Lecture 14: Discrete Control

[Chapter: Sequential Control + These Slides]

- Discrete Event Systems
- State Machine-Based Formalisms
- Statecharts
- Grafcet
- Laboratory 2
- Petri Nets
- Implementation
  - Not covered in the lecture. Homework.

Discrete Event Systems

Definition:

A Discrete Event System (DES) is a discrete-state, event-driven system, that is its state evolution depends entirely on the occurrence of asynchronous discrete events over time.

Sometimes the name Discrete Event Dynamic System (DEDS) is used to emphasize the dynamic nature of DES.

Continuous System

State trajectory is the solution of a differential equation

\[ x(t) = f(x(t), u(t), t) \]

\[ X = R \]

Discrete Control Systems

All processes contain discrete elements:

- continuous processes with discrete sensors and/or actuators
- discrete processes
  - manufacturing lines, elevators, traffic systems, ...
- mode changes
  - manual/auto, startup/shutdown
  - production (grade) changes
- alarm and event handling

Discrete Event System

State trajectory (sample path) is piecewise constant function that jumps from one value to another when an event occurs.
**Discrete Logic**

- Larger in volume than continuous control
- Very little theoretical support
  - verification, synthesis
  - formal methods beginning to emerge
  - still not widespread in industry

**Basic Elements**

- Boolean (binary) signals – 0, 1, false, true, a, a
- expressions
  
  \[
  a \quad \quad \quad a \text{ or } b \quad \quad \quad (a \text{ or } b) \quad \quad \quad (a \text{ and } b) \quad \quad \quad \text{Truth values} \quad \text{Truth value tables}
  \]

  Boolean algebra

- events
  
  \[
  a \quad \quad \quad a^\prime \quad \quad \quad a^\prime \quad \quad \quad a^\prime
  \]

**Logic Nets**

- Combinatorial nets
  - outputs = f(inputs)
  - interlocks, "förreglingar"
- Sequence nets
  - newstate = f(state,inputs)
  - outputs = g(state,inputs)
  - state machines
  - automata

Asynchronous nets or synchronous (clocked) nets

**State Machines**

Formal properties \(\Rightarrow\) analysis possible in certain cases

Using state machines is often a good way to structure code.

Systematic ways to write automata code often not taught in programming courses.

**Moore Machine**

State transitions in response to input events

Output events (actions) associated with states
Mealy Machine

State Machine Extensions

Ordinary state machines lack structure
Extensions needed to make them practically useful
- hierarchy
- concurrency
- history (memory)

Statecharts

D. Harel, 1987
Statecharts =
- state machine
- hierarchy
- concurrency
- history

Statechart Syntax

AND Superstates:

Y is the orthogonal product of A and D
When in state (B,F) and event a occurs, the system transfers simultaneously to (C,G).
**History Arrows**

On event ‘a’ the last visited state within D becomes active.

**Syntax**

Interfaces for AND Superstates:

- $\delta$ exit from $J \Rightarrow (B, E)$
- $\alpha$ exit from $K \Rightarrow (C, F)$
- $\nu$ exit from $J \Rightarrow (B, F)$
- $\beta$ exit from $L \Rightarrow (C, \text{most recently visited state in } D)$
- $\omega$ exit from $(B, G) \Rightarrow K$
- $\eta$ exit from $(B, F) \Rightarrow H$
- $\theta$ exit from $(C, D) \Rightarrow K$
- $\epsilon$ exit from $(A, D) \Rightarrow L$

**Statechart Semantics**

Unfortunately, Harel only gave an informal definition of the semantics. As a result, a number of competing semantics were defined. In 1996, Harel presented his semantics (the Statemate semantics) of Statechart and compared with 11 other semantics. The lack of a single semantics is still the major problem with Statecharts. Each tool vendor defines his own.

- Discrete Event Systems
- State Machine-Based Formalisms
- Statecharts
- Grafcet
- Laboratory 2
- Petri Nets
- Implementation
  - Not covered in the lecture. Homework.
Grafnet

Extended state machine formalism for implementation of sequence control

Industrial name: Sequential Function Charts (SFC)

Defined in France in 1977 as a formal specification and realization method for logical controllers

Part of IEC 61131-3 (industry standard for PLC controllers)

Basic elements

Steps:
- active or inactive

Steps:
- active or inactive

Transitions ("övergång"):
- condition true and/or event occurred + previous step active

Control structures

Alternative paths:
- branches
- repetition

Parallel paths:

Legal Grafnet

Legal Grafnet

Semantics

1. The initial step(s) is active when the function chart is initiated.

2. A transition is fireable if:
   - all steps preceding the transition are active (enabled).
   - the receptivity (transition condition and/or event) of the transition is true

   A fireable transition must be fired.

3. All the steps preceding the transition are deactivated and all the steps following the transition are activated when a transition is fired

4. All fireable transitions are fired simultaneously

5. When a step must be both deactivated and activated it remains activated without interrupt
a = 1 or 0
a = 0

a = 1

a) Not enabled
b) Enabled but not firable
c) Firable
d) After the change from c)

Unreachable grafcets

Unsafe grafcets

Actions

Action types:
- standard action (level action)
- stored action (impulse action)

logical assignment

• stored action (impulse action)

Unstable situation
(stored actions performed)
Time-limited action

Time-delayed action

Hierarchy

Macro Steps:

Graf cet/SFC and IEC-1131 Editors

A large number of graphical IEC 1131-3 editors are available. Generates PLC code or C-code.

Laboratory 2

Sequential Control
- bead sorter process

JGrafchart - Lund University
- Graf cet/SFC graphical editor
- Graf cet/SFC run-time system

- Discrete Event Systems
- State Machine-Based Formalisms
- Statecharts
- Graf cet
- Laboratory 2
- Petri Nets
- Implementation
  - Not covered in the lecture. Homework.
Process

Bead Sorter process

- Discrete Event Systems
- State Machine-Based Formalisms
- Statecharts
- Grafcet
- Laboratory 2
- Petri Nets
  - Implementation
    - Not covered in the lecture. Homework.

Petri Nets

C.A Petri, TU Darmstadt, 1962

A mathematical and graphical modeling method.

Describe systems that are:
- concurrent
- asynchronous or synchronous
- distributed
- nondeterministic or deterministic

Petri Nets

Can be used at all stages of system development:
- modeling
- analysis
- simulation/visualization ("playing the token game")
- synthesis
- implementation (Grafcet)

Application areas

- communication protocols
- distributed systems
- distributed database systems
- flexible manufacturing systems
- logical controller design
- multiprocessor memory systems
- dataflow computing systems
- fault tolerant systems
- ...

Introduction

A Petri net is a directed bipartite graph consisting of places $P$ and transitions $T$.

Places are represented by circles.

Transitions are represented by bars (or rectangles)

Places and transitions are connected by arcs.

In a marked Petri net each place contains a cardinal (zero or positive integer) number of tokens of marks.
Firing rules

1. A transition $t$ is enabled if each input place contains at least one token.
2. An enabled transition may or may not fire.
3. Firing an enabled transition $t$ means removing one token from each input place of $t$ and adding one token to each output place of $t$.

The firing of a transition has zero duration.

The firing of a sink transition (only input places) only consumes tokens.

The firing of a source transition (only output places) only produces tokens.

Typical interpretations of places and transitions:

<table>
<thead>
<tr>
<th>Input Places</th>
<th>Transition</th>
<th>Output Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>Event</td>
<td>Postconditions</td>
</tr>
<tr>
<td>Input data</td>
<td>Computation step</td>
<td>Output data</td>
</tr>
<tr>
<td>Input signals</td>
<td>Signal processor</td>
<td>Output signals</td>
</tr>
<tr>
<td>Resources needed</td>
<td>Task or job</td>
<td>Resources needed</td>
</tr>
<tr>
<td>Conditions</td>
<td>Clause in logic</td>
<td>Conclusions</td>
</tr>
<tr>
<td>Buffers</td>
<td>Processor</td>
<td>Buffers</td>
</tr>
</tbody>
</table>

Generalized Petri Nets

Firing rules:

1. A transition $t$ is enabled if each input place $p$ of $t$ contains at least $w(p,t)$ tokens.
2. Firing a transition $t$ means removing $w(p,t)$ tokens from each input place $p$ and adding $w(t,q)$ tokens to each output place $q$.

Petri Net Variants

Timed Petri Nets:
Times associated with transitions or places

High-Level Petri Nets:
Tokens are structured data types (objects)

Continuous & Hybrid Petri Nets:
The markings are real numbers instead of integers

Mixed continuous/discrete systems
Analysis

Properties:
- **Live**: No transitions can become unfirable.
- **Deadlock-free**: Transitions can always be fired.
- **Bounded**: Finite number of tokens.
- ...

Analysis

Analysis methods:
- **Reachability methods**
  - exhaustive enumeration of all possible markings
- **Linear algebra methods**
  - describe the dynamic behaviour as matrix equations
- **Reduction methods**
  - transformation rules that reduce the net to a simpler net while preserving the properties of interest

The classical real-time problems

Dijkstra's classical problems
- mutual exclusion problem
- producer-consumer problem
- readers-writers problem
- dining philosophers problem

All can be modeled by Petri Nets.

Mutual Exclusion

Process A
- Waiting for critical section
- Executing inside critical section
- Executing outside critical section

Process B
- Waiting for critical section
- Executing inside critical section
- Executing outside critical section

Producer-Consumer

Unbounded buffer:

Bounded buffer:

Producer processes
- Buffer
- Write

Consumer processes
- Read

Producer processes
- Buffer
- First places
- Write

Consumer processes
- Read
- Final places
Readers-Writers

Writers processes

Readers processes

Access Control

Dining Philosophers

Coding State Machines

- Discrete Event Systems
- State Machine-Based Formalisms
- Statecharts
- Grafcet
- Laboratory 2
- Petri Nets
- Implementation
  - Not covered in the lecture. Homework.

Using state machines is often a good way to structure code. Systematic ways to write automata code often not taught in programming courses.

Issues:
- active or passive object
- Mealy vs Moore machines
- states with timeout events
- states with periodic activities

Often convenient to implement state machines as periodic processes with a period that is determined by the shortest time required when making a state transition.
Example: Passive state machine
The state machine is implemented as a synchronized object

```java
class PassiveMealyMachine {
    private static final int STATE0 = 0;
    private static final int STATE1 = 1;
    private static final int STATE2 = 2;
    private int state;
    PassiveMealyMachine() {
        state = STATE0;
    }
    private void generateEvent(int outEvent) {
        // Do something
    }
    public synchronized void inputEvent(int event) {
        switch (state) {
            case STATE0 : switch (event) {
                case INA : generateEvent(OUTA);
                            state = STATE1;
                            break;
                case INB : generateEvent(OUTB);
                            break;
                default : break;
            }; break;
            case STATE1 : switch (event) {
                case INC : generateEvent(OUTC);
                            state = STATE2;
                            break;
                default : break;
            }; break;
            case STATE2 : switch (event) {
                case INA : generateEvent(OUTB);
                            state = STATE0;
                            break;
                case INC : generateEvent(OUTC);
                            break;
                default : break;
            }; break;
        }
    }
}
```

Active State Machines
The state machine could also be implemented as an active object (thread)

The thread object would typically contain an event-buffer (e.g., an RTEventBuffer).
The run method would consist of an infinite loop that waits for an incoming event (RTEvent) and switches state depending on the event.

```java
class ActiveMachine1 extends Thread {
    private static final int STATE0 = 0;
    private static final int STATE1 = 1;
    private int state;
    ActiveMachine1() {
        state = STATE0;
    }
    private boolean cond0() {
        // Returns true if condition 0 is true
    }
    public void run() {
        long t = System.currentTimeMillis();
        long duration;
        while (true) {
            switch (state) {
                case STATE0 : {
                    action0();
                    t = t + 20;
                    duration = t - System.currentTimeMillis();
                    if (duration > 0) {
                        try {
                            sleep(duration);
                        } catch (InterruptedException e) {
                            
                    }
                    if (cond0()) {state = STATE1;}
                    break;
                case STATE1 : {
                    // Similar as for STATE0. Executes action1, waits for 50 ms, checks cond1 and then changes to STATE2
                    break;
                case STATE2 : {
                    // Similar as for STATE0. Executes action2, waits for 10 ms, checks cond2 and then changes to STATE0
                    break;
                }
            }
        }
    }
}
```
Comments

- Conditions tested at a frequency determined by the activity frequencies of the different states.
- `sleep()` spread out in the code

Example: Active state machine 2

The thread runs at a constant (high) base frequency. Activity frequencies multiples of the base frequency. Conditions tested at the base frequency.

```java
public void run() {
    long t = System.currentTimeMillis();
    long duration;
    int counter = 0;
    while (true) {
        counter++;
        switch (state) {
            case STATE0 : {
                if (counter == 4) { counter = 0; action0(); }
                if (cond0()) { counter = 0; state = STATE1; }
            } break;
            case STATE1 : {
                // Similar as for STATE0. Executes action1 if counter == 10. Changes to STATE2 if cond;
                break;
            } case STATE2 : {
                // Similar as for STATE0. Executes action2 if counter == 12. Changes to STATE0 if cond;
            } break;
        }
        t = t + 5; // Base sampling time
        duration = t - System.currentTimeMillis();
        if (duration > 0) {
            try {
                sleep(duration);
            } catch (InterruptedException e) {} } 
        }
    }
}
```

Comments

- Polled time handling
- Complicated handling of counter
- Conditions tested at a high rate