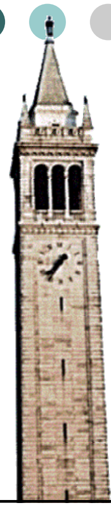
Embedded Software:

Leveraging Concurrent Models of Computation



Edward A. Lee

Professor, UC Berkeley

Center for Hybrid and Embedded Software Systems (CHESS)

Citris Information Days

June 30 – July 1, 2004

Infineon, Munich, Germany

Are Resource Limitations the Key Defining Factor for Embedded Software?

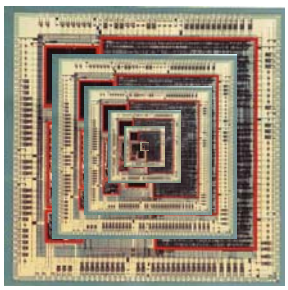
- small memory
- small data word sizes
- relatively slow clocks

To deal with these problems, emphasize efficiency:

- write software at a low level (in assembly code or C)
- avoid operating systems with a rich suite of services
- develop specialized computer architectures
  - programmable DSPs
  - network processors

This is how embedded SW has been designed for 25 years

Why hasn't Moore's law changed all this in 25 years?



Hints that Embedded SW Differs Fundamentally from General Purpose SW

- object-oriented techniques are rarely used
  - classes and inheritance
  - dynamic binding
- processors avoid memory hierarchy
  - virtual memory
  - dynamically managed caches
- memory management is avoided
  - allocation/de-allocation
  - garbage collection

To be fair, there are some applications: e.g. Java in cell phones, but mainly providing the services akin to general purpose software.

More Hints: Fundamentally Different Techniques Applied to Embedded SW.

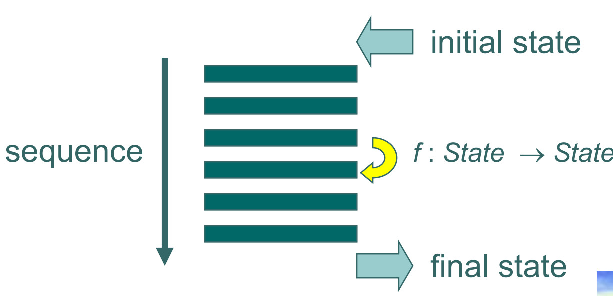
method-call models of computation

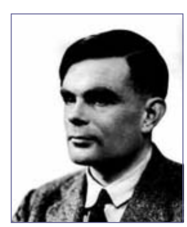
- nesC/TinyOS
  - developed for programming very small programmable sensor nodes called "motes"
- Click
  - created to support the design of software-based network routers

actor-oriented models of computation

- Simulink with Real-Time Workshop
  - created for embedded control software and widely used in the automotive industry
- Lustre/SCADE
  - created for safety-critical embedded software (e.g. avionics software)

Standard Software Abstraction (20-th Century Computation)



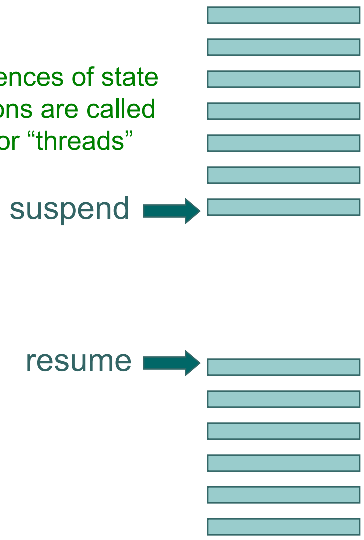


Alan Turing

- Time is irrelevant
- All actions are ordered

Standard Software Abstraction: Processes or Threads

Infinite sequences of state transformations are called "processes" or "threads"



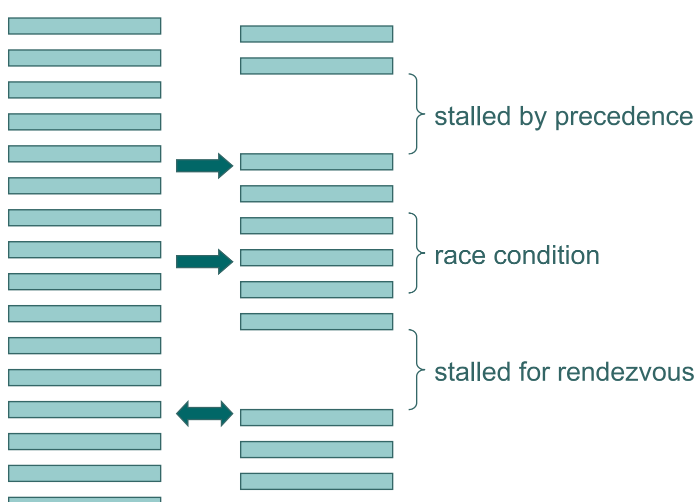
The operating system (typically) provides:

- suspend/resume
- mutual exclusion
- semaphores

Standard Software Abstraction: Concurrency via Interacting Threads

Potential for race conditions, deadlock, and livelock severely compromises software reliability.

These methods date back to the 1960's (Dijkstra).

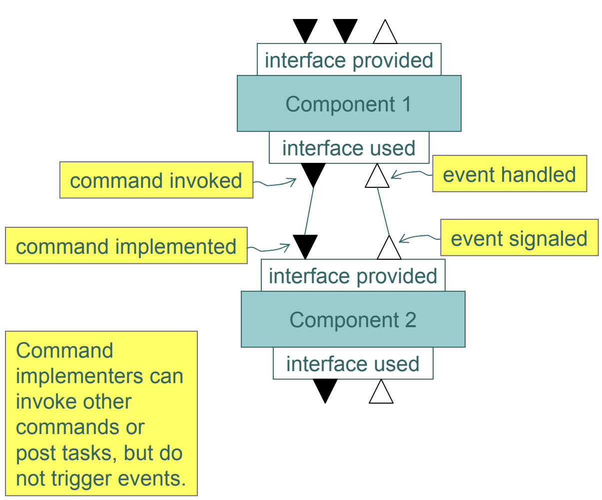


A Stake in the Ground

Nontrivial concurrent programs based on processes, threads, semaphores, and mutexes are incomprehensible to humans, and should not be used in safety critical software.

- No amount of software engineering process is going to solve this problem.
  - the human brain doesn't work this way.
- Formal methods may help
  - scalability? understandability?
- Better concurrency abstractions will help more
  - four promising examples: nesC/TinyOS, Click, Lustre/SCADE, and Simulink.

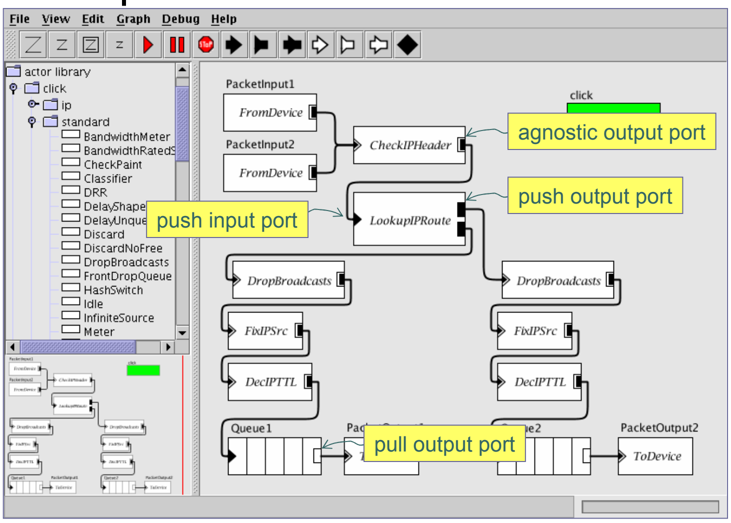
Alternative Concurrency Models: First example: nesC and TinyOS



Typical usage pattern:

- hardware interrupt signals an event.
- event handler posts a task.
- tasks are executed when machine is idle.
- tasks execute atomically w.r.t. one another.
- tasks can invoke commands and signal events.
- hardware interrupts can interrupt tasks.
- exactly one monitor, implemented by disabling interrupts.

Alternative Concurrency Models: Second example: Click



Implementation of Click with a visual syntax in Mescal (Keutzer, et al.)

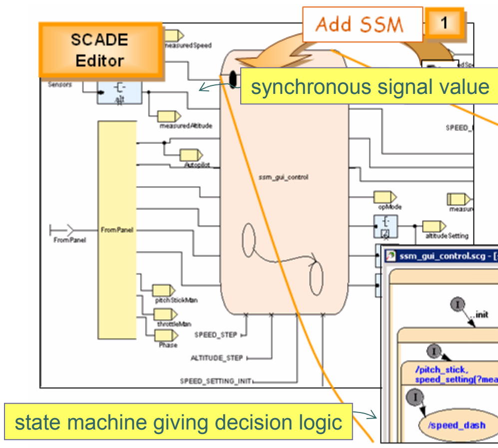
Typical usage pattern:
- queues have push input, pull output.
- schedulers have pull input, push output.
- thin wrappers for hardware have push output or pull input only.

Observations about nesC/TinyOS & Click

- Very low overhead
- Bounded stack sizes
- No (unintended) race conditions
- No threads or processes
- Access to timers
- Can create thin wrappers around hardware

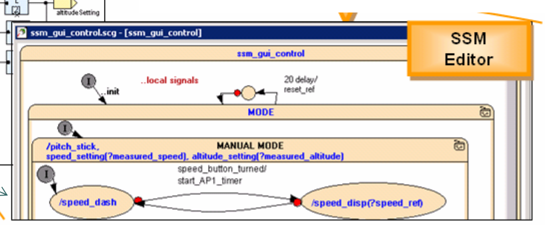
- But rather specialized
  - Unfamiliar to programmers
  - No preemption (tasks must be decomposed)

Alternative Concurrency Models: Third example: Lustre/SCADE



Typical usage pattern:

- specify tasks aligned to a master “clock” and subclocks
- clock calculus checks for consistency and deadlock
- decision logic is given with hierarchical state machines.



from http://www.esterel-technologies.com/

Observations about Lustre/SCADE

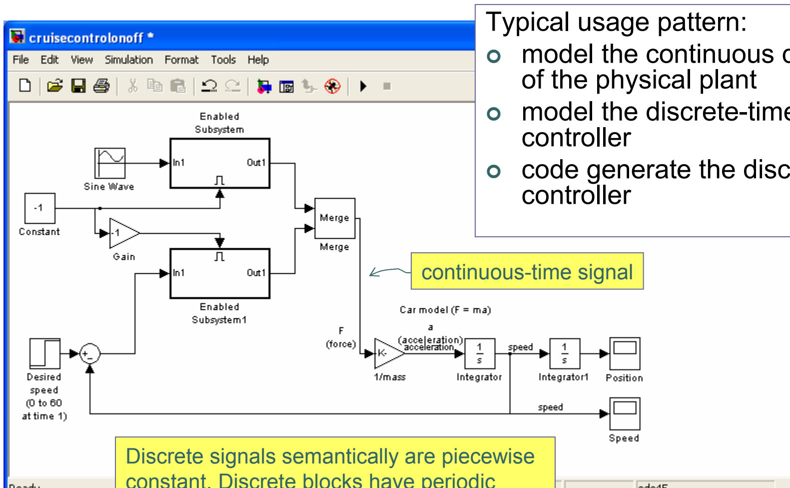
- Very low overhead
- Bounded stack sizes
- No (unintended) race conditions
- No threads or processes
- Verifiable (finite) behavior
- Certified compiler (for use in avionics)

- But rather specialized
  - Unfamiliar to programmers
  - No preemption
  - No time

The Real-Time Problem

- Programming languages have no time in their core semantics
- Temporal properties are viewed as "non-functional"
- Precise timing is poorly supported by hardware architectures
- Operating systems provide timed behavior on a best-effort basis (e.g. using priorities).
- Priorities are widely misused in practice

Alternative Concurrency Models: Fourth example: Simulink



Typical usage pattern:
- model the continuous dynamics of the physical plant
- model the discrete-time controller
- code generate the discrete-time controller

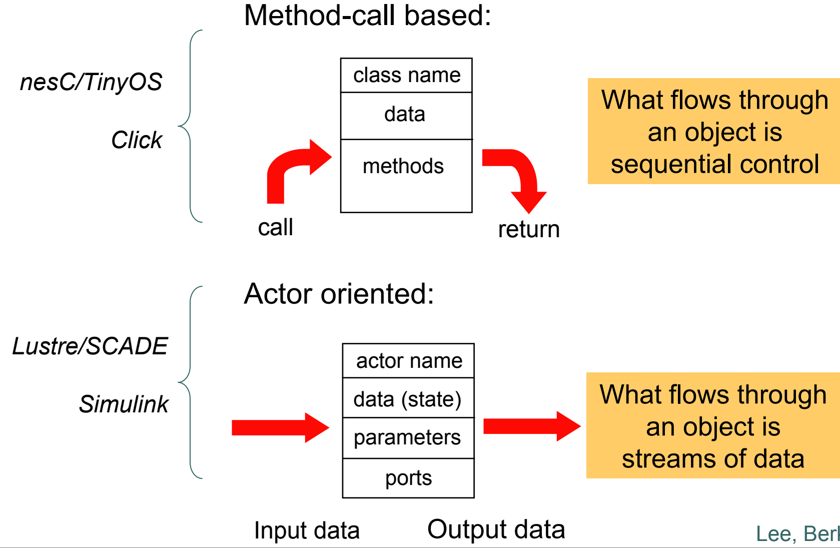
continuous-time signal

Discrete signals semantically are piecewise constant. Discrete blocks have periodic execution with a specified "sample time."

Observations about Simulink

- Bounded stack sizes
- Deterministic (no race conditions)
- Timing behavior is explicitly given
- Efficient code generator (for periodic discrete-time)
- Supports concurrent tasks
- No threads or processes visible to the programmer
  - But cleverly leverages threads in an underlying O/S.
- But rather specialized
  - Periodic execution of all blocks
  - Accurate schedulability analysis is difficult

Two Distinct Component Interaction Mechanisms



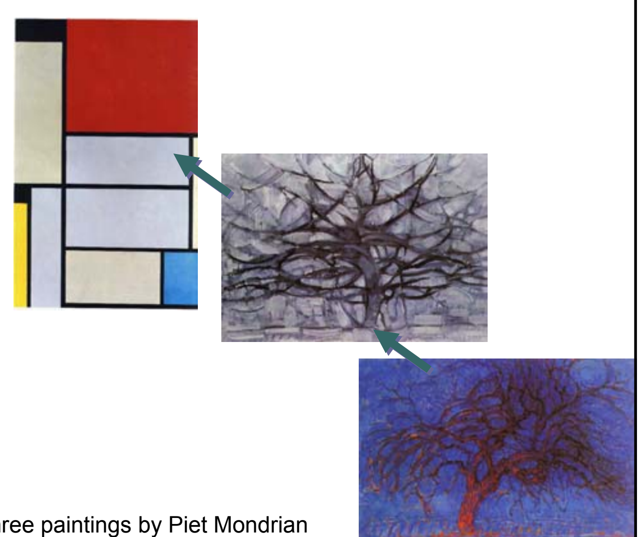
Terminology Problem

- Of these, only nesC is recognized as a "programming language."
- I will call them "platforms"
  - A platform is a set of possible designs:
    - the set of all nesC/TinyOS programs
    - the set of all Click configurations
    - the set of all SCADE designs
    - the set of all Simulink block diagrams

Abstraction

These four "platforms" offer distinct alternative abstractions (*models of computation*).

They are highly concurrent, and very different from the traditional threads and processes.



Three paintings by Piet Mondrian

How Many More (Useful) Models of Computation Are There?

Here are a few actor-oriented platforms:

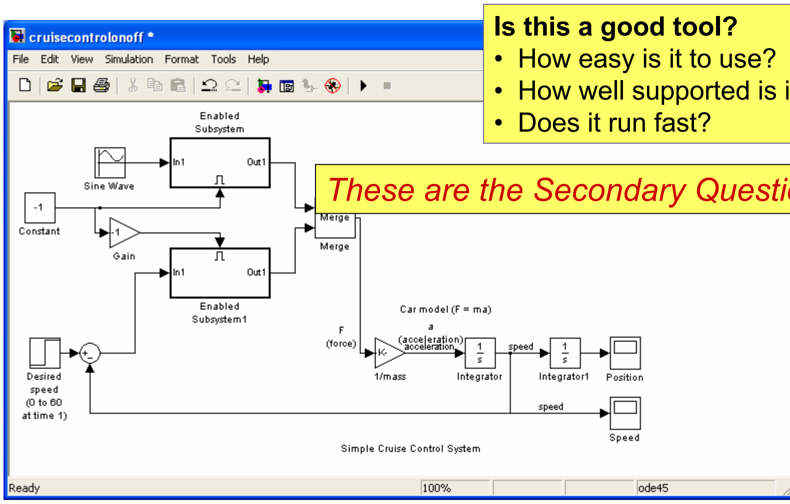
- Labview (synchronous dataflow)
- Modelica (continuous-time, constraint-based)
- CORBA event service (distributed push-pull)
- SPW (synchronous dataflow)
- OPNET (discrete events)
- VHDL, Verilog (discrete events)
- SDL (process networks)
- …

Many Variants – Consider Dataflow Alone:

- Computation graphs [Karp & Miller - 1966]
- Process networks [Kahn - 1974]
- Static dataflow [Dennis - 1974]
- Dynamic dataflow [Arvind, 1981]
- K-bounded loops [Culler, 1986]
- Synchronous dataflow [Lee & Messerschmitt, 1986]
- Structured dataflow [Kodosky, 1986]
- PGM: Processing Graph Method [Kaplan, 1987]
- Synchronous languages [Lustre, Signal, 1980’s]
- Well-behaved dataflow [Gao, 1992]
- Boolean dataflow [Buck and Lee, 1993]
- Multidimensional SDF [Lee, 1993]
- Cyclo-static dataflow [Lauwereins, 1994]
- Integer dataflow [Buck, 1994]
- Bounded dynamic dataflow [Lee and Parks, 1995]
- Heterochronous dataflow [Girault, Lee, & Lee, 1997]
- …

Many tools, software frameworks, and hardware architectures have been built to support one or more of these.

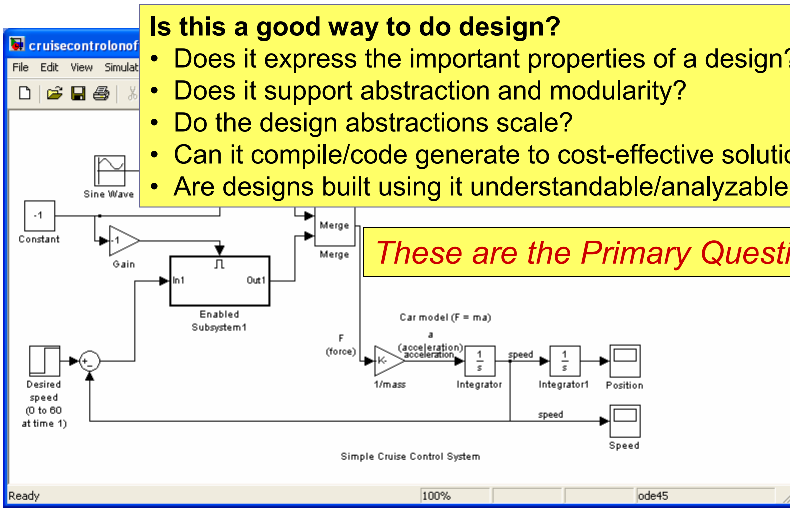
How to Choose a Platform: Tools Focus



Is this a good tool?
- How easy is it to use?
- How well supported is it?
- Does it run fast?

These are the Secondary Questions!

How to Choose a Platform: Abstraction Focus



Is this a good way to do design?
- Does it express the important properties of a design?
- Does it support abstraction and modularity?
- Do the design abstractions scale?
- Can it compile/code generate to cost-effective solutions?
- Are designs built using it understandable/analyzable?

These are the Primary Questions!

The Meta Question

How can we objectively evaluate the alternatives?

Meta Platforms
Supporting Multiple Models of Computation

- Ptolemy Classic and Ptolemy II (UC Berkeley)
- GME (Vanderbilt)
- Metropolis (UC Berkeley)
- ROOM (Rational)
- SystemC (Synopsys and others)

To varying degrees, each of these provides an *abstract semantics* that gets specialized to deliver a particular model of computation.

ROOM is evolving into an OMG standard (composite structures in UML 2)

Conclusion

- Embedded software is an immature technology
- Focus on “platforms” not “languages”
- Platforms have to:
  - expose hardware (with thin wrappers)
  - embrace time in the core semantics
  - embrace concurrency in the core semantics
- API’s over standard SW methods won’t do
- Ask about the “abstractions” not the “tools”

- Many questions remain…