EECS 507: Introduction to Embedded Systems Research
Specification

Robert Dick

University of Michigan

Topics

Number	Topic
30	Internet-of-things and embedded sensing systems
22	Wearable and medical devices
22	Special-purpose and configurable architectures
19	Embedded machine learning
18	Automotive and autonomous vehicles
16	Low-power, temperature-aware, and reliability-aware design
15	Embedded security and privacy
9	Embedded and real-time operating systems
9	Wireless communication
7	Robotics and controls
5	Augmented and virtual reality
4	Computer vision
3	Human/computer interaction and interfaces
2 each	Signal processing, Firmware
1 each	High-level synthesis, Aerospace, Embedded electrical architecture, Localization, Integrated Microsystems, Environmental monitoring, Brain-computer interface, Sustainability, Software-defined radios

May do more on wearables than initially planned.

Outline

Why learn about specification languages

Understand different ways of describing design to other people.

Find most appropriate language for your application.

Use for synthesis.

Use for model checking.

Separation of requirements and implementation decisions

Costs and constraints, including functionality constraints and available resources define the problem.

The system-level architecture is the solution.

Coupling these undermines optimization

- Presupposes that particular architectural solutions are good, without evaluating them.
- Prevents evaluation of costs in the context of the requirements, alone.

Available resources – system-level

General-purpose SW processors

Digital signal processors (DSPs)

Application-specific integrated circuits

Dynamically reconfigurable hardware

E.g., field-programmable gate arrays (FPGAs)

Busses

Wireless communication channels

Wires

Available resources – high-level

Simple arithmetic/logic units.

Multiplexers.

Registers.

Wires.

Specification language requirements

Describe hardware (HW) and software (SW) requirements.

Specify constraints on design.

Indicate system-level building blocks.

To allow flexibility in synthesis, must be abstract.

- Differentiate HW from SW only when necessary.
- Concentrate on requirements, not implementation.
- Make few assumptions about platform.

Section outline

Software oriented design representations

ANSI-C.

SystemC.

Other SW language-based.

ANSI-C

Advantages.

- Huge code base.
- Many experienced programmers.
- Efficient means of SW implementation.
- Good compilers for many SW processors.

Disadvantages.

- Little implementation flexibility.
  - Strongly SW oriented.
  - Makes many assumptions about platform.
- Poor support for fine-scale HW synchronization.

SystemC.

Advantages.

- Support from big players.
- Synopsys, Cadence, ARM, Red Hat, Ericsson, Fujitsu, Infineon Technologies AG, Sony Corp., STMicroelectronics, and Texas Instruments.
- Familiar for SW engineers.

Disadvantages.

- Extension of SW language.
- Not designed for HW from the start.

Many SW language-based

Numerous competitors.

ANSI-C, C++, and Java are most popular starting points.

In the end, few can be widely used.

SystemC has broad support.

Section outline

Hardware oriented design representations

VHDL.

Verilog.

Esterel.

VHDL

Advantages.

- Supports abstract data types.
- System-level modeling supported.
- Better support for test harness design.

Disadvantages.

- Requires extensions to easily operate at the gate-level
- Difficult to learn
- Slow to code

Verilog

Advantages.

- Easy to learn.
- Easy for small designs.

Disadvantages.

- Not designed to handle large designs.
- Not designed for system-level.

Verilog vs. VHDL

March 1995, Synopsys Users Group meeting.

Create a gate netlist for the fastest fully synchronous loadable 9-bit increment-by-3 decrement-by-5 up/down counter that generated even parity, carry and borrow.

5 / 9 Verilog users completed.

0 / 5 VHDL users competed.

Does this mean that Verilog is better? Maybe, but maybe it only means that

Verilog is easier to use for simple designs.

Esterel

Easily allows synchronization among parallel tasks.

Works above RTL.

Doesn't require explicit enumeration of all states and transitions.

Recently extended for specifying datapaths and flexible clocking schemes.

Amenable to theorem proving.

Translation to RTL or C possible.

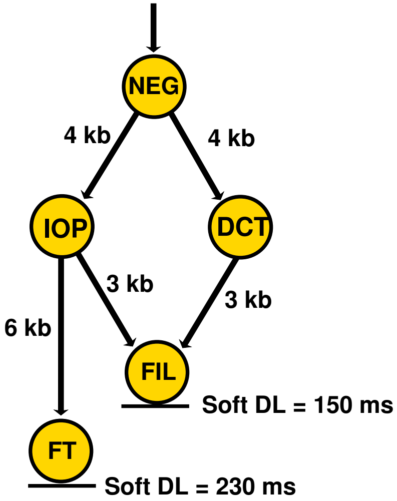
Commercialized by Esterel Technologies.

Section outline

Graph based design representations

- Dataflow graph (DFG).
- Synchronous dataflow graph (SDFG).
- Control flow graph (CFG).
- Control dataflow graph (CDFG).
- Finite state machine (FSM).
- Petri net.
- Periodic vs. aperiodic.
- Real-time vs. best effort.
- Discrete vs. continuous timing.
- Example from research.

Dataflow graph (DFG)



Nodes are tasks.

Edges are data dependencies.

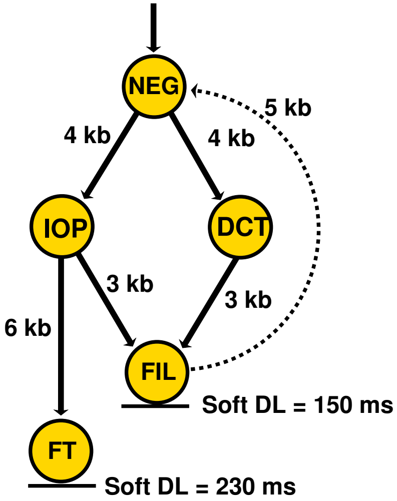
Edges have communication quantities.

Used for digital signal processing (DSP).

Often acyclic when real-time.

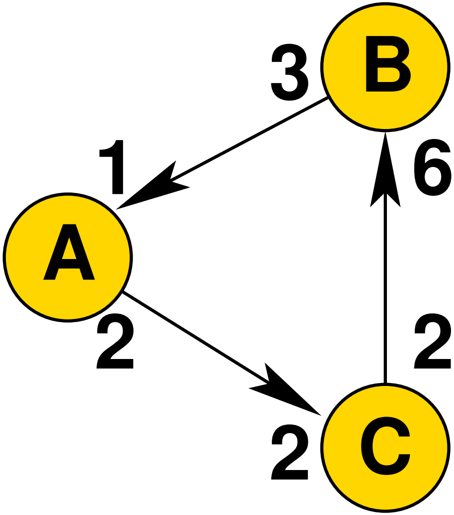
Can be cyclic when best-effort.

Dataflow graph (DFG)

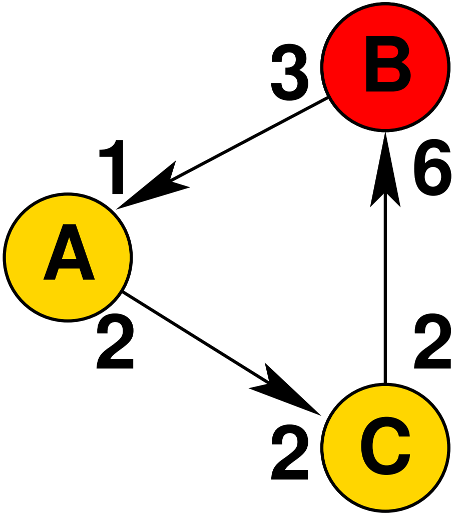


- Nodes are tasks.
- Edges are data dependencies.
- Edges have communication quantities.
- Used for digital signal processing (DSP).
- Often acyclic when real-time.
- Can be cyclic when best-effort.

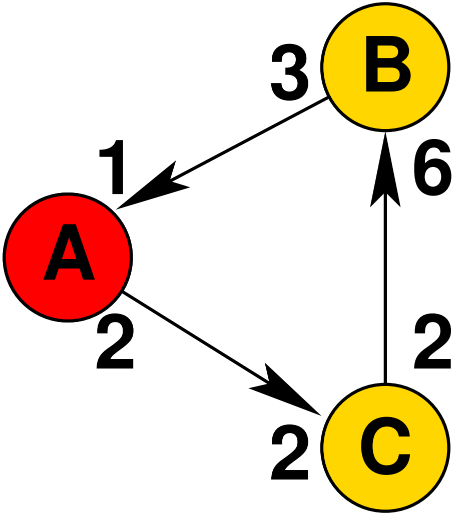
Synchronous dataflow graph (SDFG)



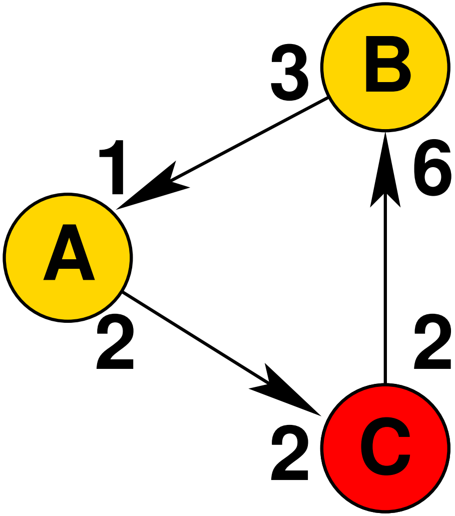
Synchronous dataflow graph (SDFG)



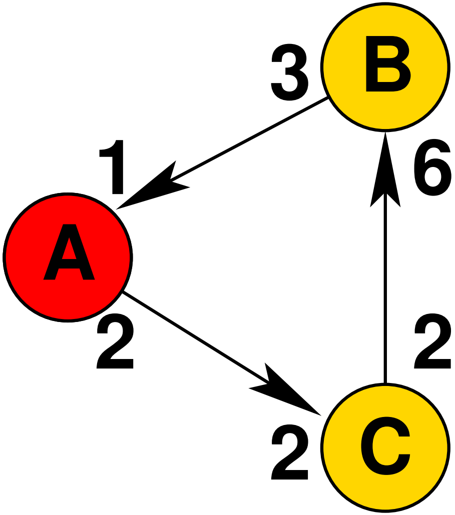
Synchronous dataflow graph (SDFG)



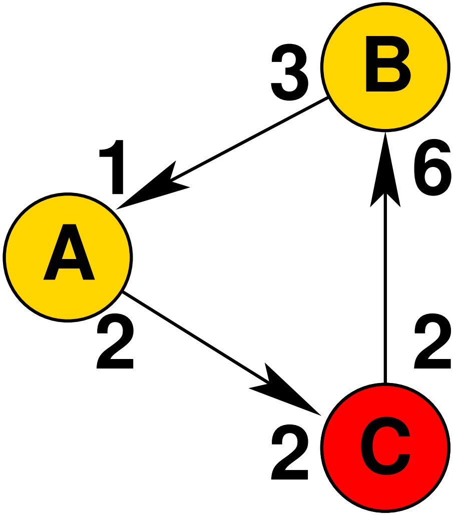
Synchronous dataflow graph (SDFG)



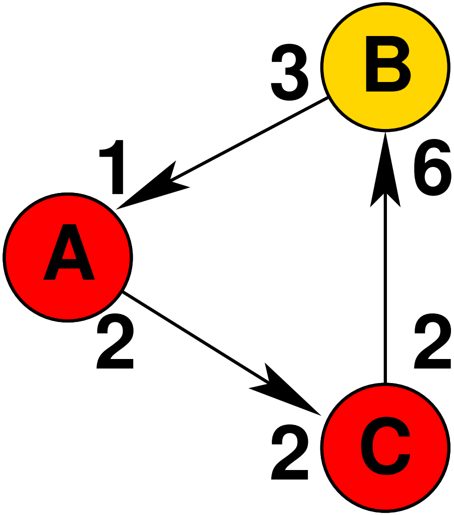
Synchronous dataflow graph (SDFG)



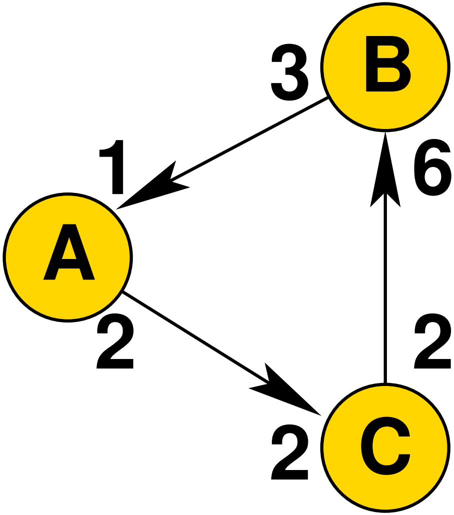
Synchronous dataflow graph (SDFG)



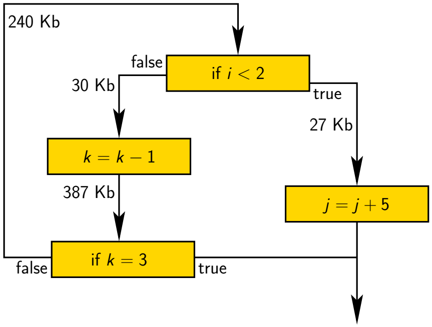
Synchronous dataflow graph (SDFG)



Synchronous dataflow graph (SDFG)



Control flow graph (CFG)



Nodes are tasks.

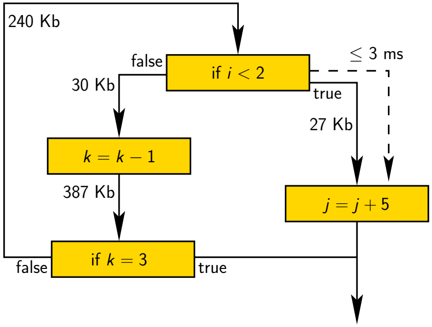
Supports conditionals, loops.

No communication quantities.

SW background.

Often cyclic.

Control dataflow graph (CDFG)

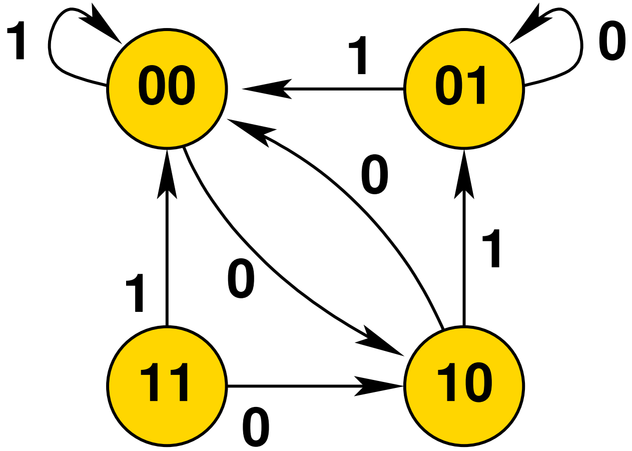


Supports conditionals, loops.

Supports communication quantities.

Used by some high-level synthesis algorithms.

Finite state machine (FSM)


$$\begin{tikzcd}
00 \arrow[loop left, "1"] \arrow[dr, bend right, "0"'] & 01 \arrow[l, "1"'] \arrow[loop right, "0"] \\
11 \arrow[u, "1"] \arrow[r, "0"'] & 10 \arrow[ul, bend right, "0"'] \arrow[u, "1"']
\end{tikzcd}$$

Finite state machine (FSM)

	input	
	0	1
00	10	00
01	01	00
10	00	01
11	10	00
current	next	

Normally used at lower levels.

Difficult to represent independent behavior.

State explosion.

No built-in representation for data flow.

Extensions have been proposed.

Extensions represent SW, e.g., co-design finite state machines (CFSMs).

Petri net

Graph composed of places, transitions, and arcs.

Tokens are produced and consumed.

Useful model for asynchronous and stochastic processes.

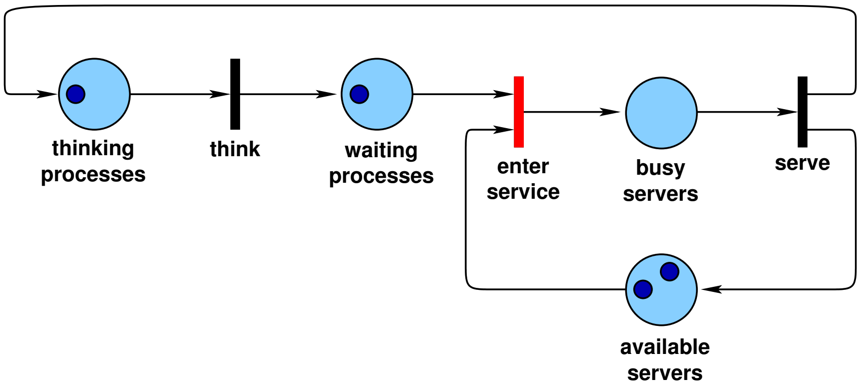
Places can have priorities.

Not well-suited for representing dataflow systems.

Timing analysis quite difficult.

Large flat graphs difficult to understand.

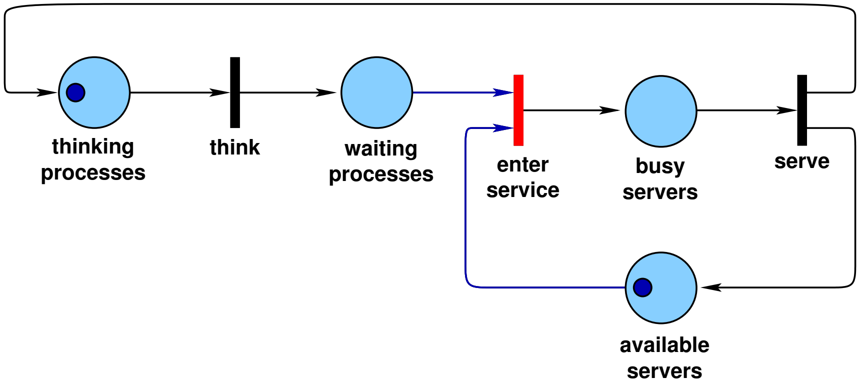
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

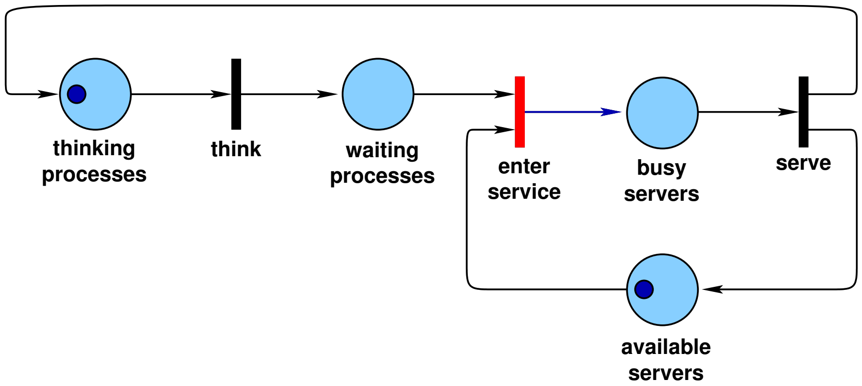
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

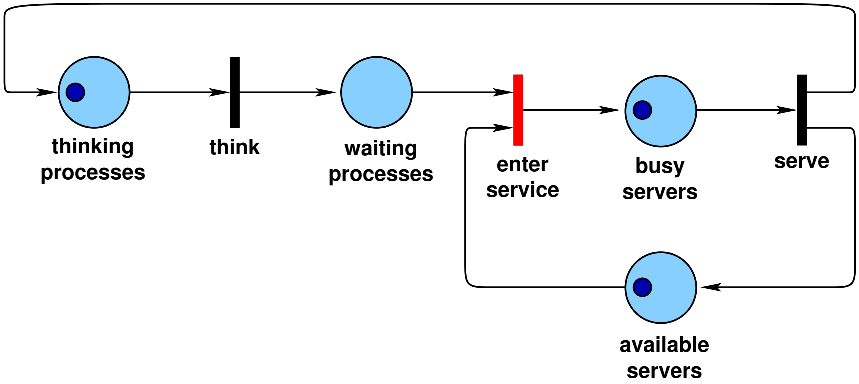
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

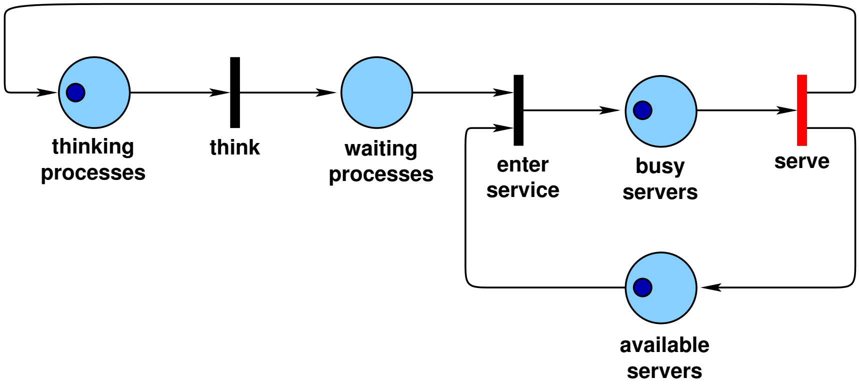
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

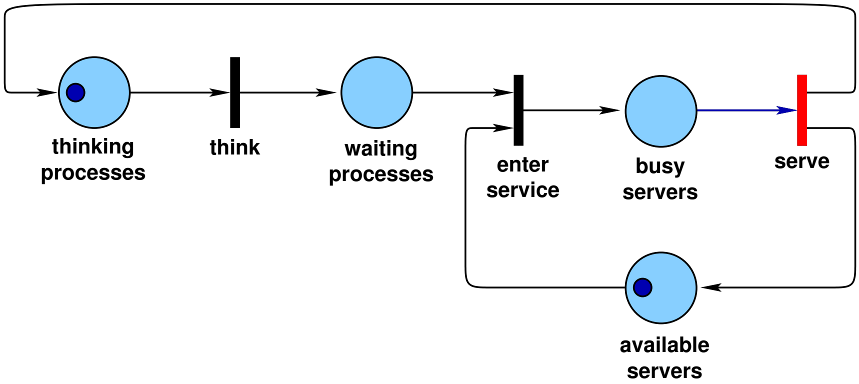
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

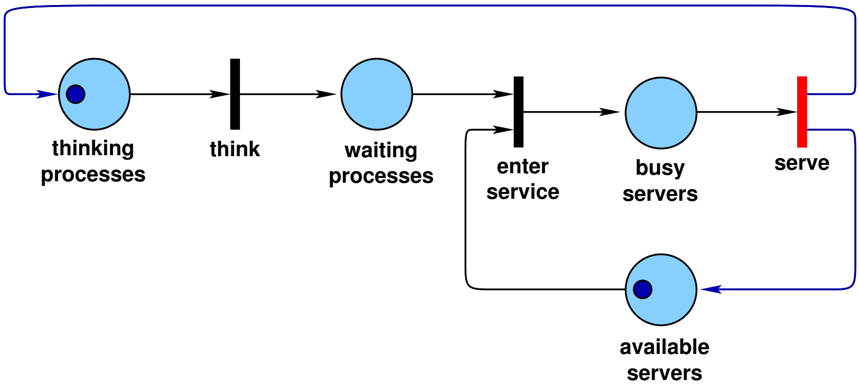
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

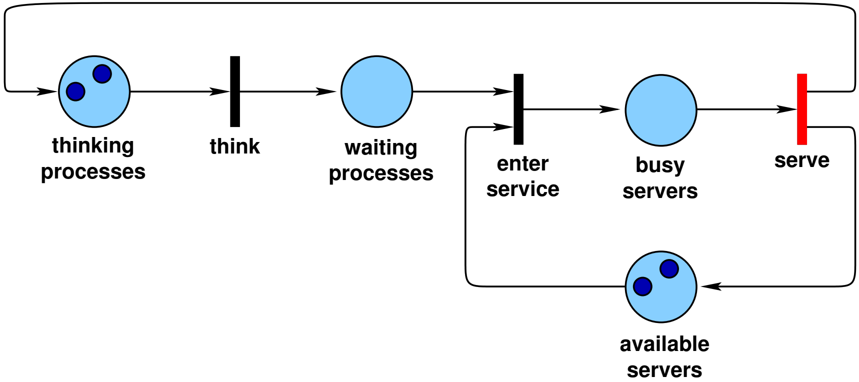
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

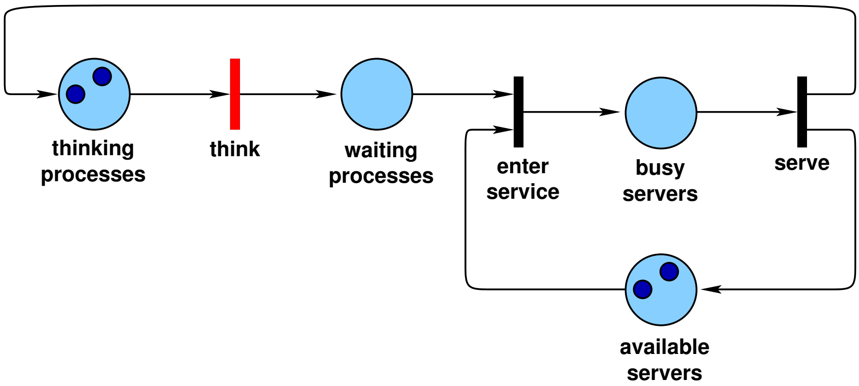
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

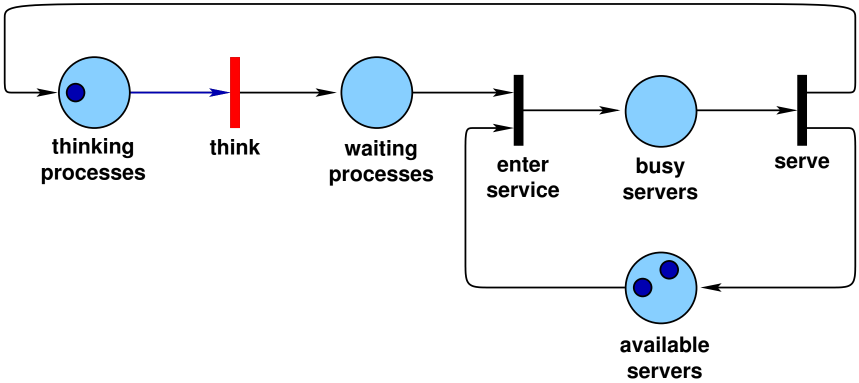
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

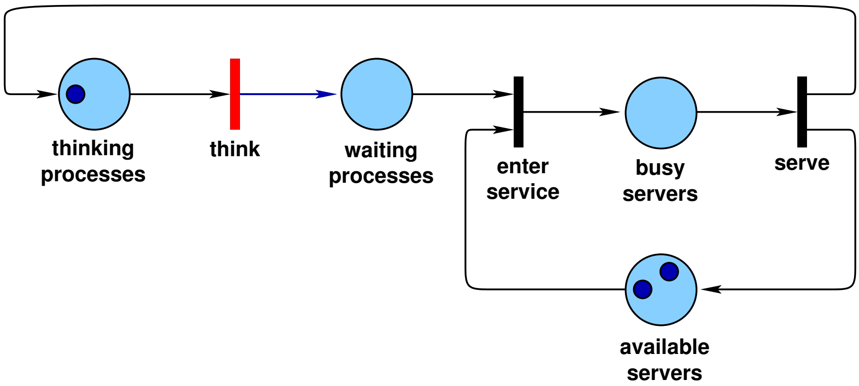
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

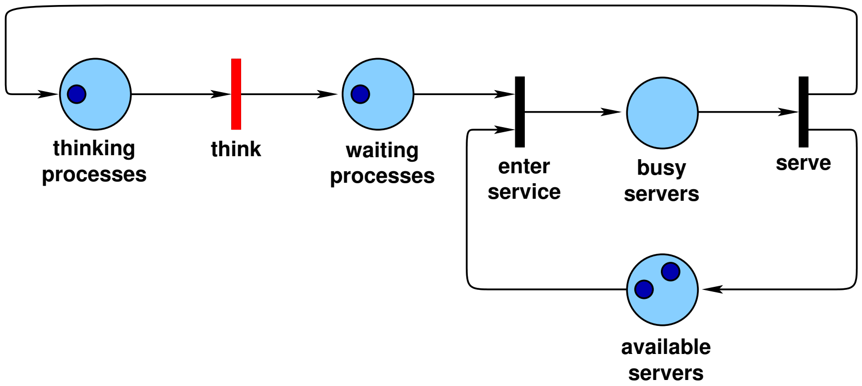
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

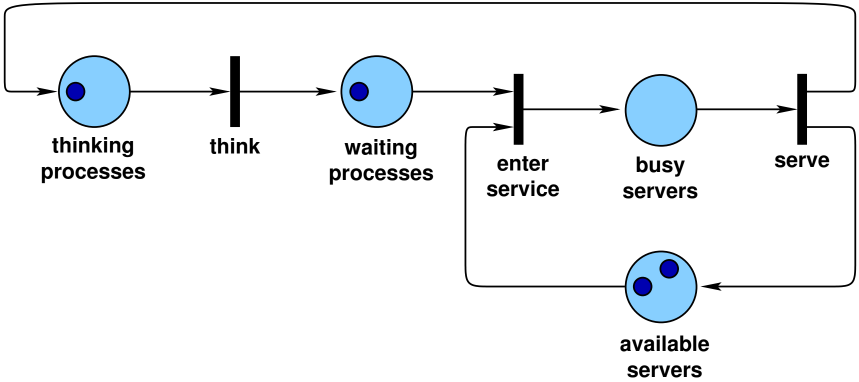
Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

Petri net



M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann's token game demonstration.

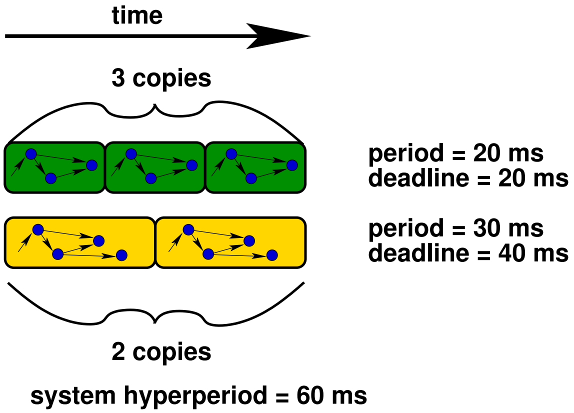
Periodic graphs

Some system specifications contain periodic graphs.

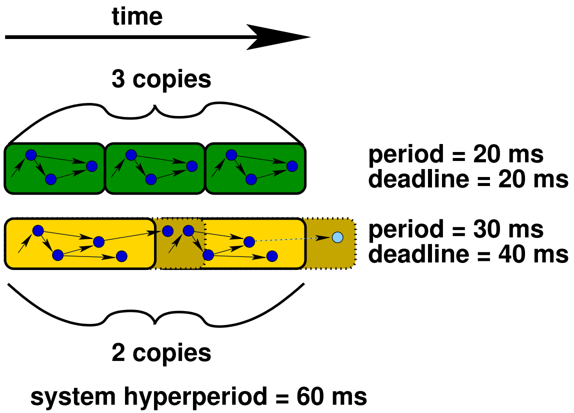
Can guarantee scheduling validity by scheduling to the least common multiple of periods.

Can also meet aperiodic specifications, however, resources will sometimes be idle.

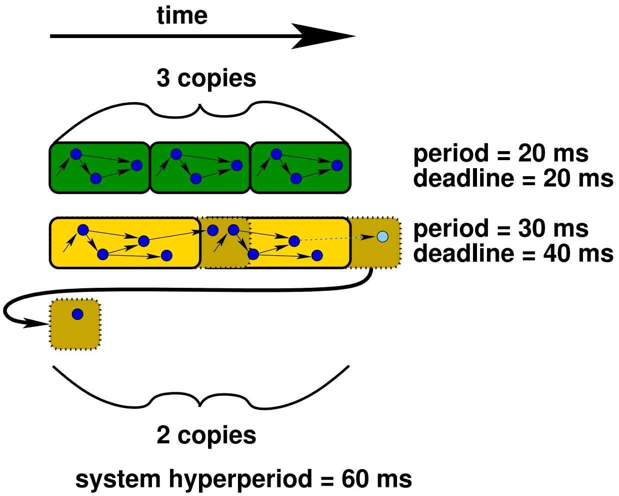
Periodic graphs



Periodic graphs



Periodic graphs



Aperiodic graphs

No precise periods imposed on task execution.

Useful for representing reactive systems.

Difficult to guarantee hard deadlines in such systems.

Possible if minimum inter-arrival time known.

Periodic vs. aperiodic

Periodic applications.

- Power electronics.
- Transportation applications.

Engine controllers.

Brake controllers.

Many multimedia applications.

- Video frame rate.
- Audio sample rate.

Many digital signal processing (DSP) applications.

However, devices which react to unpredictable external stimuli have aperiodic behavior

Many applications contain periodic and aperiodic components

Aperiodic to periodic

Can design periodic specifications that meet requirements posed by aperiodic specifications.

Some resources will be wasted.

Example

- At most one aperiodic task can arrive every 50 ms.
- It must complete execution within 100 ms of its arrival time.

Aperiodic to periodic

Can easily build a periodic representation with a deadline and period of 50 ms.

Problem, requires a 50 ms execution time when 100 ms should be sufficient.

Can use overlapping graphs to allow an increase in execution time.

Parallelism required.

The main problem with representing aperiodic problems with periodic representations is that the tradeoff between deadline and period must be made at the time of synthesis.

Real-time vs. best effort

Why make decisions about system implementation statically?

Allows easy timing analysis, hard real-time guarantees.

If a system doesn't have hard real-time deadlines, resources can be more efficiently used by making late, dynamic decisions.

Can combine real-time and best-effort portions within the same specification.

- Reserve time slots
- Take advantage of slack when tasks complete sooner than their worst-case finish times

Discrete vs. continuous timing

System-level: continuous.

Operations are not small integer multiples of the clock cycle.

High-level: discrete.

Operations are small integer multiples of the clock cycle.

Implications

- System-level scheduling is more complicated. . .
- . . . however, high-level also very difficult.

Section outline

Processing resource description

Often table-based.

Price, area.

For each task.

- Execution time
- Power consumption
- Preemption cost
- etc.

Similar characterization for communication resources.
Wise to use process-based.

Communication resource description

Can use bus-bridge based models for distributed systems.

Wireless models.

However, in the future, it will become increasingly important to base SOC communication model on process parameters.

System-level representations summary

No single representation has been decided upon.

Software-based representations becoming more popular.

System-level representations will become more important.

This is still an active area of research.

Notes on clustering and partitioning

Interdependence with architecture.

Heterogeneity's impact on partitioning.

Applications to grid computing.

Dynamic partitioning.

Some future direction

Can specification be so simple for some embedded application domains that application experts who are not computer engineers easily do it?

What HCI, compiler, and synthesis support is required?

What impact will increasing use of machine learning in embedded systems have on specification?

Outline

Jantsch and Sander '05 I

A. Jantsch and I. Sander, "Models of computation and languages for embedded system design," *IEEE Proc.*, pp. 114–129, 2005.

The authors seek to explain valuable characteristics of embedded system models and specification languages, and relate these to real-world modeling approaches and languages.

Starts from implementation components: not ideal.

Back on right track in S 1.1: start from (functionality) requirements.

States that current design practice is to settle on HW/SW interface too early. Actually worse than that. Often settle on solution before defining problem.

HW/SW divide explained.

Jantsch and Sander '05 II

Nice examples of perils of defining detailed interface too early; it is challenging to write these examples well, and they improve the paper.

"A formal model of design should consist of the following components: (1) a functional specification..., (ii) a set of [required] properties..., (iii) a set of performance indexes [costs]..., and (iv) a set of constraints on [costs]."

Recommends separating computation–communication, function–architecture.

Handling time causes many problems. Safe and general-purpose approaches are very expensive.

More person-hours go into verification than design in complex projects.

Often refers to formal verification without defining it: define now.

Jantsch and Sander '05 III

Claims that C and VHDL do not have mathematically defined semantics; they do in some cases, but they are often ambiguous.

Perils of specifying low-level implementations too early; but doing otherwise is hard if costs and constraints matter. Hard to know performance of primatives without implementing them.

They establish a hierarchy of degrees of model explicitness; I think others might structure it differently.

Continuous-time models can be expensive to evaluate. Generally diffeq-based simulation used. Use sparingly.

Synchronous models are very popular. However, they bias toward synchronous implementations. Some have proposed ordered synchronous models.

Jantsch and Sander '05 IV

Zero-time feedback loops can lead to non-determinism and oscillation. Can prevent with a variety of restrictions.

"Analogue synthesis is still in its infancy..." It is a gray-haired infant, then.

VHDL: Designed for simulation, but synchronous subset can be synthesized.

Different temporal models can be used at different times during simulation, e.g., SimOS.

Commonly, more constrainted models make verification more practical.

Implicit infinite buffers in dataflow models are often unrealistic.

Established languages can be used as foundations for more restrictive languages/models enabling synthesis and formal verification.

Jantsch and Sander '05 V

They may overestimate difficult of new languages replacing old. It does happen, but rarely.

Outline

Assignments

6 Sep: project team formation, 5pm–7pm in 1050 FMCRB.

13 Sep: L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar, "High-performance operating system controlled on-line memory compression," *ACM Trans. Embedded Computing Systems*, vol. 9, no. 4, pp. 30:1–30:28, Mar. 2010 summary.

15 Sep: R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proc. Int. Wkshp. Hardware/Software Co-Design*, May 2002, pp. 73–78 summary.