ECE382N.23: Embedded System Design and Modeling

Lecture 2 – Application Models

Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu

Lecture 2: Outline

- **System specification**
  - Specification modeling
  - Formal models
- **Models of Computation (MoCs)**
  - Concurrency & communication
  - Process- and dataflow-based models
  - State-based models

System Specification

- **"Golden" reference model**
  - First model/input description in the design flow
  - All implementations derived from and compared to this one
  - System-level "programming" model
- **High abstraction level**
  - No premature HW/SW implementation assumptions
  - Unrestricted exploration of design space
- **Formal for analysis & synthesis**
  - Verification & implementation (system "compilation")

➢ **Executable**
  - Simulate for functional validation

➢ **Pure behavior: functional, no absolute timing**
  - No structural or implementation information

(Engineering) Models vs. Reality

- **"You can't strike oil by drilling through a map" [Golob'68]**
  - Yet, maps are incredibly useful
- **"All models are wrong, some are useful" [Box'76]**
  - Abstraction of reality

➢ **We can make definitive statements about models from which we can *infer* properties of system realizations [Kopetz]**
  - ➢ Validity of inference depends on model fidelity
  - ➢ Always approximate

➢ **Assertions about (predicted) properties are always assertions about a model of the system**
  - ➢ Never truly properties of the final implemented system

Source: E. Lee, CEDA Keynote, DAC'13.

Formal Model of a Design

- **Most tools and designers describe the behavior of a design as a relation between a set of inputs and a set of outputs**
  - This relation may be informal, even expressed in natural language
  - Such informal, ambiguous specifications may result in unnecessary redesigns…
- **A formal model of a design should consist of the following components:**
  - Functional specification
  - Set of properties
  - Set of performance indices
  - Set of constraints on performance indices

Source: M. Jacome, UT Austin.

Formal Model of a Design (2)

- **A *functional specification*, given as a set of explicit or implicit relations which involve inputs, outputs and possibly internal (state) information**

  Fully characterizes the operation of a system

- **A *set of properties* that the design must satisfy**

  Redundant: in a properly constructed system, the functional specification satisfies these properties. Yet properties are simpler / more abstract compared to the functional specification.

- **A *set of performance indices* that evaluate the quality of the design in terms of cost, reliability, speed, size, etc.**
- **A *set of constraints* on performance indices, specified as a set of inequalities**

  Bound the cost of a system

Source: M. Jacome, UT Austin.

Properties

- **A property is an assertion about the behavior, rather than a description of the behavior**
  - It is an abstraction of the behavior along a particular axis
- **Examples:**
  - *Liveness* property: when designing a network protocol, one may require that the design never *deadlocks*
  - *Fairness* property: when designing a network protocol, one may require that *any request will eventually be satisfied*

The above properties do not completely specify the behavior of the protocol, they are instead properties we require the protocol to have

- **Can include other non-functional requirements**
  - *Timeliness*: guarantees about meeting deadlines in the worst case (*real-time*)

Source: M. Jacome, UT Austin.

Properties & Models

- **Properties can be classified in three groups:**
  1. Properties that are *inherent* to the model (i.e., that can be shown formally to hold for *all specifications* described using that model)
  2. Properties that can be verified *syntactically* for a given specification (i.e., that can be shown to hold with a simple, usually polynomial-time analysis of the specification)
  3. Properties that must be verified *semantically* for a given specification (i.e., that can be shown to hold by executing, at least implicitly, the specification for all inputs that can occur)

Source: M. Jacome, UT Austin.

Model Validation

- **By construction**
  - property is inherent
- **By verification**
  - property is provable syntactically
- **By simulation**
  - check behavior for all inputs
- **By intuition**
  - property is true, I just know it is…

better be higher in this list…

Source: M. Jacome, UT Austin.

Model Validation Example

- ***Determinate Behavior* Property: the fact that the output of a system depends only on its inputs and not on some internal, hidden choice**
  - Any design described by a dataflow network is determinate, and hence this property is *inherent* (that is, need not be checked)
  - If the design is represented by a network of FSMs, determinacy can be assessed by inspection of the state transition function, and hence the property can be verified *syntactically*
  - In the discrete event models embodied in Verilog and VHDL determinacy is difficult to prove, it must be checked by exhaustive simulation, and thus the property requires *semantic* verification

Source: M. Jacome, UT Austin.

Lecture 2: Outline

- ✓ **System specification**
  - ✓ Essential issues
  - ✓ Specification modeling guidelines
  - ✓ Formal models
- **Models of Computation (MoCs)**
  - Models for reactive systems
  - Concurrency & communication

Models of Computation (MoCs)

- **A MoC is a framework in which to express what actions must be taken to complete a computation**
  - Objects and their relationships
  - Basic semantics of computation & communication
  - Vs. concrete languages (many-to-many relation)
- **MoCs need to**
  - Be *powerful/expressive enough* for the application domain
  - Have appropriate *synthesis* and *validation* semantics
- **Why different models?**
  - Different models ⇒ different properties
  - Analyzability vs. expressiveness
  - Existing programming models are poor match

➢ **Domain-specific models & languages (DSLs)**

Source: M. Jacome, UT Austin.

System-Level MoCs

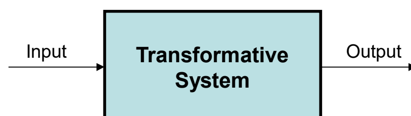
- **Consider essential aspects of systems**
  - Concurrency
  - Communication & synchronization
  - Order/time
  - Heterogeneity
- **Classify models based on**
  - How to specify behavior (computation)
  - How to specify communication
  - Implementability
  - Composability
  - Availability of tools for validation and synthesis

Source: M. Jacome, UT Austin.

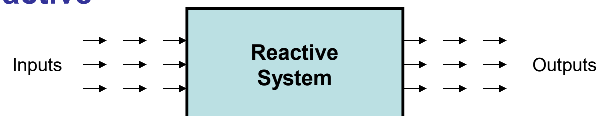
ECE382N.23: Embedded Sys Dsgn/Modeling, Lecture 2 © 2024 A. Gerstlauer 13

Cyber-Physical Systems (CPS)

- **Not transformative**



  - Output = F(Input)
  - ➢ Procedural/batch processing
- **But reactive**



  - Continuous interaction with environment
  - ➢ Sense and act on the physical world

➢ **Concurrency and time**

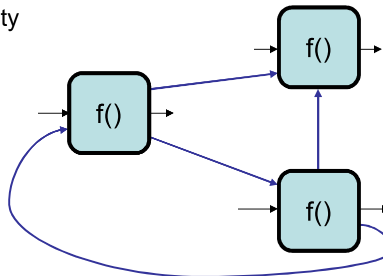
ECE382N.23: Embedded Sys Dsgn/Modeling, Lecture 2 © 2024 A. Gerstlauer 14

Models of Time (Order)

- **Untimed**
  - Partial order based on causality only
    - No ordering in time, explicit dependencies only
    - ➢ Free of implementation (purely behavioral)
    - ➢ Specification & programming, Models of computation
- **Logical**
  - Discrete time, partial order
    - Discrete instants of time (time tags $t_0 < t_1 < \cdots t_k < \cdots$), nothing in between
    - Unspecified interleaving of events with same time tag
    - ➢ Freedom of implementation
    - ➢ Simulation & execution, Design languages
- **Physical**
  - Continuous time, total order
    - Physical components naturally interleaved in (very fine) time
    - ➢ Differential equations, Hybrid models

Concurrency

- **Events/actions happening "at the same time"**
  - Undefined, unspecified or unknown order
    - Implementation will determine actual interleaving
- **Communication/synchronization establishes order**
  - Partial order, causal dependencies
    - Behavior/functionality



- **Fundamental issues: communication semantics**
  - ➢ Non-determinism, causality loops
  - ➢ Deadlocks

Determinism

- **Deterministic: same inputs always produce same results**
- **Random: probability of certain behavior**
- **Non-deterministic: undefined behavior (for some inputs)**
  - Undefined execution order
    - Statement evaluation in imperative languages: `f(a++, a++)`
    - Process & thread race conditions:

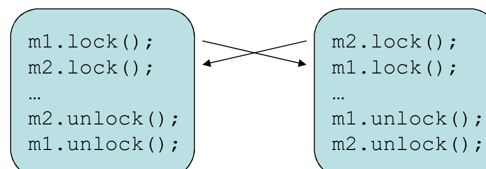
```
x = a;        a = 1;
y = b;        b = 2;
```

x = ?, y = ?

➢ **Can be desired or undesired**
  - ➢ How to ensure correctness?
    - ➢ Many possible behaviors, large verification space
    - ➢ Simulator will pick one behavior, not sufficient for verification
  - ➢ But: over-specification?
    - ➢ Leave freedom of implementation choice (concurrency)

Deadlocks

- **Circular chain of 2 or more processes which each hold a shared resource that the next one is waiting for**
  - Circular dependency through shared resources



  - ➢ Prevent chain by using the same precedence
  - ➢ Use timeouts (and retry), but: livelock

➢ **Dependency can be created when resources are shared**
  - ➢ Side effects, e.g. when blocking on filled queues/buffers

MoC Examples

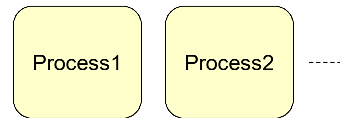
- **Programming models**
  - Imperative [C] or declarative [Lisp, Prolog]
  - ➢ Transformative not reactive, no concurrency
- **Parallel programming models**
  - Threads/processes, multi-tasking/-threading [any (RT)OS]
  - ➢ Non-determinism, race conditions, deadlocks
  - ➢ Best effort only, incomprehensible to humans/tools [Lee'06]
- **Control and logic design**
  - Finite state machines (FSMs), synchronous reactive (SR)
  - ➢ Synchronous, fine granularity of concurrency
- **Hardware description languages (HDLs)**
  - Discrete event (DE)
  - ➢ Global time, simulation but not synthesis

MoCs for System Specification

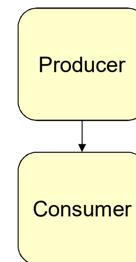
- **Process-based models**
  - Kahn Process Networks (KPNs)
  - (Synchronous) Dataflow models ((S)DF)
  - Directed Acyclic Task Graphs (DAGs)
  - …
- **State-based models**
  - Finite State Machines (FSMs)
  - Hierarchical, Concurrent State Machines (HCFSMs)
  - Petri Nets
  - …

Process-Based Models

- **Data flow**
  - Data processing w/ communication & synchronization
- **Set of processes/threads**
  - Execute in parallel
    - Concurrent composition
  - Each process is sequential
    - Imperative program



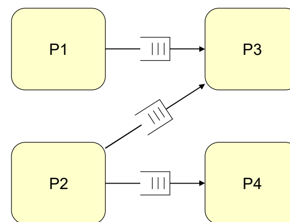
- **Inter-process communication**
  - Shared memory model
    - Synchronization: critical section/mutex, monitor, …
  - Message-passing model
    - Queues, but data losses, deadlocks



- **Traditional thread models are poor match**
  - Best effort, no correctness guarantees
  - Race conditions, deadlocks, non-determinism

Kahn Process Network (KPN) [Kahn74]

- **C-like processes communicating via FIFO channels**
  - Unbounded, uni-directional, point-to-point queues
    - Sender (`send()`) never blocks
    - Receiver (`wait()`) blocks until data available



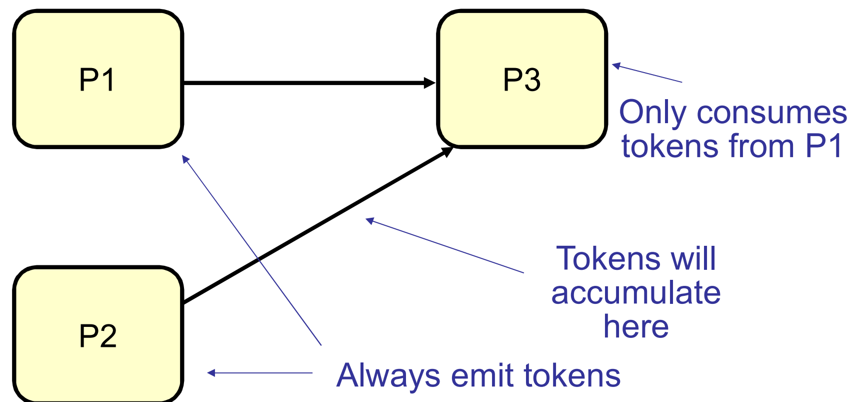
- **Deterministic**
  - Behavior does not depend on scheduling strategy
  - Focus on causality, not order (implementation independent)

Kahn Process Network (KPN) (2)

- **Determinism**
  - Process can't peek into channels and can only wait on one channel at a time
  - Output data produced by a process does not depend on the order of its inputs
  - ➢ Terminates on global deadlock
    - ➢ All process blocked on `receive()` (or have otherwise ended)
- **Formal mathematical representation**
  - Process = continuous function mapping input to output streams
- **Turing-complete, undecidable (in finite time)**
  - Terminates (deadlocks)?
  - Can run in bounded buffers (memory)?

KPN Scheduling

- **Scheduling determines memory requirements**
- **Data-driven scheduling**
  - Run processes whenever they are ready:

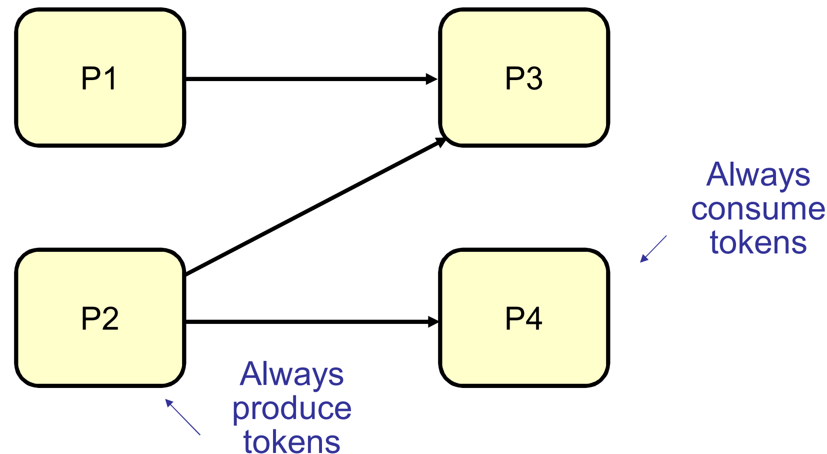


Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

© 2024 A. Gerstlauer

Demand-Driven Scheduling

- **Only run a process whose outputs are being solicited**
  - Synchronous, unbuffered message-passing
- **However...**



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

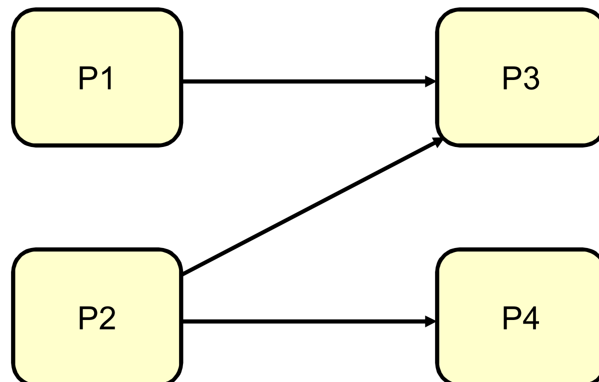
KPN Scheduling

- **Inherent tradeoffs**
  - Completeness
    - Run processes as long as they are ready
    - ➢ Might require unbounded memory
  - Boundedness
    - Block senders when reaching buffer limits
    - ➢ Potentially incomplete, artificial deadlocks and early termination
  - ➢ Data driven: completeness over boundedness
  - ➢ Demand driven: boundedness over completeness and even non-termination

➢ **Hybrid approach [Parks95]**
  - Start with smallest bounded buffers
  - Schedule with blocking `send()` until artificial deadlock
    - – At least one process blocked on `send()`
  - Increase size of smallest blocked buffer and continue

Parks' Algorithm

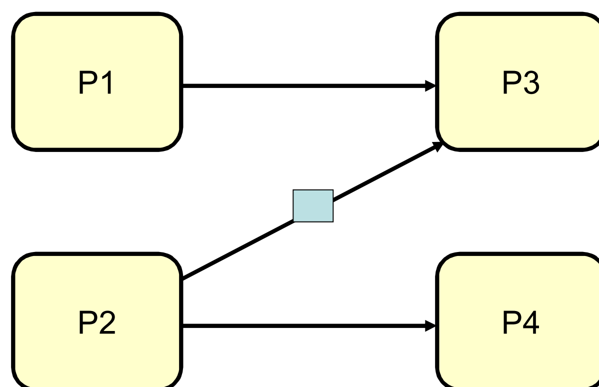
- Start with buffer size 1
- Run P1, P2, P3, P4



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

Parks' Algorithm

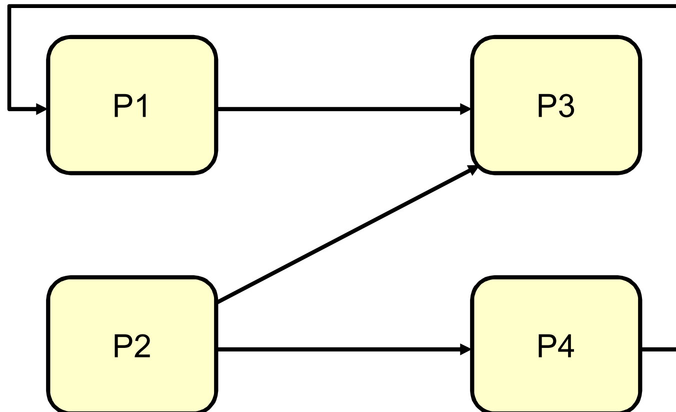
- P2 blocked
- Run P1, P3, P1, … indefinitely



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

Parks' Algorithm

- **But …**

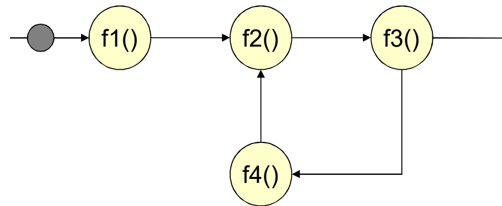


Kahn Process Networks (KPN)

- **Difficult to implement right**
  - Size of infinite FIFOs in limited physical memory?
  - Dynamic memory allocation, dependent on schedule
  - Dynamic scheduling & context switching
  - Boundedness vs. completeness vs. non-termination (deadlocks), are undecidable, depend on runtime schedule
  - Message-passing communication [MPI, Unix pipes]
  - How to model non-determinism? (e.g. merge process)
- **Parks' algorithm**
  - Bounded over complete (non-terminating) execution
  - ➢ Does not find every complete, bounded schedule [Geilen03]
  - ➢ Does not guarantee minimum memory usage
  - ➢ Deadlock detection?

Dynamic Dataflow [Dennis74]

- **Breaking processes down into network of *actors***
  - Atomic blocks of computation, executed when *firing*
    - *Functional,* no side effects, no state: outputs purely a function of inputs
  - Fire when required number of input *tokens* are available
    - Consume & produce number of tokens on input(s) & output(s)
  - ➢ Separate computation & communication/synchronization
    - ➢ Actors (units of computation) & tokens (units of communication)
  - ➢ Dataflow graph (DFG)

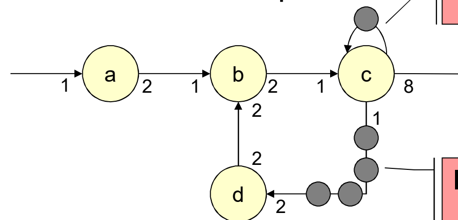


➢ **Firing rules dynamically chosen based on token patterns**
- ➢ Deterministic, but still undecidable, still Turing-complete…

Synchronous Dataflow (SDF) [Lee86]

- **Fixed number of tokens per firing**
  - Consume fixed number of inputs
    - Single firing rule with fixed wildcard patterns
  - Produce fixed number of outputs

Actors are stateless
➢ Explicit self-loop to model state



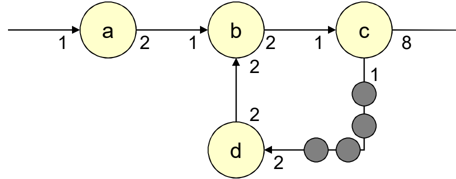
Initialization
➢ Delay
➢ Prevent deadlock

➢ **Can be scheduled statically**
- ➢ Flow of data through system does not depend on values

➢ **Find a repeated sequence of firings**
- ➢ Run actors in proportion to their rates
- ➢ Fixed buffer sizes, no under- or over-flow

SDF Scheduling (1)

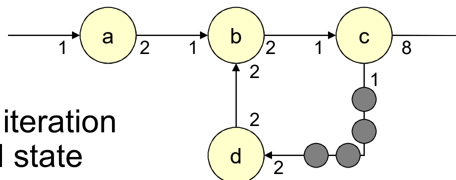
- **Solve system of linear rate equations**
  - Balance equations per arc
    - *2a = b*
    - *2b = c*
    - *b = d*
    - *2d = c*
    - ➢ *4a = 2b = c = 2d*
  - Inconsistent systems
    - Only solvable by setting rates to zero
    - Would otherwise (if scheduled dynamically) accumulate tokens
  - Underconstrained systems
    - Disjoint, independent parts of a design



➢ **Compute repetitions vector**
- ➢ Linear-time depth-first graph traversal algorithm

SDF Scheduling (2)

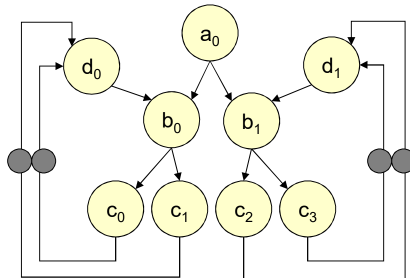
- **Periodically schedule actors in proportion to their rates**
  - Smallest integer solution
    - *4a = 2b = c = 2d*
    - ➢ *a = 1, b = 2, c = 4, d = 2*
  - Symbolically simulate one iteration of graph until back to initial state
    - *Admissible* iff no deadlock
    - Repeatedly execute this schedule
    - ➢ *adbccdbcc = a(2db(2c))*
    - ➢ *a(2db)(4c)*



- ➢ Periodic admissible sequential scheduling (PASS)
  - ➢ Single processor: memory requirements (buffer size) vs. code size
    - *a(2db(2c))*: 2 token slots on each arc for total of 8 token buffers
    - *a(2db)(4c)*: 12 token buffers
    - ➢ Single appearance schedule & looped code generation
- ➢ Periodic admissible parallel scheduling (PAPS)
  - ➢ Multi-processor: latency/throughput vs. buffer sizes

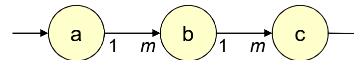
SDF Scheduling (3)

- **Precedence graph**
  - Homogeneous SDF (HSDF) conversion
    - All rates are 1, each node represents one actor instance/firing

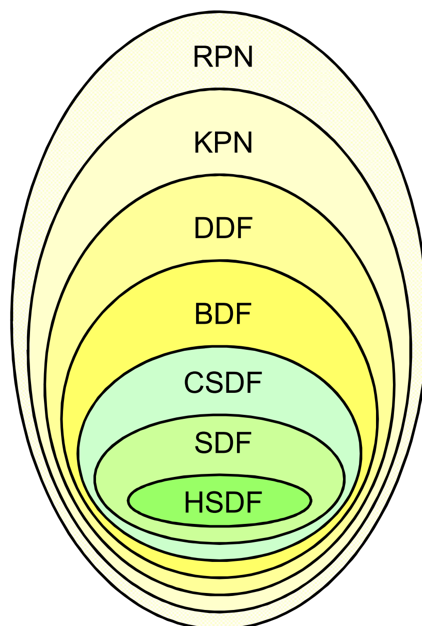


  - ➢ Scheduling = graph traversal

- ➢ **Worst-case exponential complexity**
  - Number of nodes in HSDF vs. SDF



Process-Based MoCs



Yellow: Turing complete

RPN	Reactive Process Network
KPN	Kahn Process Network
DDF	Dynamic Dataflow
BDF	Boolean Dataflow
CSDF	Cyclo-Static Dataflow
SDF	Synchronous Dataflow
HSDF	Homogeneous SDF

Source: T. Basten, MoCC 2008.

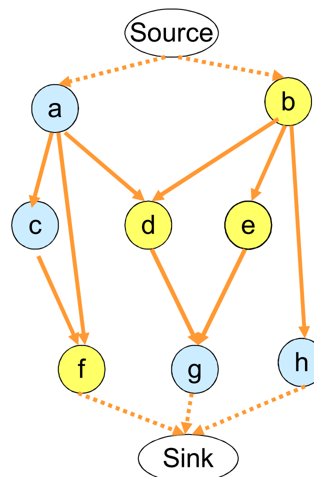
Dataflow Variants

- **Dynamic dataflow extensions**
  - Modal models
    - Reactive process networks (RPN) [Geilen'04]
    - Parameterized dataflow (PDF) [Bhattacharya'01]
    - Heterochronous dataflow (HDF) [Lee'05]
    - Scenario-aware dataflow (SADF) [Theelen'06]
    - ➢ Parameter changes between iterations driven by state machine model
  - Dataflow & actor languages
    - Various languages w/ extensions [CAL, …]
    - Structured dataflow w/ branching [LabView's G language]
    - Deterministic peeking, teleport/bypass messages [StreamIt]
- **Timed dataflow extensions**
  - Time synchronous dataflow (TSDF) [Agilent ADS]
    - Fixed sampling/execution rates on arcs and actors
  - Hybrid continuous-discrete time models
    - Discrete models as piecewise constant continuous signals [Simulink]
    - Sampling at discrete/continuous interfaces [SystemC-AMS]

Task Graphs

- **HSDF models without back-edges/cycles**
- **Directed acyclic graphs (DAGs)**
  - Task-level parallelism
  - Actors/nodes are tasks
  - Imperative functions

➢ **Widely-used in system design**
  - DAG scheduling in cluster, cloud & super-computing [Apache Spark]
  - ML deployment & compilers [TensorFlow, Apache TVM, …]



Process Calculi

- **Rendezvous-style, synchronous communication**
  - Communicating Sequential Processes (CSP) [Hoare78]
  - Calculus of Communicating Systems (CCS) [Milner80]
  - ➢ Restricted interactions
- ➢ **Formal, mathematical framework: process algebra**
  - Algebra = <objects, operations, axioms>
    - Objects: processes {*P, Q, …*}, channels {*a, b, …*}
    - Composition operators: parallel ($P \parallel Q$), prefix/sequential ($a \rightarrow P$), choice ($P+Q$)
    - Axioms: indemnity ($\emptyset \parallel P = P$), commutativity ($P+Q=Q+P$, $P \parallel Q = Q \parallel P$)
  - ➢ Manipulate processes by manipulating expressions
- ➢ **Parallel programming languages**
  - ➢ CSP-based [Occam/Transputer, Handle-C]

Lecture 2: Summary

- **System & application specification**
  - „Golden“ input to design flow
- **Models of Computation (MoCs)**
  - Formally express behavior
  - Tradeoffs between analyzability vs. expressiveness
- **Process-based models**
  - Task-level parallelism, dataflow driven
- **State-based models**
  - Focus on control flow within single process/thread
  - Synchronous, operation-level concurrency

© 2024 A. Gerstlauer