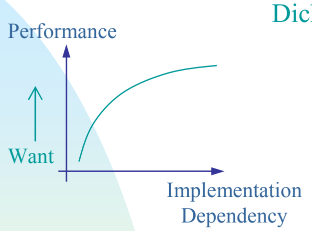
System Synthesis

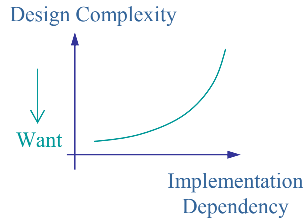
郭大維 教授
ktw@csie.ntu.edu.tw
即時暨嵌入式系統實驗室
(Real-Time and Embedded Systems Laboratory)
國立臺灣大學資訊工程系

- Paper for discussion:
  Aloysius K. Mok, "A Graph-Based Computation Model for Real-time Systems," IEEE Proceedings of The International Conference on Parallel Processing, 1985.
- Major Reference:
  Aloysius K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment," Ph.D. Thesis, MIT, 1983.

The Problem:

The Efficiency vs. Maintainability Dichotomy



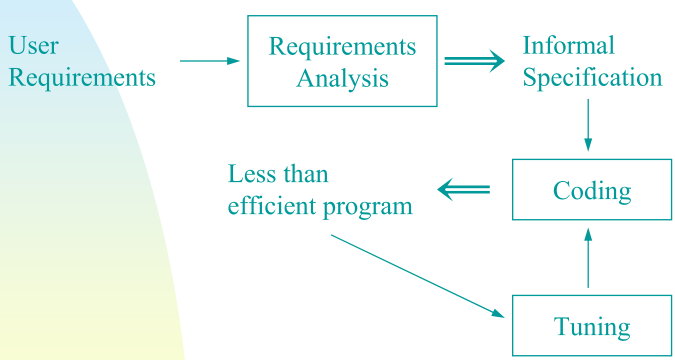


- Highly optimized code is hard to read. It often involves too many coding tricks!!
- Real systems are often compromise between structured design and efficiency hacks.
- But, compromise may not be possible for many time-critical systems.

Is there a way out of this dilemma ?

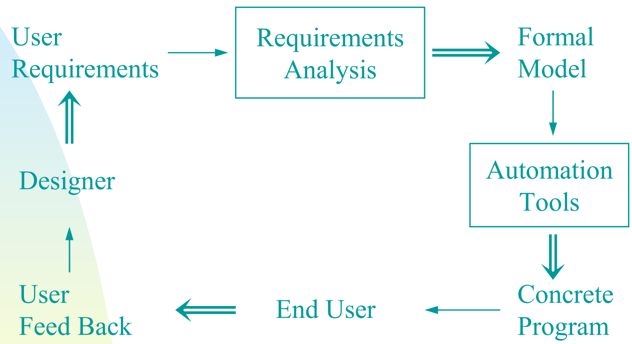
Software Technology Paradigms

- Current Practice



Software Technology Paradigms (cont.)

- The way to Go ?



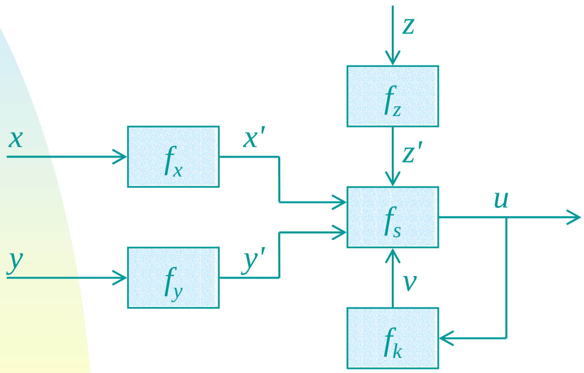
How do we get from here to there ??

A Software Automation Strategy:

- Capture the computational requirements of the application domain in terms of an appropriate model.
- Translate requirements specifications into an instance of the domain-specific model for resource allocation analysis.
- Solve the well-defined optimization problems to minimize chosen cost/risk criteria.

An example:

A control system function block diagram



An example (cont.)

- System Requirements
  - Sample x at rate $1/P_x$ per sec update u. Then update v with the new value of u.
  - Sample y at rate $1/P_y$ per sec update u. Then recompute v with the new value of u.
  - When z changes state, update u within d_z sec. The output signal u must also be recomputed before d_z.
- Let's try some parameters:

$P_x = 80$ $d_z = 80$

$P_y = 160$ $d_y = 160$

$c_x = c_y = c_z = c_s = c_k = 10$

Problem:

Translate user requirements into a set of processes ?!!

But,

English is too informal !

We need a more precise language which must also be natural to the application domain !

A Graph-Based Model M

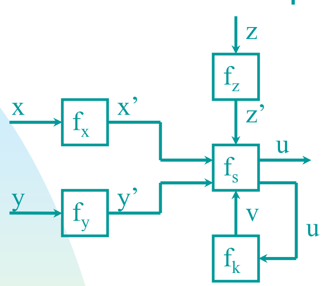
M = (G, T)

- G is the communication graph.
  (A digraph with vertex and edge weights)
- $T = T_P + T_A$ is a set of timing constraints. A timing constraint is a tuple (C, r, d, p)
  - C is a task graph that must be compatible with G.
    $\exists$ a mapping h s.t.
    $$\forall v,\ \ v \in C \rightarrow h(v) \in G$$
    $$\forall u, v,\ \ (u, v) \in C \rightarrow (h(u), h(v)) \in G$$
  - r is the ready time
  - d is the deadline
  - p is the period/minimum separation
- The "computation time" of a timing constraint (C, r, d, p) is the sum of the weights of the vertices of C.

$T = T_P + T_A$

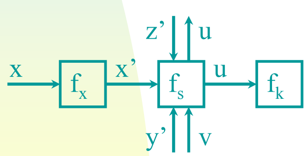
- T_P is the set of periodic timing constraint.
  - Periodic timing constraints are invoked at fixed intervals:
  - kp + phasing, k=0,1,…
- T_A is the set of asynchronous timing constraint.
  - Asynchronous timing constraints are invoked at arbitrary times,but two successive invocations must be separated by p time units.
- A task graph C is said to be executed in the interval $[t_1, t_2]$ if there is a multiset of functional element (vertices) executions in $[t_1, t_2]$ which is consistent with the partial ordering C.
  - In a distributed environment, edges in C denote transmission of information from one functional element to another.
- aIf a timing constraint is invoked at time t, it must be executed in $[t + r,\ t + d]$.

- Communication Graph

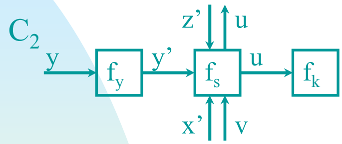


- Timing Constraints

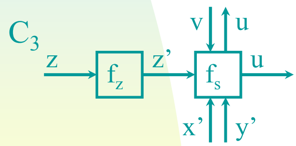
C_1



type = periodic
period = 80
deadline = 80



type = periodic
period = 160
deadline = 160



type = asynchronous
period = default
(or p_z if known)
deadline = 80

- Our job:
  Given a graph-based specification of a real-time system, output a set of processes (programs).
- A process-based language (programming model)
  - Process declaration:
    Process <Name>
    activated by (<signal>|Timer)
    <Body>
    End
  - Synchronization (precedence) constraints are enforced by:
    Rendezvous <Process>
  - Mutual Exclusion constraints are enforced by:
    Rendezvous <monitor>
  - A monitor is declared by:
    Monitor <Name>
    <Body>
    End

Decomposition Strategies

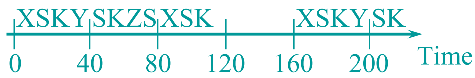
Decomposition By Critical Timing Constraints (CTC)

- Use a process for each timing constraint.
  - Process XSK
    activated by timer;
    attribute period=80, deadline=80;
    x = sensor_x();
    x' = f_x(x);
    rendezvous S;
    redezvous K;
    end XSK
  - Process YSK
    activated by timer;
    attribute period=160, deadline=160;
    y = sensor_y();
    y' = f_y(y);
    rendezvous S;
    rendezvous K;
    end YSK

(cont.)

(cont.)

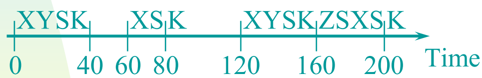
- Process ZS
  activated by z;
  attribute deadline=80, period=default;
  z = sensor_z();
  z' = f_z(y);
  rendezvous S;
  end ZS
- monitor S
  u = f_s(x', y', z', v);
  end S
- monitor K
  v = f_k(x', y', z', v);
  end K



Strength:

Straightforward: easy to understand.
Maintainability is high!

However, the unnecessary duplication of some computation is serious.



Throwing away duplicates may make the sampling of x and y at a higher rate!

px = 60, py = 120

(old px = 80, py = 160)

Decomposition By Centralizing Concurrency Control (CCC) on Minimizing Interprocess Communication

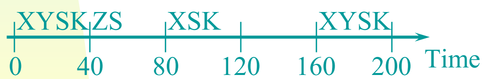
1. Partition the computation required by the timing constraints into sets such that
   - (i) Only compatible timing constraints are assigned to the same set, and
   - (ii) Only timing constraints that share some of the function calls are assigned to the same set
2. The computation in each set is assigned to a periodic process whose period attribute is set to the GCD of the periods in the set.

   * Each asynchronous timing constraint is assigned to a sporadic process as before.

✻Pre-period deadlines need priori analysis or…

Merge XSK and YSK

Process XYSK
  activated by timer;
  attribute period=80, deadline=80;
  x = sensor_x();
  x' = f_x(x);
  if skip_y() == FALSE then { y = sensor_y();
    y' = f_y(y); }
  rendevous S;
  v = f_k(u);
end XYSK



- Strength:
  - Efficiency is improved by eliminating substantial redundant computation!
  - With fewer processes and more independent process, less inter process communication may be required!

  However, maintainability becomes more difficult!
- Suppose C_y = 40ms
  - Use two-stage pipeline implementation!

It works!

However, the control logic adopted in XYSK implements internal scheduling decisions and make itself very sensitive to system parameters, e.g. "workload". Maintainability becomes nightmares for programmers.

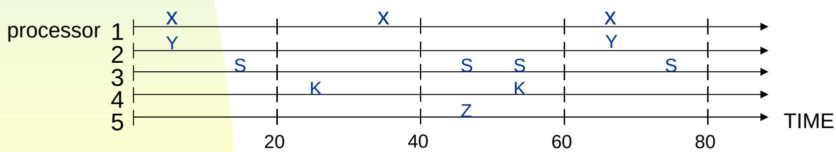
Decomposition By Distribution Concurrency Control (DCC) or Maximizing Concurrent Process

Partition the required computation into as many process as possible so as to maximize parallelism !

⁎In general, if a node is involved in the computation required by one or more periodic timing constraints, the process assigned to the node has a period equal to GCD of periods of relevant timing constraint !

⁎Each asynchronous timing constraint is assigned a sporadic process which contains appropriate function calls.

=> Periodic processes must synchronize with processes which precede it and which it precedes !



X and Y can be even sampled at rates 30 and 60, respectively!!

Comparison of Decomposition strategies

	By Timing constraint	By minimizing communication	By Maximizing Parallelism
Processor Speed Requirement	Higher	1*	Lower
Communication Bandwidth Requirement		Lower	Higher
Ease of Understanding	Good	Poor	
Ease of Modification		Poor	2*

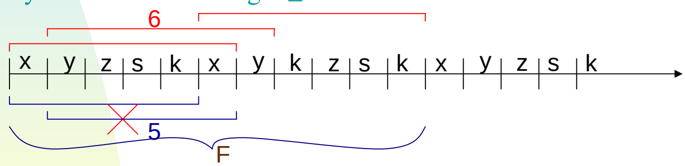
1* Less locking problems, more efficient utilization of processor power.

2* Additional timing constraint may not involve any change in program, but it may require more difficult analysis !

Another way to meet timing constraints:

- Latency Scheduling
  - An execution trace of a processor is a mapping F from the non-negative integers to the set of nodes in a communication graph G plus a null symbol ϕ such that

    F(i) = u if u is executed in the time internal [i, i+1]
  - An execution trace F have a latency of K time units with respect to a timing constraint (c, p, d) iff F contains an execution of C in any time interval of length $\geq$ K.



  - A static schedule L has a latency of K time units with respect to the timing constraint (c, p, d) iff the execution trace which a " round-robin " scheduler generates by repeating L ad infinitum has a latency of K time units with respect to (c, p, d)

- A static schedule L is feasible with respect to a set of asynchronous timing constraints T_a iff L has a latency of d time units with respect to every timing constraint (c, p, d) $\in T_a$

Theorem [Mok 85] If there is an execution trace which has latency d with respect to every asynchronous timing constraint in a graph-based model (G,T), then there must be a feasible static schedule (finite by definition) with respect to $T_a \in$ T .

Theorem [Mok 85] The problem of determining whether a feasible static schedule exists for a graph-based model (G,T) is NP-hard in the strong sense for the following two restricted cases:

(i) All the functional elements in G have unit computation time and all the task graphs in T are chains of length 1 or 3.

(ii) Every task graph in T consists of a single operation; all but one of the deadlines are the same and the functional elements cannot be pipelined into chains of subfunctions.

Theorem [Mok 85] Let w_i , d_i be the computation time and deadline of the *i*th timing constraint. If (i) $\sum \frac{w_i}{d_i} \le \frac{1}{2}$; and (ii) $\frac{d_i}{2} \ge w_i$; and (iii) all the functional elements can be pipelined, then a feasible static schedule always exists.

- Cluster all timing constraints into a single periodic process:

```
Porcess  XYZSK
        activated by timer;
        attribute period = 50, deadline = 50;
        x = sensor_x();
        x' = f_x(x)
        If skip_Y() = FALSE THEN  { y=sensor_y(); y' =f_y(y)}
        If skip_Z()  = FALSE THEN  { z=sensor_z(); z' =f_z(z)}
        u = f_x (x', y', z', v);
        v = f_k(u);
end XYZSK
```

X can be sampled at a rate 1/50 cycles/ms !

There is no best decomposition algorithm for all architectures ! We still have to tune !

More fundamental problem with process-based models:

- A process serves conflicting goals.
  - As a unit for processor scheduling.
  - As a unit to enforce integrity constraints.
  - As a unit to organize computation to meet a goal.

  A good decomposition strategy must consider all three goals !

- Process models, being abstractions of Von Neuman type machines may be an artificial architectural constraint!