $(\mathtt{er}, \mathtt{r}_1, \mathtt{r}_2)$, and for outputs , $\mathtt{error}, \mathtt{alive}_1, \mathtt{alive}_2$ then an execution

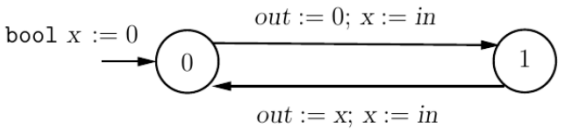
$$(false, 0, 0) \xrightarrow{\top/false,\top,\bot} (false, 1, 1) \xrightarrow{\top/false,\bot,\top} (false, 2, 2) \xrightarrow{\top/false,\bot,\bot}$$
$$(true, 3, 3) \xrightarrow{\top/true,\top,\bot} (true, 4, 4) \rightarrow \ldots$$

Describe a component as an extended-state machine



Solution

The extended-state-machine corresponding to the component `OddDelay` is shown below. The modes correspond to the values of the state variable y.



Mealy machine

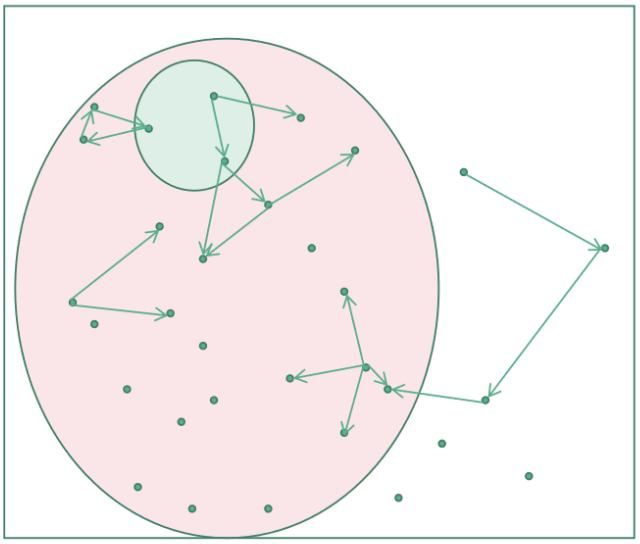
...

Safety Requirements

Safety Requirements

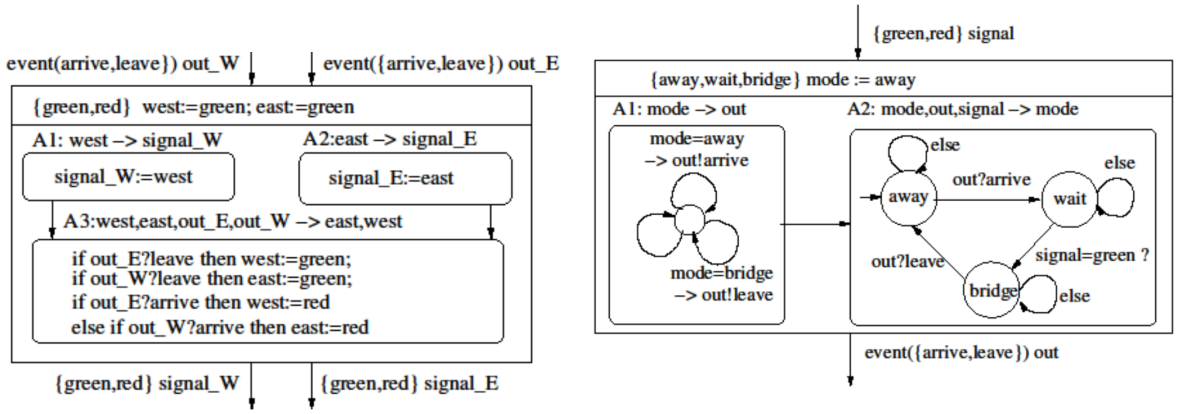
- A safety requirement states that a system always stays within "good' states (i.e. nothing bad ever happens)
- Cruise controller: Desired speed stays between bounds
- Collision avoidance: Distance between two cars is always greater than some minimum threshold
- Different class of requirements: Liveness (i.e. something good eventually happens)
  - System eventually attains its goal
  - Cruise controller: Actual speed eventually equals desired speed
- Formalization and analysis techniques for safety and liveness differ significantly, so let us first focus on safety

Reachable states



A state s of a transition system is reachable if there is an execution starting in an initial state and ending in the state s

Example



Exercise 2: Reachable states
The composed system

$$\texttt{RailRoadSystem1} = \texttt{Controller1} \| \texttt{Train}_\texttt{W} \| \texttt{Train}_\texttt{E}$$

(Chap 3, Slides 30-33) has four state variables, `east` and `west`, each of which can take two values, and $\texttt{mode}_W$ and $\texttt{mode}_E$, each of which can take three values. Thus, `RailRoadSystem1` has 36 states. How many of these 36 states are reachable?

Solution

Each state is denoted by listing the values of the variables `west`, `east`, $\texttt{mode}_W$, and $\texttt{mode}_E$, in that order. We use a, w, b, g, and r, as abbreviations for the values `away`, `wait`, `bridge`, `green`, `red`, respectively. Then, the initial state is $ggaa$, and has transitions to itself and to the states $rgaw$, $grwa$, and $rgww$. To compute the set of reachable states, we need to explore transitions from these three newly discovered states, and keep repeating till no new states are found to be reachable. It turns out that the following 13 states are reachable:

$$\{ggaa, rgaw, grwa, rgww, rgab, rrwb, grba, rrbw, rgwb, ggwa, ggba, rgbw, rgbb\}$$

Inductive Invariant

Inductive Invariant

- A property φ is an inductive invariant of transition system T if
  1. Every initial state of T satisfies φ
  2. If a state satisfies φ and (s,t) is a transition of T, then t must satisfy φ
- If φ is an inductive invariant, then all reachable states of T must satisfy φ, and thus, it is an invariant

Proving Invariants

- Given a transition system (S, Init, Trans), and a property φ, prove that all reachable states of T satisfy φ
- Inductive definition of reachable states
  - All initial states are reachable using 0 transitions
  - If a state s is reachable in k transitions and (s,t) is a transition, then the state t is reachable in (k+1) transitions
- Prove: for all n, states reachable in n transitions satisfy φ
  - Base case: Show that all initial states satisfy φ
  - Inductive case:

    Assume that a state s satisfies φ

    Show that if (s,t) is a transition then t must satisfy φ

Proving Inductive Invariant Example (1)

- Consider transition system T given by
  - State variable int x, initialized to 0
  - Transition description given by "if (x < m) then x:=x+1"
- Is the property φ : (0 <= x <= m) an inductive invariant of T ?
- Base case: Consider initial state (x=0). Check that it satisfies φ
- Inductive case:
  - Consider an arbitrary state s, suppose s(x) = a
  - Assume that s satisfies φ, that is, assume 0 <= a <=m
  - Consider the state t obtained by executing a transition from s
  - If a < m then t(x) = a+1, else t(x) = a
  - In either case, 0 <= t(x) <= m
  - So t satisfies the property φ, and the proof is complete

Proving Inductive Invariant Example (2)

- Consider transition system T given by
  - State variables int x, y; x is initially 0, y is initially m
  - Transition description given by "if (x < m) then {x:=x+1; y:=y-1}"
- Is the property φ : (0 <= y <= m) an inductive invariant of T ?
- Base case: Consider initial state (x=0,y=m). Check that it satisfies φ
- Inductive case:
  - Consider an arbitrary state s with x=a and y=b
  - Assume that s satisfies φ, that is, assume 0 <= b <=m
  - Consider the state t obtained by executing a transition from s
  - If a < m then t(y) = b-1, else t(y) = b
  - Can we conclude that 0 <= t(y) <= m ?
  - No! The proof fails! In fact, φ is not an inductive invariant of T!

Proving Inductive Invariant Example (3)

- Consider transition system T given by
  - State variables int x, y; x is initially 0, y is initially m
  - Transition description given by "if (x < m) then {x:=x+1; y:=y-1}"
- Property ψ : (0 <= y <= m) & (x+y = m)
- Base case: Consider initial state (x=0,y=m). Check that it satisfies ψ
- Inductive case:
  - Consider an arbitrary state s with x=a and y=b
  - Assume that s satisfies ψ, that is, assume 0 <= b <=m and a+b =m
  - Consider the state t obtained by executing a transition from s
  - If a < m then t(x) = a+1 and t(y) = b-1, else t(x)=a and t(y)=b
  - But if a<m, since a+b=m holds, b >0, and thus b-1 >=0
  - In either case, the condition (0<= t(y) <=m & t(x)+t(y)=m) holds!
- Conclusion: Property ψ is an inductive invariant!

Example

Exercise 3: Inductive invariants

Consider a transition system T with two integer variables x and y and a Boolean variable z. All the variables are initially 0. The transitions of the system correspond to executing the conditional statement

$$\textbf{if } (z = 0) \textbf{ then } \{x := x + 1;\ z := 1\} \textbf{ else } \{y := y + 1;\ z := 0\}$$

Consider the property φ given by $(x = y) \vee (x = y+1)$. Is φ an invariant of the transition system T? Is φ an inductive invariant of the transition system T? Find a formula ψ such that ψ is stronger than φ and is an inductive invariant of the transition system T . Justify your answers.

Solution

The system has a single execution given by (a state is specified by listing the values of x, y, and z, in that order):

$$(0,0,0) \to (1,0,1) \to (1,1,0) \to (2,1,1) \to (2,2,0) \to \ldots$$

The formula φ given by $(x = y \vee x = y+1)$ holds in every state of this execution, and is an invariant of the system. The formula φ, however, is not an inductive invariant. The state (1, 0, 0) satisfies the formula φ, and has a transition to the state (2, 0, 1), which does not satisfy the formula φ. Consider the formula ψ given by $(z = 0 \wedge x = y) \vee (z = 1 \wedge x = y + 1)$. Observe that if a state s satisfies ψ, it must satisfy one of the disjuncts in ψ, and thus, must satisfy either $(x = y)$ or $(x = y + 1)$, and thus, must satisfy φ. Thus, the property ψ is stronger than φ. The initial state (0, 0, 0) satisfies ψ. Consider a state s that satisfies ψ. Then s satisfies either $(z = 0 \wedge x = y)$ or $(z = 1 \wedge x = y + 1)$. In the former case, executing a transition from the state s increments x and sets z to 1, and thus, the resulting state satisfies $(z = 1 \wedge x = y + 1)$. By a similar reasoning if the state s satisfies $(z = 1 \wedge x = y + 1)$, then executing one transition from it leads to a state that satisfies $(z = 0 \wedge x = y)$. It follows that if there is a transition from the state s to state t, then the state t must satisfy ψ. Thus, the property ψ is an inductive invariant.

Image Computation

...

Safety Monitor

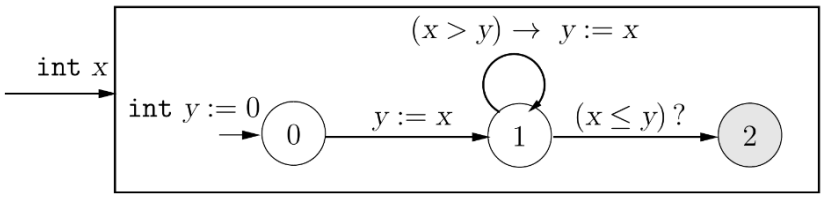
- Monitor M for a system observes its inputs/outputs, and enters an error state if undesirable behavior is detected
- Monitor M is specified as extended state machine
  1. The set of input variables of M = input/output variables of system being monitored
  2. An output of M cannot be an input to system (Monitor does not influence what the system does)
  3. A subset F of modes of state-machine declared as accepting
- Undesirable behavior: An execution that leads monitor state to F
- Safety verification: Check whether (monitor.mode not in F) is an invariant of System C || M

Example

Exercise 1: Monitor
Consider a component `C` with an output variable `x` of type int. Design a safety monitor to capture the requirement that the sequence of values output by the component `C` is strictly increasing (that is, the output in each round should be strictly greater than the output in the preceding round).

Solution

The monitor, shown below, maintains a state variable y to record the value of the input from the preceding round. The monitor enters the mode 2 exactly when the desired safety requirement gets violated.



Properties exercise example

(a) Consider transition system T with state variables a, b of type `bool`, and initial state s_0. The following set of executions

$$\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid \forall_i . s_i \models a \wedge b\}$$

describe the set of executions satisfying the property "Every state satisfies a and b". Using similar mathematical notation, describe the sets of executions satisfying the following properties:

(i) Every state satisfies a or b.

(ii) There is no state satisfying b before the first occurrence of a.

(iii) Every a will be eventually followed by an b.

(iv) Exactly three states satisfy a.

(v) If there are infinitely many a there are infinitely many b.

(vi) There are only finitely many a.

(b) Intuitively, violations (counter-examples) of safety properties are finite executions, whereas counter-examples of liveness properties are infinite executions. Which of the properties above are invariants, which are safety properties, and which are liveness properties?

Solution

(a) (i) $\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid \forall i. a \in s_i \models a \vee b\}$

(ii) $\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid \forall i. s_i \models b \Rightarrow \exists j \leq i. s_j \models a\}$

(iii) $\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid \forall i. s_i \models a \Rightarrow \exists j > i. s_j \models b\}$

(iv) $\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid \#\{i \mid s_i \models a\} = 3\}$

(v) $\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid (\forall i. \exists j > i. s_j \models a) \Rightarrow (\forall i. \exists j > i. s_j \models b)\}$

(vi) $\{s_0 \to s_1 \to s_2 \to s_3 \cdots \mid (\exists i. \forall j > i. s_j \not\models a)\}$

(b) Property (i) is an invariant and therefore a safety property. Property (ii) is a safety property. Property (iii,v,vi) are liveness properties. Property (iv) is neither safety nor liveness property. The property can be described as (3 or less states satisfy a) and (3 or more states satisfy a), the former is a safety property and the latter is a liveness property.

Asynchronous Model

- Recap: In a synchronous model, all components execute in a sequence of (logical) rounds in lock-step
- Asynchronous: Speeds at which different components execute are independent (or unknown)
  - Processes in a distributed system
  - Threads in a typical operating system such as Linux/Windows
- Key design challenge: how to achieve coordination?

Create Asynchronous process example

Exercise 1: Asynchronous process `Split`

We want to design an asynchronous process `Split` that is the dual of `Merge` (c.f. Figure 1 or Chapter 4, Slide 6). The process `Split` has one input channel `in` and two output channels `out1` and `out2` . The messages received on the input channel should be routed to one of the output channels in a nondeterministic manner so that all possible splittings of the input stream are feasible executions. Describe all the components of the desired process `Split`.

..... Solution .....

The asynchronous process `Split` has a single input variable `in` of type `msg`. Its output variables are $\texttt{out}_1$ and $\texttt{out}_2$ of type `msg`. It maintains a single queue as its state variable with the declaration given by

$$\texttt{queue(msg)x} := \texttt{null}.$$

The input task A_i specified by

$$\neg\texttt{Full(x)} \rightarrow \texttt{Enqueue(in, x)}$$

stores the messages arriving on the input channel `in` the queue `x`. The output task A_o^1 is enabled when the queue `x` is nonempty and if so, removes a message from the queue and transmits it on the output channel $\texttt{out}_1$:

$$\neg\texttt{Empty(x)} \rightarrow \texttt{out}_1 := \texttt{Dequeue(x)}.$$

The output task A_o^2 is symmetric, and transmits messages on the output channel $\texttt{out}_2$:

$$\neg\texttt{Empty(x)} \rightarrow \texttt{out}_2 := \texttt{Dequeue(x)}.$$

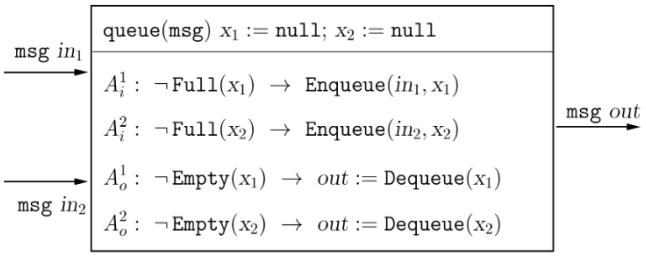


Fig. 1: Asynchronous process `Merge`

Asynchronous process composition

- Given asynchronous processes P1 and P2, how to define P1 | P2 ?

- Note: In each step of execution, only one task is executed
  - Concepts such as await-dependencies, compatibility of interfaces, are not relevant

- Sample case (see textbook for complete definition):
  - if y is an output channel of P1 and input channel of P2, and
  - A1 is an output task of P1 for y with code: Guard1 → Update1
  - A2 is an input task of P2 for y with code: Guard2 → Update2, then
  - Composition has an output task for y with code:
    (Guard1 & Guard2) → Update1 ; Update2

Example

Exercise 2: Asynchronous process composition

The asynchronous process `DoubleBuffer` is the result of the parallel composition of two `Buffer` components. In Chapter 4, Slide 17 we describe the "compiled" version the `DoubleBuffer` process. In the following, consider the asynchronous process

$$\texttt{Merge}[\texttt{out} \mapsto \texttt{temp}] \| \texttt{Merge}[\texttt{in}_1 \mapsto \texttt{temp}][\texttt{in}_2 \mapsto \texttt{in}_3]$$

obtained by connecting two instances of the process `Merge` (c.f. Figure 1). Show the "compiled" version of this composite process similar to the description of `DoubleBuffer`. Explain the input/output behavior of this composite process.

Solution

The input channels are $\texttt{in}_1$, $\texttt{in}_2$, and $\texttt{in}_3$, all of type `msg`. The output channel is `out`. When composing the two instances of `Merge`, we need to make sure that the state variables have distinct names. The state variables of the composite process and their initialization is specified by

$$\texttt{queue(msg)}\, \texttt{x}_1 := \texttt{null}; \texttt{x}_2 := \texttt{null}; \texttt{y}_1 := \texttt{null}; \texttt{y}_2 := \texttt{null}.$$

The composite process has three input tasks corresponding to its three input channels specified by:

$$\begin{array}{lcl} A_i^1 & : & \neg\texttt{Full}(\texttt{x}_1) \rightarrow \texttt{Enqueue}(\texttt{in}_1, \texttt{x}_1) \\ A_i^2 & : & \neg\texttt{Full}(\texttt{x}_2) \rightarrow \texttt{Enqueue}(\texttt{in}_2, \texttt{x}_2) \\ A_i^3 & : & \neg\texttt{Full}(\texttt{y}_2) \rightarrow \texttt{Enqueue}(\texttt{in}_3, \texttt{y}_2) \end{array}$$

The composition has two internal tasks, each of which is obtained by synchronizing an output of the first instance on the channel `temp` with a corresponding input processing by the second:

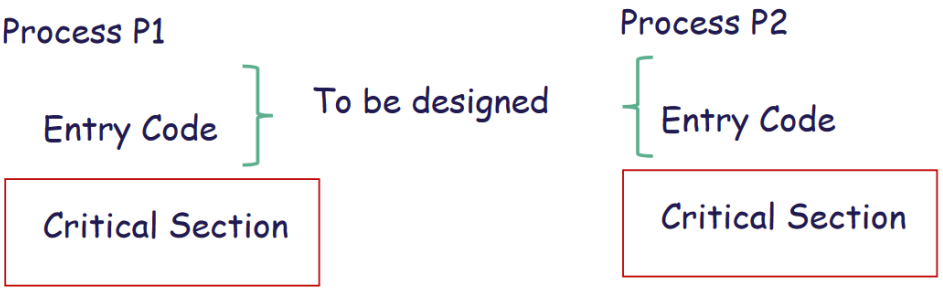
$$\begin{array}{lcl} A^1 & : & \neg\texttt{Empty}(\texttt{x}_1) \wedge \neg\texttt{Full}(\texttt{y}_1) \rightarrow \\ & & \{\texttt{local msg temp} := \texttt{Dequeue}(\texttt{x}_1);\ \texttt{Enqueue}(\texttt{temp}, \texttt{y}_1)\} \\ A^2 & : & \neg\texttt{Empty}(\texttt{x}_2) \wedge \neg\texttt{Full}(\texttt{y}_1) \rightarrow \\ & & \{\texttt{local msg temp} := \texttt{Dequeue}(\texttt{x}_2);\ \texttt{Enqueue}(\texttt{temp}, \texttt{y}_1)\} \end{array}$$

Finally, the composite process has two output tasks that remove messages from the queues $\texttt{y}_1$ and $\texttt{y}_2$ in order to transmit them on the output channel `out`:

$$\begin{array}{lcl} A_o^1 & : & \neg\texttt{Empty}(\texttt{y}_1) \rightarrow \texttt{out} := \texttt{Dequeue}(\texttt{y}_1); \\ A_o^2 & : & \neg\texttt{Empty}(\texttt{y}_2) \rightarrow \texttt{out} := \texttt{Dequeue}(\texttt{y}_2); \end{array}$$

The sequence of values output by the composite process represents a merge of the sequences of input values received on the three input channels. The relative order of values received on each of the input channels is preserved in the output sequence, but the three input sequences can be interleaved in any nondeterministic order.

Mutual exclusion protocol



- Critical Section: Part of code that an asynchronous process should execute without interference from others
  - Critical section can include code to update shared objects/database
- Mutual Exclusion Problem: Design code to be executed before entering critical section by each process
  - Coordination using shared atomic registers
  - No assumption about how long a process stays in critical section
  - A process may want to enter critical section repeatedly

Fairness assumptions

- Fairness assumption is an assumption made about the underlying platform or scheduler
  - Less the assumptions, the better
- For each output and internal task, we can assume weak or strong fairness, as needed
  - Strong fairness needed if the task can switch between enabled and disabled due to execution of other tasks
- Restricts the set of possible infinite executions, and allows satisfaction of more requirements
  - Does not affect the set of reachable states and safety properties (realizable fairness)
  - Fairness assumptions do not change underlying coordination
- Key distinction: Fairness assumption for tasks (which ensure tasks get executed as expected) vs "fairness" requirements for protocols (which are about high-level goals of the problem being solved)

Weak vs Strong Fairness

- Definition 2 (Weak fairness): An infinite execution is fair to a task A, if repeatedly, either task A is executed or is disabled (if almost always enabled then inf often taken)

Process P3

nat x:=0; y:=0
A_x: x := x+1
A_y: even(x) → y :=y+1

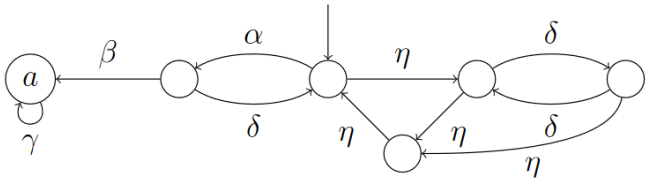
- Is this fair to task A_y of P3?

  (0,0) $-A_x\rightarrow$ (1,0) $-A_x\rightarrow$ (2,0) $-A_x\rightarrow$ (3,0) ... $-A_x\rightarrow$ (105,0) $-A_x\rightarrow$ ...
- Task A_y is not continuously enabled, and thus, this is fair according to definition 2
- Definition 3 (Strong fairness): An infinite execution is fair to a task A, if task A is either executed repeatedly, or disabled from a certain step onwards (if inf often enabled then inf often taken)
- Above execution is weakly-fair to task A_y, but not strongly-fair

Exercise example

Exercise 4: Fairness assumptions

Consider the following extended state machine with tasks $\{\alpha, \beta, \gamma, \delta, \eta\}$:



Under which fairness assumptions does the system satisfy the property "for all executions, eventually a"? Justify your answer.

(a) Infinitely often γ

(b) Infinitely often α and γ

(c) Infinitely often α or γ

(d) Strong-fairness for β

(e) Strong-fairness for α and β.

(f) Strong-fairness for α, β and η.

(g) Weak-fairness for α, β and η.

(h) Strong-fairness for α, β and Weak-fairness for η.

Solution

(a) Holds: demands that γ occurs infinitely often, so it occurs at least once. Therefore, the left state must be reached eventually.

(b) Holds: demands that γ occurs infinitely often. A closer inspection yields that there is no fair run. However, this means that the property is vacuously true.

(c) Does not hold: A fair run is the α-δ-loop, since only one of α and γ has to occur infinitely often.

(d) Does not hold: The η-loop is fair, since β is not infinitely often enabled.

(e) Does not hold: The right-most δ-loop is fair, since neither α nor β are infinitely often enabled.

(f) Holds: A fair run cannot stay in the right most loop, since η is infinitely often enabled. Therefore η must be infinitely often taken, so the starting state is visited infinitely often. Then α is infinitely often enabled; it is therefore infinitely often taken and the second state from the left is visited infinitely often. Therefore β has to be taken.

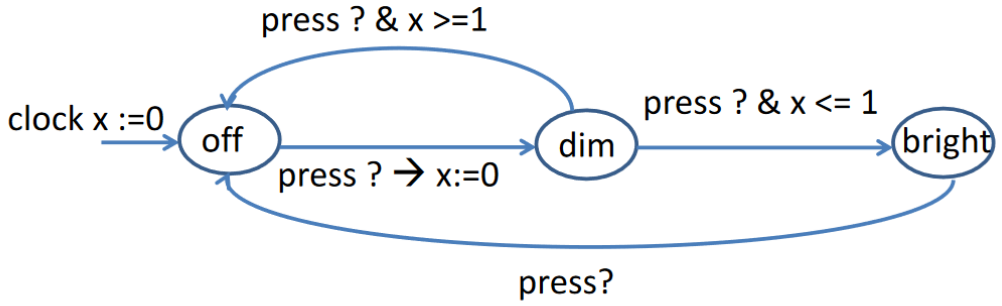
(g) Does not hold: The η-loop is fair. Since α is infinitely often not enabled, it does not have to be taken.

(h) Holds: The δ-loop is not fair, because η is always enabled in that loop. Therefore η has to be taken and α is infinitely often enabled. Thus as in an earlier case, α and therefore β have to be taken.

Transition systems

Timed model/Automata

- Timed model
  - Like asynchronous for communication of information
  - Can rely on global time for coordination



Initial state = (mode = off, x = 0)

Timed transition: (off, 0) – 0.5 → (off, 0.5)

Input transition: (off, 0.5) – press? → (dim, 0)

Timed transition: (dim, 0) – 0.8 → (dim, 0.8)

Input transition: (dim, 0.8) – press? → (bright, 0.8)

Timed transition: (dim, 0.8) – 1 → (dim, 1.8)

Input transition: (dim, 1.8) – press? → (off, 1.8)

Timed Automata

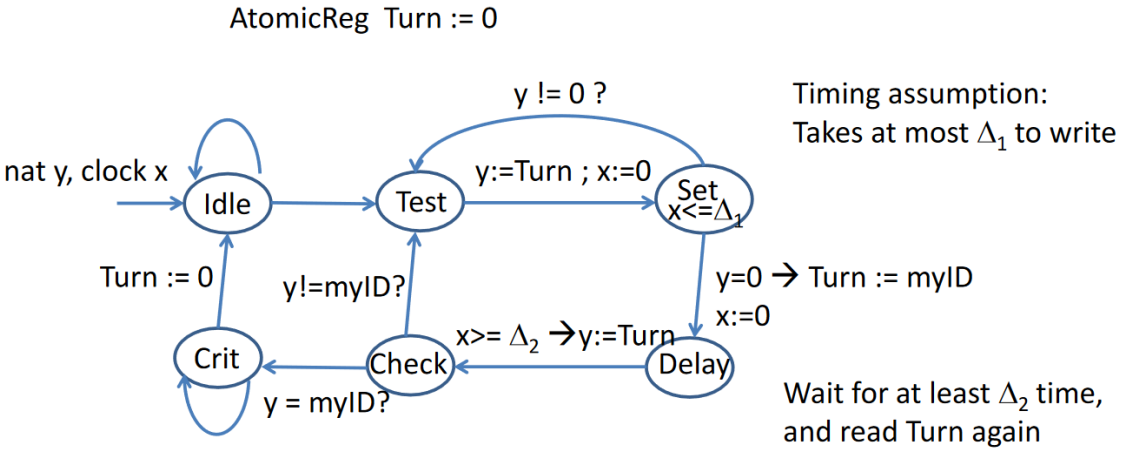
- Motivation: When is exact analysis of timing constraints possible?
- A timed process TP is a timed automaton if for every clock variable x
  - Assignments to x in the description of TP are of the form x:=0
  - An atomic expression involving x in the description of TP (in clock-invariants or in guards) must of the form "x~ k", where k is a constant and ~ is a comparison operation (=, <=, >, <, >=)
- In such a model, one can express constant lower and upper bounds on timing delays
  - Closed under parallel composition: If TP1 and TP2 are timed automata then TP1 | TP2 is also a timed automaton
- Finite-state timed automaton: A timed automaton where all variables other than clock variables have finite types (e.g. Boolean, enumerated)
  - State-space is still infinite due to clock variables, but verification is solvable exactly

Fisher's Protocol for Mutual Exclusion

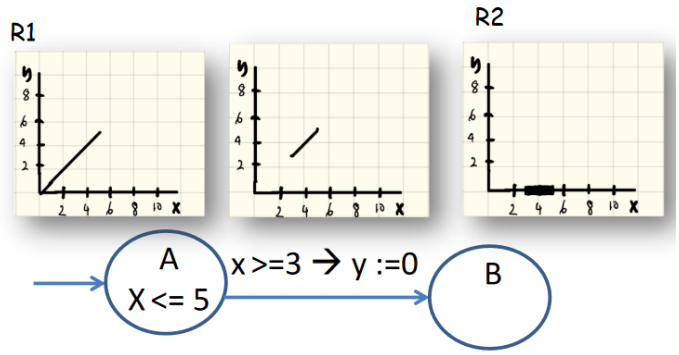
Properties:

- Assuming $\Delta_2 > \Delta_1$, the algorithm satisfies:
  - Mutual exclusion: Two processes cannot be in critical section simultaneously
  - Deadlock freedom: If a process wants to enter critical section then some process will enter critical section
- Protocol works for arbitrarily many processes (not just 2)
  - Note: In the asynchronous model, mutual exclusion protocol for N processes is lot more complex than Peterson's algorithm
- Does the protocol satisfy the stronger property of starvation freedom: If a process P_i wants to enter critical section then it eventually will ?
- If $\Delta_2 <= \Delta_1$ then does mutual exclusion hold? Does deadlock freedom hold?

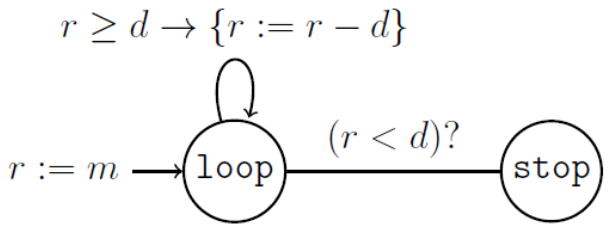
Example:



Timing Analysis example



Symbolic Transition System



Describe the transitions system example 1

The transition system `REM(m,n)` has state variable r and $mode$ of the enumerated type $\{loop, stop\}$. The initialization is given by the formula

$$(mode = loop) \wedge (r = m)$$

The transition formula φ is given as:

$$\begin{array}{ll} & [(mode = loop) \wedge (r \geq d) \wedge (r' = r - d) \wedge (mode' = loop)] \\ \vee & [(mode = loop) \wedge (r < d) \wedge (r' = r) \wedge (mode' = stop)] \end{array}$$

Computing image example 1

- Conjuction of A and φ, note $A \equiv (100 \leq r \wedge r \leq 290)$

$$(100 \leq r \leq 290) \wedge [(mode = loop) \wedge (r \geq 9) \wedge (r' = r - 9) \wedge (mode' = loop)]$$

- Existententially quantify $mode$

$$(100 \leq r \leq 290) \wedge (r \geq 9) \wedge (r' = r - 9) \wedge (mode' = loop)$$

- Existententially quantify r

$$(100 \leq r' + 9) \wedge (r' + 9 \leq 290) \wedge (mode' = loop)]$$

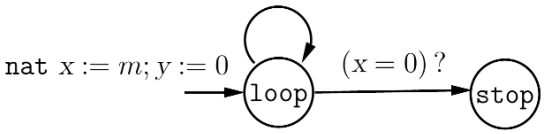
- Renaming

$$(91 \leq r \leq 281) \wedge (mode' = loop)$$

Describe the transitions system example 2

Exercise 2: Symbolic transition system `Mult`
The following state machine is used to compute the multiplication of two natural numbers:

$$(x > 0) \;\rightarrow\; \{x := x - 1;\; y := y + n\}$$



Describe this transition system symbolically using initialization and transition formulas.

Solution

The transition system `Mult(m,n)` has state variables x of type nat, y of type nat, and $mode$ of the enumerated type $\{loop, stop\}$. The initialization is given by the formula

$$(mode = loop) \wedge (x = m) \wedge (y = 0).$$

The transition formula is given as:

$$\begin{array}{ll} & [(mode = loop) \wedge (x > 0) \wedge (x' = x - 1) \wedge (y' = y + n) \wedge (mode' = loop)] \\ \vee & [(mode = loop) \wedge (x = 0) \wedge (x = x) \wedge (y = y) \wedge (mode = stop)] \end{array}$$

Computing image example 2

Exercise 3: Image computation
Consider the symbolic image computation for a transition system with two real-valued variables x and y and transition description given by the formula $x' = x + 1 \wedge y' = x$. Suppose the region A is described by the formula $0 \leq x \leq 4 \wedge y \leq 7$. Compute the formula describing the post-image of A.

Solution

Conjunction of the given region A and the transition formula gives

$$(x' = x + 1) \wedge (y' = x) \wedge (0 \leq x \leq 4) \wedge (y \leq 7)$$

The existential quantification of the unprimed variables leads to $(x' = y' + 1) \wedge (0 \leq y' \leq 4)$. Renaming the primed variables to their unprimed counterparts gives the desired post-image:

$$(x = y + 1) \wedge (0 \leq y \leq 4).$$

Real-Time Scheduling

Problem: only one CPU.

- only one task can execute at a time

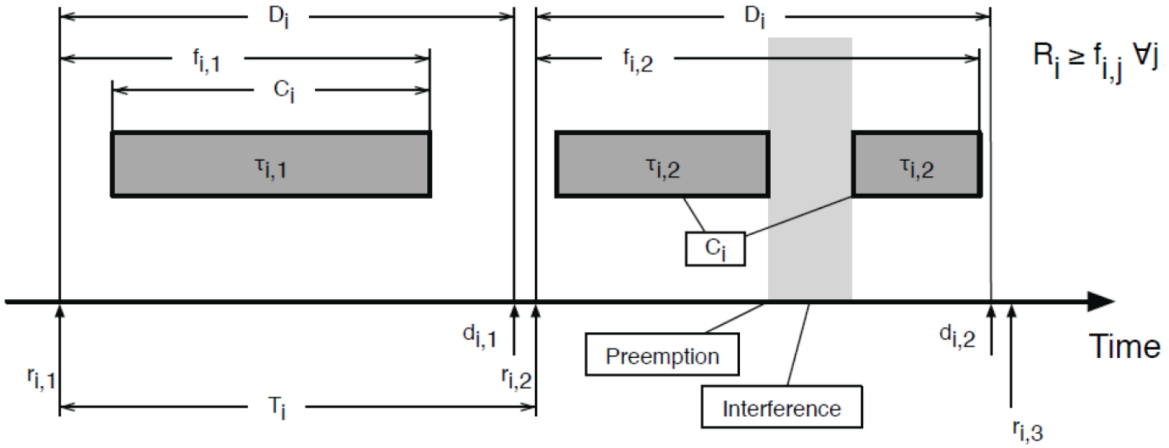
Scheduling:

- which task should execute a given point in time.

Questions:

- how frequent should a task execute in order not to miss important events?
- What is the execution time of a task.

Symbol	Meaning
τ_i	i-th task
$\tau_{i,j}$	j-th instance of τ_i
T_i	period of τ_i
D_i	relative deadline of τ_i
C_i	cost (WCET) of τ_i
R_i	worst-case response time of τ_i
$r_{i,j}$	release time of $\tau_{i,j}$
$f_{i,j}$	response time of $\tau_{i,j}$
$d_{i,j}$	absolute deadline of $\tau_{i,j}$



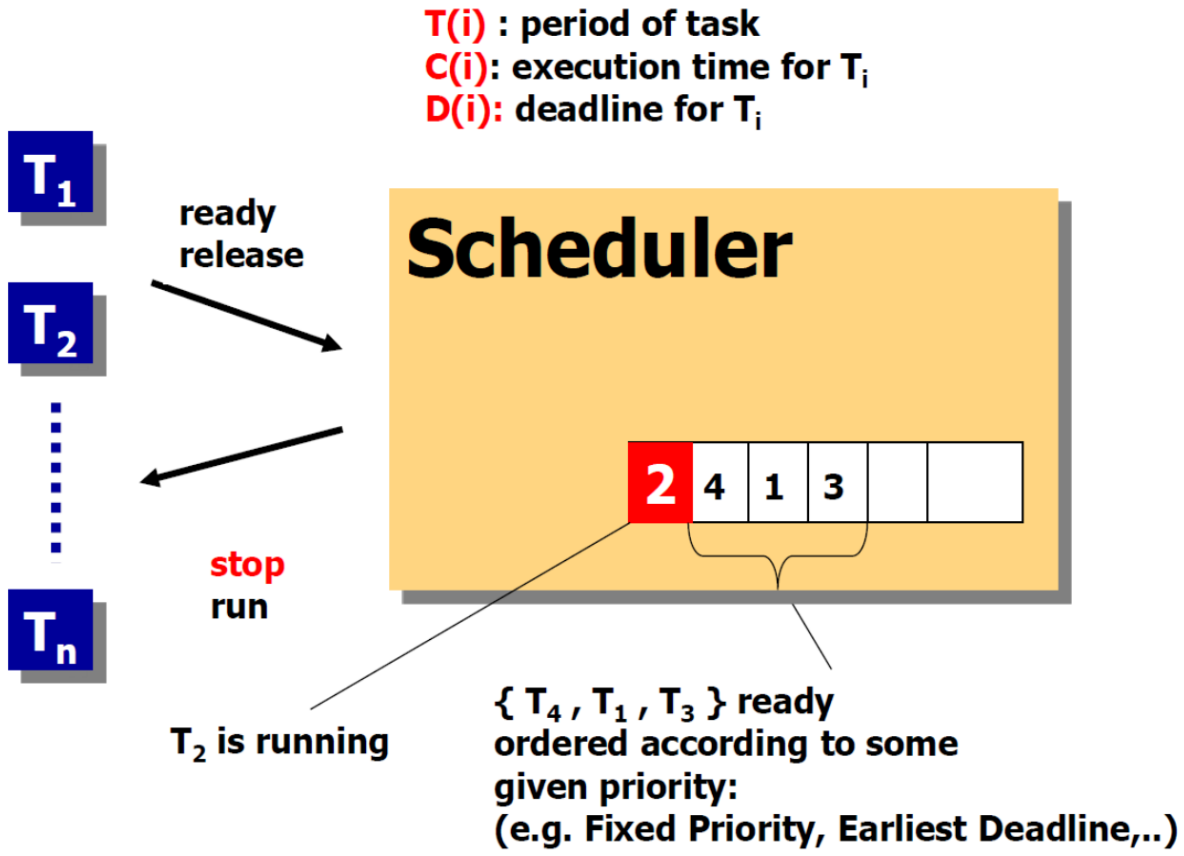
Feasible Schedule: $\forall i. R_i \leq D_i$

The Cyclic Executive Approach

Properties

- Offline generation of static schedule
- Tasks (or parts thereof) are implemented as procedures
- Procedures are mapped onto a sequence of minor cycles
- Minor cycles constitute the complete schedule: the major cycle
- Minor cycle determines the minimum period
- Major cycle determines the maximum cycle time
- Uses real time clock/interrupt for synchronisation
- Minor cycles are synchronisation points
- Concurrent design, but sequential code (collection of procedures)

Task Scheduling Architectures



Fixed Priority Scheduling (FPS)

Definition (FPS)

- Each process has a fixed, static, priority assigned before run-time
- Priority determines execution order

Why FPS?

- Most widely used approach
  - Conceptually simple
  - Well-understood
  - Well-supported
- Main focus of the course

Priority $\neq$ Importance

In RTSs the "priority" of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity

How to assign priorities for FPS

- Rate Monotonic Priority Assignment
- Each process is assigned a (unique) priority based on its period: $T_i < T_j \implies P_i > P_j$
- Note: priority 1 (one) is the lowest (least) priority

Example (Priority Assignment)

Process	Period(T)	Priority (P)
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Earliest Deadline First (EDF)

Definition (EDF)

- Execution order is determined by the absolute deadlines
- The next process to run is the one with the shortest (nearest) deadline
- Absolute deadlines must be computed at run-time (dynamic scheduling)

Why (not) EDF?

- Optimal (anything schedulable in FPS can be scheduled in EDF)
- Less supported by language/OS
- Harder to implement / greater overhead

Uppaal

Reachable states

(I ran `verifyta.exe -u GCD.xml` inside \uppaal64-4.1.26\bin-Windows because Windows).

Verifier

Cheat sheet

Name	Property	Equivalent to
Possibly	`E<> p`	
Invariantly	`A[] p`	`not E<> not p`
Potentially always	`E[] p`	
Eventually	`A<> p`	`not E[] not p`
Leads to	`p --> q`	`A[] (p imply A<> q)`

“A[]” = This holds for all branches (Invariant)
“E[]” = (Potentially always)

- There is a branch where this holds for all states

“E<>” = There is a branch where this holds (Possibly)

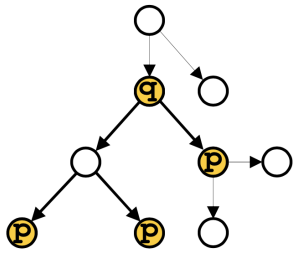
- There is a branch where this holds at some point

“A<>” = All branches will eventually hold for this (Eventually)

- Eventually, all branches will be in this state at some point

“-->” = The left side leads to the right side (Leads to)

`q-->p` “q always leads to p”
Any path that “starts” with a state in which q holds reaches later a state in which p holds



“deadlock” = look for deadlock. Usage example: “A[] not deadlock”
“imply” = left side implies the right

MatLab

(A, B, C, D) Notation

- Suppose a linear continuous-time component has
  - n state variables $S = \{x_1, x_2, \ldots x_n\}$
  - m input variables $I = \{u_1, u_2, \ldots u_m\}$
  - k output variables $O = \{y_1, y_2, \ldots y_k\}$
- Then the dynamics is given by $dS/dt = A\,S + B\,I$ and $O = C\,S + D\,I$
  - A is (n x n) matrix
  - B is (n x m) matrix
  - C is (k x n) matrix
  - D is (k x m) matrix
- Rate of change of i-th state variable given by

  $dx_i/dt = A_{i,1}x_1 + A_{i,2}x_2 + \ldots + A_{i,n}x_n + B_{i,1}u_1 + B_{i,2}u_2 + \ldots + B_{i,m}u_m$
- Value of j-th output variable given by

  $y_j = C_{j,1}x_1 + C_{j,2}x_2 + \ldots + C_{j,n}x_n + D_{j,1}u_1 + D_{j,2}u_2 + \ldots + D_{j,m}u_m$

Cheat sheet

Functions list/explanations

place()

Using the place function, you can compute a gain matrix K that assigns these poles to any desired locations in the complex plane (provided that (A,B) is controllable). For example, for state matrices A and B , and vector p that contains the desired locations of the closed loop poles, K = place(A,B,p);

plot()

plot(X , Y) creates a 2-D line plot of the data in Y versus the corresponding values in X .

ss()

sys = ss(A,B,C,D) creates a continuous-time state-space model object of the following form:

- $x=A_x+B_u$
- $y=C_x+D_u$

step()

step(sys) plots the response of a dynamic system model to a step input of unit amplitude.

ode45()

[t,y] = ode45(odefun,tspan,y0), where tspan = [t0 tf], integrates the system of differential equations y′=f(t,y) from t0 to tf with initial conditions y0. Each row in the solution array y corresponds to a value returned in column vector t.

eig()

e = eig(A) returns a column vector containing the eigenvalues of square matrix A.

inv()

Y = inv(X) computes the inverse of square matrix X.

syms()

syms var1 ... varN creates symbolic scalar variables var1 ... varN of type sym. Separate the different variables with spaces.

sym()

x = sym('x') creates symbolic variable x.

expm()

Y = expm(X) computes the matrix exponential of X.

rank()

k = rank(A) returns the rank of matrix A. (Linearly independent rows)

zeroes()

X = zeros(n) returns an n-by-n matrix of zeros.

Problem examples

Find Eigenvalues

```
A = [ [-3  2]; [-2  2] ];
eig(A)
```

```
ans =
-2
 1
```

Create gain matrix

```
A = [ [-3  2]; [-2  2] ];
B = [ 1; 0 ];
K = place(A,B,[-2,-3])
lambdas = eig(A - B*K)   % double check, not required
```

```
K = [4    -8]
lambdas =
-3
-2
```

Compute numerical gain matrix solution

gain matrix values from:

(c) Compute by hand the gain matrix K that ensures that $A - BK$ has poles/eigenvalues -1 and -2.

(d) Cross-check the gain matrix by computing it using the MATLAB command `place`. Compute afterwards a numeric solution using the commands `ss` and `initial`. Use as initial condition $(x(0), v(0)) = (-100, 0)^T$.

(d) A possible Matlab code is:

Listing 1: Matlab code

```
% (A,B,C,D) representation
A = [ [ 0  1]; [0  -0.05] ];
B = [0;  0.001];
C = [ 1  0 ];
D = 0;
% Place the poles
K = place(A,B,[-1,-2])
% Create closed loop system
syscl = ss(A-B*K,zeros(2,1),C,D);
% Solve numerically and plot
initial(syscl,[-100,0])
```

Solve differential equation with ode45

Exercise 2: Lipschitz continuity

(a) Consider the differential equation $\frac{d}{dt}x(t) = g(x(t))$ with $g(x) = x^2 - x$. Using the MATLAB commands `ode45s` and `plot`, compute and plot the numeric solution of $\frac{d}{dt}x(t) = g(x(t))$ in the case of initial value $x(0) = 2.0$ on time intervals $[0; 0.67]$, $[0; 0.68]$, $[0; 0.69]$ and $[0; 0.70]$. Can you explain what happens?

Solution

(a) Create a Matlab file called `odeExample.m` with the content below (file names are important because the file name identifies the main function of a file). Once done, run the file.

Listing 3: Matlab code

```
% main function
function odeExample()
    % finite time horizon
    T = 0.67; % try also 0.68, 0.69 and 0.70
    % initial condition
    x0 = [2.0];
    % time points at which solution should be approximated
    dt = T / 100;
    I = 0:dt:T;
    % invocation of numeric ODE solver
    [I,x]=ode45(@odeDrift,I,x0);
    % plot the vector/matrix
    plot(I(:),x(:,1));
end

% auxiliary function describing the ODE
function dx = odeDrift(t,x)
  dx=zeros(1,1);
  dx(1) = x(1)*x(1) - x(1);
end
```

Definitions

Timed Computation Tree Logic (TCTL)