

Imperative languages

Peter Marwedel
Informatik 12
TU Dortmund
Germany

Model of computation

<i>Communication/ Computation</i>	Shared memory	Message passing	
		Synchronous	Asynchronous
FSM	StateCharts		SDL
Data flow model			Kahn process networks, SDF
Imperative von Neumann computing	C, C++, Java	C, C++, Java with message passing libraries	
		CSP, ADA	

Java (1)

Potential benefits:

- Clean and safe language
- Supports multi-threading (no OS required?)
- Platform independence (relevant for telecommunications)

Problems:

- Size of Java run-time libraries? Memory requirements.
- Access to special hardware features
- Garbage collection time
- Non-deterministic dispatcher
- Performance problems
- Checking of real-time constraints

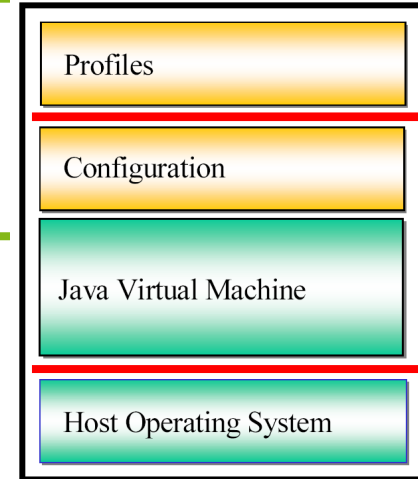
Overview over Java 2 Editions



“J2ME ... addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers, all the way up to the TV set-top box..”

Based on <http://java.sun.com/products/cldc/wp/KVMwp.pdf>

Software stack for J2ME



- **Java Virtual Machine:** implementation of a Java VM, customized for a particular device's host OS and supports a particular J2ME configuration.
- **Configuration:** defines the minimum set of Java VM features and Java class libraries available on a particular "category" of devices representing a particular "horizontal" market segment. In a way, a configuration defines the "lowest common denominator" of the Java platform features and libraries that the developers can assume to be available on all devices.
- **Profile:** defines the minimum set of Application Programming Interfaces (APIs) available on a particular "family" of devices representing a particular "vertical" market segment. Profiles are implemented "upon" a particular configuration. Applications are written "for" a particular profile and are thus portable to any device that "supports" that profile. A device can support multiple profiles.

Based upon
<http://java.sun.com/products/cldc/wp/KVMwp.pd>

KVM and CLDC

- ***The K Virtual Machine:***

Highly portable Java VM designed for small memory, limited-resource, network-connected devices, e.g.: cell phones, pagers, & personal organizers. Devices typically contain 16- or 32-bit processors and a minimum total memory footprint of ~128 kilobytes.

- ***Connected, Limited Device Configuration (CLDC)***

Designed for devices with intermittent network connections, slow processors and limited memory – devices such as mobile phones, two way pagers and PDAs. These devices typically have either 16- or 32-bit CPUs, and a minimum of 128 KB to 512 KB of memory.

CDC Configuration and MIDP 1.0 + 2.0 Profiles

- **CDC:** Designed for devices that have more memory, faster processors, and greater network bandwidth, such as TV set-top boxes, residential gateways, in-vehicle telematics systems, and high-end PDAs. Includes a full-featured Java VM, & a larger subset of the J2SE platform. Most CDC-targeted devices have 32-bit CPUs & ≥ 2 MB of memory.
- **Mobile Information Device Profile (MIDP):** Designed for mobile phones & entry-level PDAs. Offers core application functionality for mobile applications, including UI, network connectivity, local data storage, & application management. With CLDC, MIDP provides Java runtime environment leveraging capabilities of handheld devices & minimizing memory and power consumption.

Real-time features of Java

J2ME, KVM, CLDC & MIDP not sufficient for real-time behavior. Real-time specification for Java (JSR-1) addresses 7 areas:

1. Thread Scheduling and Dispatching
2. Memory Management:
3. Synchronization and Resource Sharing
4. Asynchronous Event Handling
5. Asynchronous Transfer of Control
6. Asynchronous Thread Termination
7. Physical Memory Access

Designed to be used with any edition of Java.

[<http://www.rti.org>] [<https://rtsj.dev.java.net/rtsj-V1.0.pdf>]

Example: different types of memory areas

Area of memory may be used for the allocation of objects.

There are four basic types of memory areas

(partially excluded from garbage collection):

1. Scoped memory provides a mechanism for dealing with a class of objects that have a lifetime defined by syntactic scope.
2. **Physical memory** allows objects to be created within specific physical memory regions that have particular important characteristics, such as memory that has substantially faster access.
3. **Immortal memory** represents an area of memory containing objects that, once allocated, exist until the end of the application, i.e., the objects are immortal.
4. Heap memory represents an area of memory that is the heap. The RTSJ does not change the determinant of lifetime of objects on the heap. The lifetime is still determined by visibility.

[<https://rtsj.dev.java.net/rtsj-V1.0.pdf>]

Message passing libraries

Example: MPI/Open MPI

- Library designed for high-performance computing (hpc)
- Based on asynchronous/synchronous message passing
- Comprehensive, popular library
- Available on a variety of platforms
- Considered also for multiple processor system-on-a-chip (MPSoC) programming for embedded systems;
- MPI includes many copy operations to memory ☹️ (memory speed ~ communication speed for MPSoCs);
Appropriate MPSoC programming tools missing.

http://www.mhpcc.edu/training/workshop/mpi/MAIN.html#Getting_Started

MPI (1)

Sample blocking library call (for C):

- `MPI_Send(buffer, count, type, dest, tag, comm)` where
 - *buffer*: Address of data to be sent
 - *count*: number of data elements to be sent
 - *type*: data type of data to be sent (e.g. `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, ...)
 - *dest*: process id of target process
 - *tag*: message id (for sorting incoming messages)
 - *comm*: communication context = set of processes for which destination field is valid
 - function result indicates success

http://www.mhpcc.edu/training/workshop/mpi/MAIN.html#Getting_Started

MPI (2)

Sample non-blocking library call (for C):

- `MPI_Isend(buffer, count, type, dest, tag, comm, request)`
where
 - *buffer ... comm*: same as above
 - *request*: the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation.

http://www.mhpcc.edu/training/workshop/mpi/MAIN.html#Getting_Started

Network Communication Protocols

- e.g. JXTA -

- *Open source peer-to-peer protocol specification.*
- *Defined as a set of XML messages that allow any device connected to a network to exchange messages and collaborate independently of the network topology.*
- *Designed to allow a range of devices to communicate. Can be implemented in any modern computer language.*
- *JXTA peers create a virtual overlay network, allowing a peer to interact with other peers even when some of the peers and resources are behind firewalls and NATs or use different network transports. Each resource is identified by a unique ID, so that a peer can change its localization address while keeping a constant identification number.*

<http://en.wikipedia.org/wiki/JXTA>

Network Communication Protocols

- e.g. DPWS -

- *The **Devices Profile for Web Services (DPWS)** defines a minimal set of implementation constraints to enable secure Web Service messaging, discovery, description, and eventing on resource-constrained devices. ...*
- *DPWS specifies a set of built-in services:*
 - *Discovery services: used by a device connected to a network to advertise itself and to discover other devices.*
 - *Metadata exchange services: provide dynamic access to a device's hosted services and to their metadata.*
 - *Publish/subscribe eventing services: allowing other devices to subscribe to asynchronous event messages*
- *Lightweight protocol, supporting dynamic discovery, ... its application to automation environments is clear.*

http://en.wikipedia.org/wiki/Devices_Profile_for_Web_Services

Synchronous message passing: CSP

- **CSP** (communicating sequential processes)
[Hoare, 1985],
rendez-vous-based communication:
Example:



process A

```
..  
var a ...  
  a:=3;  
  c!a; -- output  
end
```

process B

```
..  
var b ...  
  ...  
  c?b; -- input  
end
```

Synchronous message passing: ADA

After Ada Lovelace (said to be the 1st female programmer).
US Department of Defense (DoD) wanted to avoid multitude
of programming languages

☞ Definition of requirements

☞ Selection of a language from a set of competing designs
(selected design based on PASCAL)

ADA'95 is object-oriented extension of original ADA.

Salient: task concept

Synchronous message passing: Using of tasks in ADA

procedure example1 **is**

task a;

task b;

task body a **is**

-- local declarations for a

begin

-- statements for a

end a;

task body b **is**

-- local declarations for b

begin

-- statements for b

end b;

begin

-- Tasks a and b will start before the first

-- statement of the body of example1

end;

Synchronous message passing: ADA-rendez-vous

```
task screen_out is  
  entry call_ch(val:character; x, y: integer);  
  entry call_int(z, x, y: integer);  
end screen_out;  
task body screen_out is
```

...

```
select  
  accept call_ch ... do ..  
  end call_ch;  
or  
  accept call_int ... do ..  
  end call_int;  
end select;
```



```
Sending a message:  
begin  
  screen_out.call_ch('Z',10,20);  
exception  
  when tasking_error =>  
    (exception handling)  
end;
```

Other imperative languages

- **Pearl:** Designed in Germany for process control applications. Dating back to the 70s. Popular in Europe.
- **Chill:** Designed for telephone exchange stations. Based on PASCAL.

Threads revisited

☞ Talk of Ed Lee at the Graz ARTEMIS conference:
http://www.artemis-office.org/DotNetNuke/LinkClick.aspx?link=10_00-Lee.pdf&tabid=98&mid=488



Consider a Simple Example

“The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):



Example: Observer Pattern in Java


```
public void addListener(listener) {...}

public void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

**Will this work in a
multithreaded context?**

Thanks to Mark S. Miller for the details
of this example.



Example: Observer Pattern With Mutual Exclusion (Mutexes)

```
public synchronized void addListener(listener) {...}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

**JavaSoft recommends against this.
What's wrong with it?**

Mutexes using Monitors are Minefields

```
public synchronized void addListener(listener) {...}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(), deadlock!



Simple Observer Pattern Becomes Not So Simple

```
public synchronized void addListener(listener) {...}
```

```
public void setValue(newValue) {
```

```
    synchronized(this) {
```

```
        myValue = newValue;
```

```
        listeners = myListeners.clone();
```

```
    }
```

*while holding lock, make copy
of listeners to avoid race
conditions*

*notify each listener outside of
synchronized block to avoid
deadlock*


```
for (int i = 0; i < listeners.length; i++) {
```

```
    listeners[i].valueChanged(newValue)
```

```
}
```

```
}
```

**This still isn't right.
What's wrong with it?**



Simple Observer Pattern: How to Make It Right?

```
public synchronized void addListener(listener) {...}
```

```
public void setValue(newValue) {  
    synchronized(this) {  
        myValue = newValue;  
        listeners = myListeners.clone();  
    }  
}
```

```
for (int i = 0; i < listeners.length; i++) {  
    listeners[i].valueChanged(newValue)  
}
```

```
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value changes in the wrong order!

What it Feels Like to Use the *synchronized* Keyword in Java



Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, *Scientific American*, September 1999.



Families of Possible Solutions

- Train programmers to use threads.
- Improve software engineering processes.
- Identify and apply design patterns.
- Quantify quality of service.
- Verify system properties formally.

None of these deliver a rigorous, analyzable, and understandable model of concurrency.



A stake in the ground...

*Nontrivial software written with threads,
semaphores, and mutexes is
incomprehensible to humans.*



Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes).

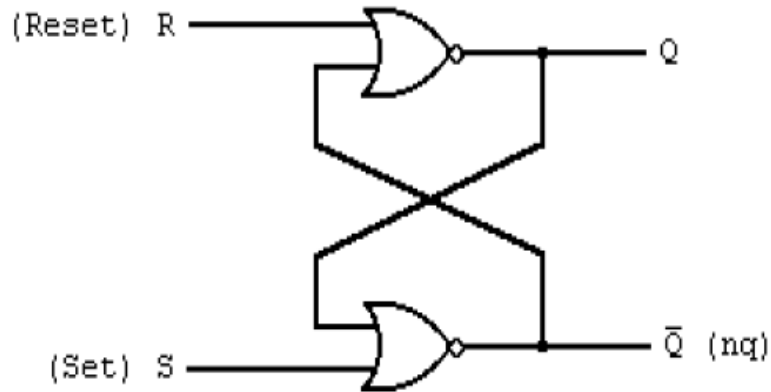


Improve Threads? Or Replace Them?

- Improve threads
 - Pruning tools (mutexes, semaphores, ...)
 - OO programming
 - Coding rules (Acquire locks in the same order...)
 - Libraries (Stapl, Java 5.0, ...)
 - Patterns (MapReduce, Transactions, ...)
 - Formal verification (Blast, thread checkers, ...)
 - Enhanced languages (Split-C, Cilk, Guava, ...)
 - Enhanced mechanisms (Promises, futures, ...)

- Change concurrency models

Threads are Not the Only Possibility: 1st example: Hardware Description Languages



SR Latch

e.g. VHDL:

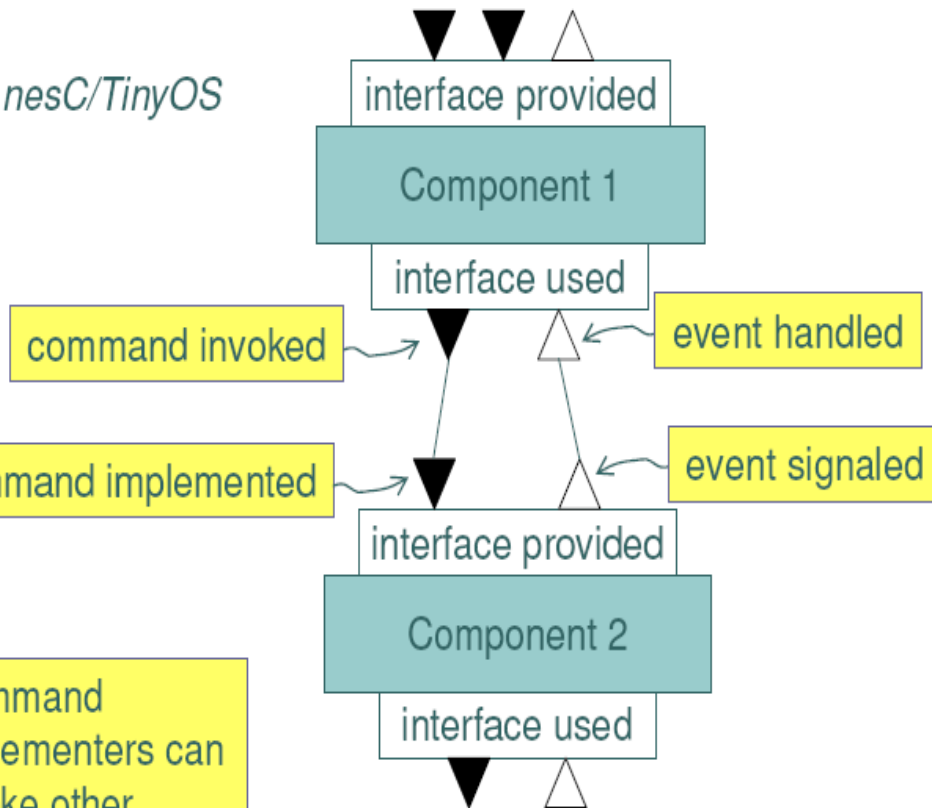
```
entity latch is  
  port (s,r : in bit;  
        q,nq : out bit);  
end latch;
```

```
architecture dataflow of latch is  
begin  
  q<=r nor nq;  
  nq<=s nor q;  
end dataflow;
```


Threads are Not the Only Possibility:

2nd example: Sensor Network Languages

e.g. nesC/TinyOS

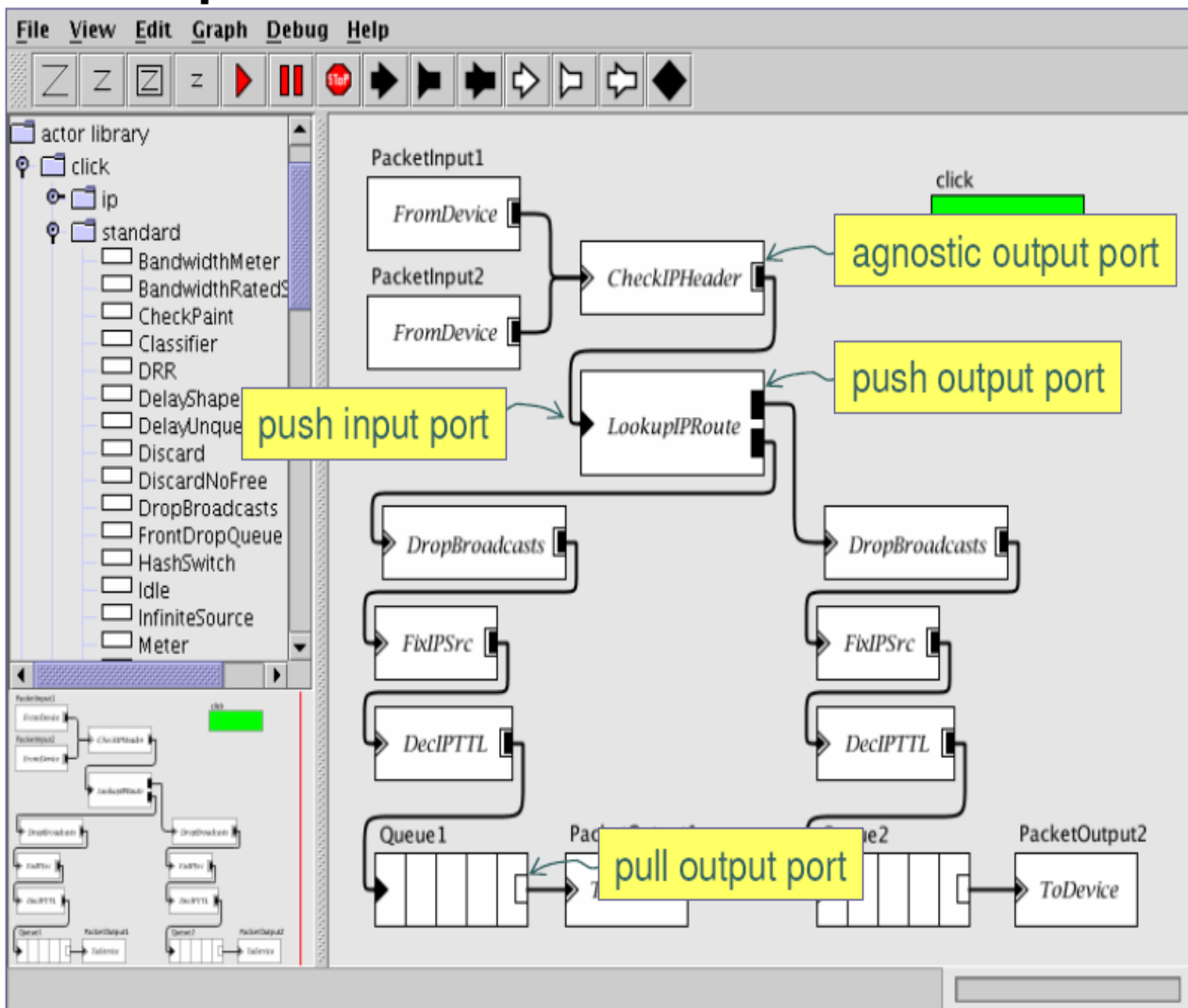


Command implementers can invoke other commands or post tasks, but do not trigger events.

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.

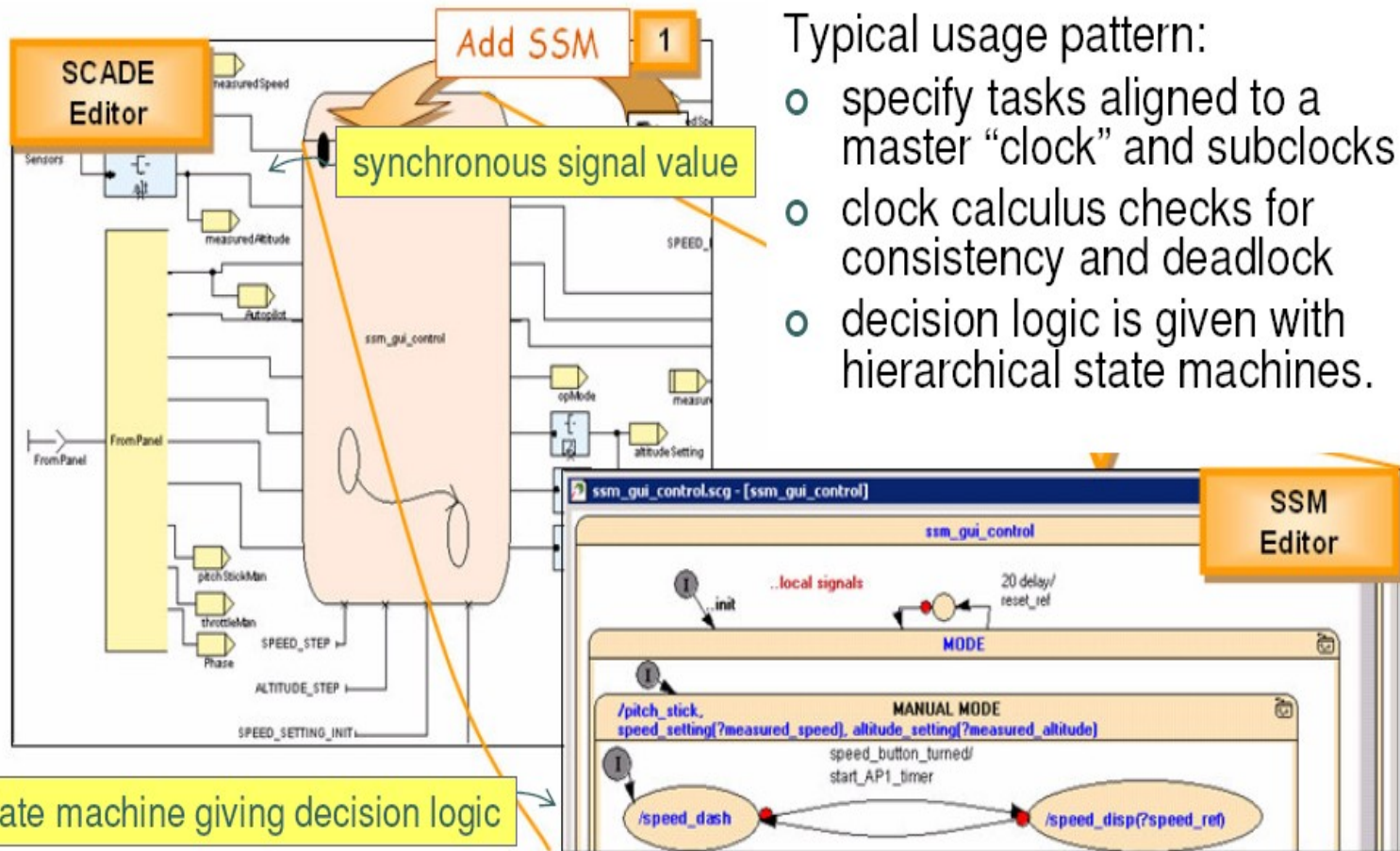
Threads are Not the Only Possibility: 3rd example: Network Languages



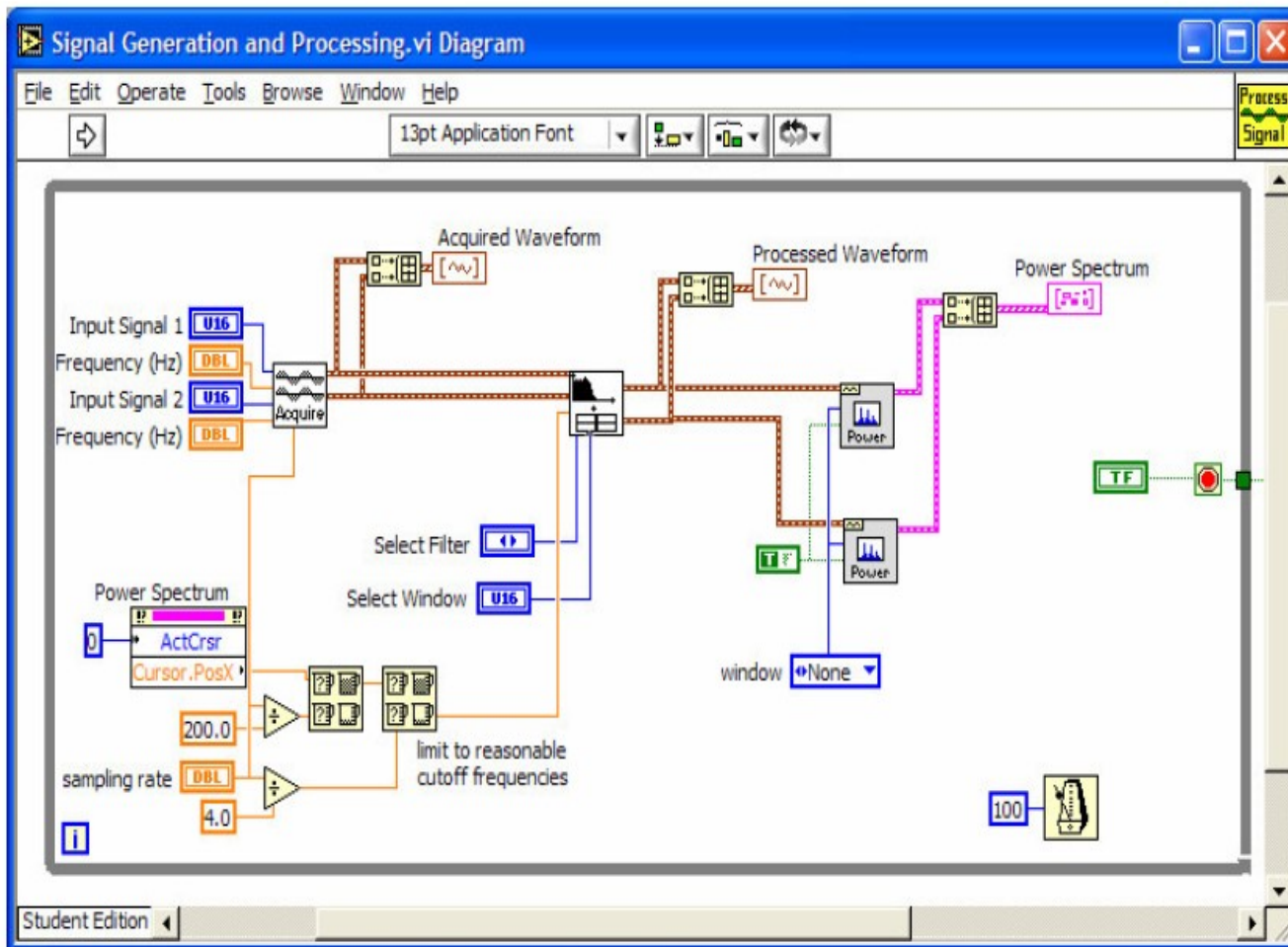
Typical usage pattern:

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

Threads are Not the Only Possibility: 4th example: Synchronous Languages



Threads are Not the Only Possibility: 5th example: Instrumentation Languages



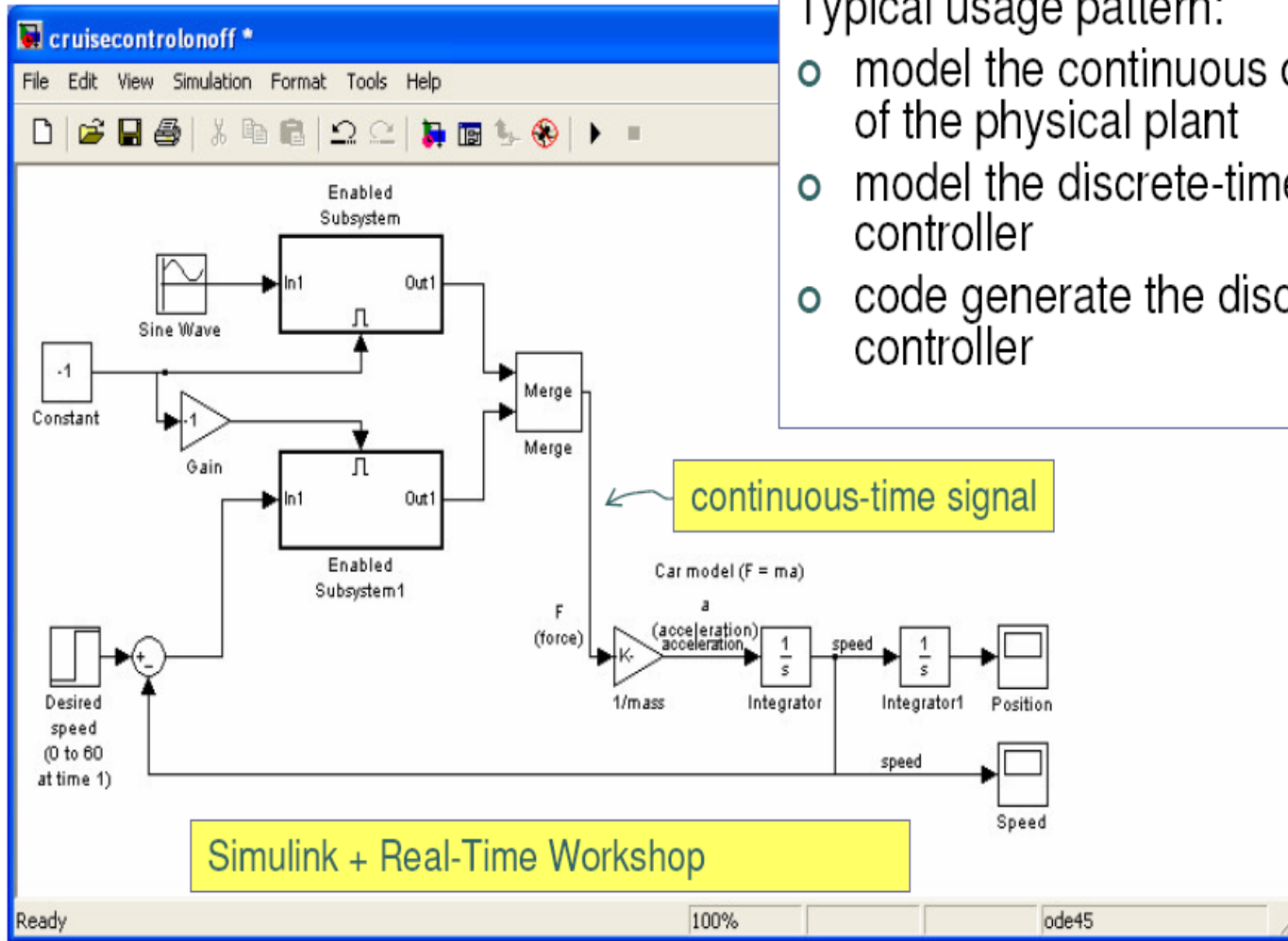
e.g. LabVIEW, Structured dataflow model of computation

Threads are Not the Only Possibility:

6th example: Continuous-Time Languages

Typical usage pattern:

- model the continuous dynamics of the physical plant
- model the discrete-time controller
- code generate the discrete-time controller





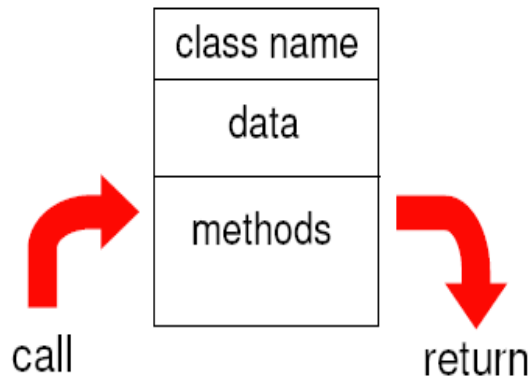
A Common Feature

- None is mainstream in computing.
- All are domain-specific.
- Emphasis on concurrent composition with determinism:
 - Composability
 - Security
 - Robustness
 - Resource management
 - Evolvability

Compared with message passing schemas, such as PVM, MPI, OpenMP, these impose stricter interaction patterns that yield determinism in the face of concurrency.

Many of These and Other Concurrent Component Models are *Actor Oriented*

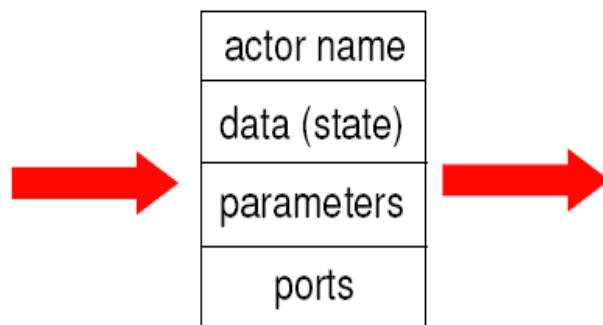
The established: Object-oriented:



What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:



Actors make things happen

What flows through an object is streams of data

Input data

Output data

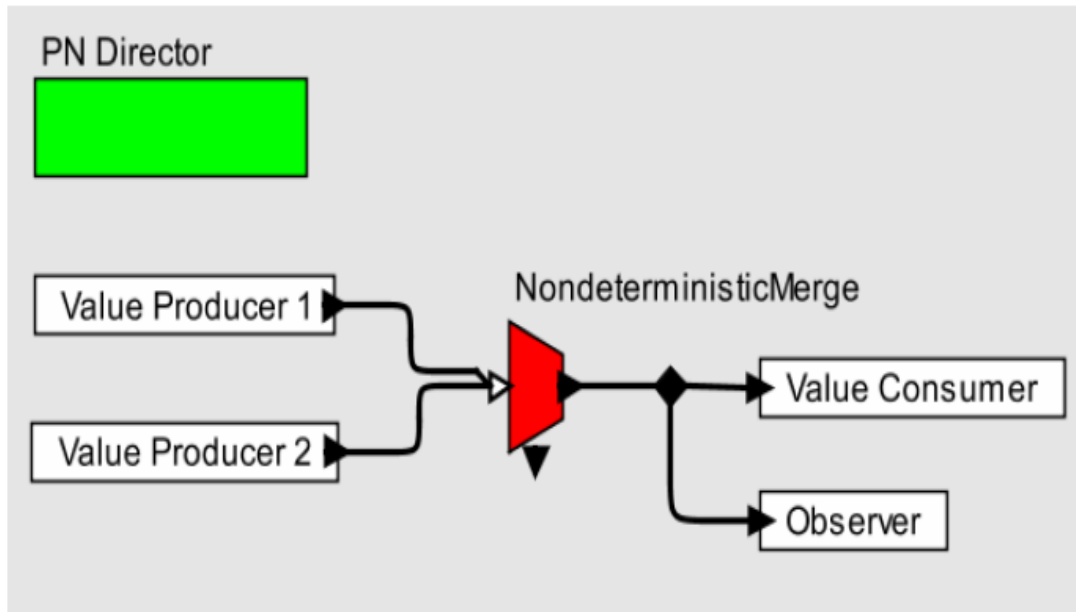


Recall the Observer Pattern

“The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

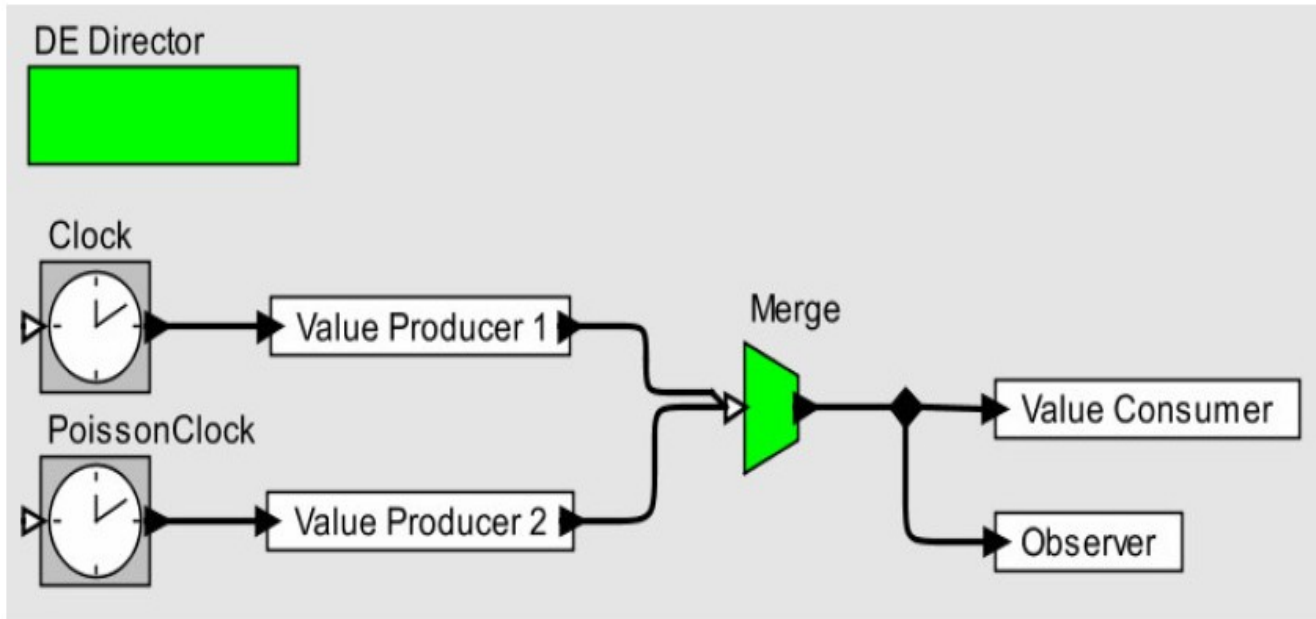
Observer Pattern using Process Networks

[Kahn 1974] Extended with
Nondeterministic Merge



Each actor is a process, communication is via streams, and the NondeterministicMerge explicitly merges streams nondeterministically.

Observer Pattern using Discrete Events



Messages have a (semantic) time, and actors react to messages chronologically. Merge now becomes deterministic.

So What is the Future of Embedded Software?



I don't know...

But I know what it should be:

Foundational architectures that combine software and models of physical dynamics with composition languages that have concurrency and time in a rigorous, composable, semantic framework.

Summary

Imperative languages

- Java
 - Java Micro edition
 - Real-time Java
- Message passing libraries
- CSP
- ADA
- Other languages: Pearl, Chill
- Threads revisited (Ed Lee)