Robotics and Embedded Systems

Technische Universität München

Specification and Modeling (1)

Kai Huang

Outline

- Model of Computation (MoC)
- StateCharts
- Data-Flow Models

Why Considering Specifications?

- The first step in designing Embedded System is to precisely tell what the system behavior should be.

> <u>Specification</u>: correct, clear and unambiguous description of the required system behavior.

- This can be extremely difficult
  - Increasing complexity of Embedded Systems
  - Desired behavior often not fully understood in the beginning
- If something is wrong with the specification, then it will be difficult to get the design right, potentially wasting a lot of time.
- How can we (correctly and precisely) capture systems behavior?

Model-Based Specifications

- Typically, we work with models of the system under design at different levels of abstraction
- Levels of abstraction alleviate the complexity problem of specification
  - Levels of abstraction has be discussed previously
- Models allow to reason about the systems under design, thereby identifying and correcting flaws in the specification

- What is a model?

Model

Definition: A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task. [Jantsch, 2004]

→Modeling means forgetting

- What are the requirements for Model-based Specification techniques for Embedded Systems?

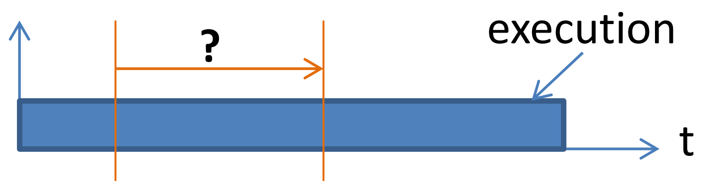
Requirements for Model & Spec. Techniques (1)

- Modularity
  - Systems specified as a composition of objects
- Represent hierarchy
  - Humans not capable to understand systems containing more than a few objects.
  - Behavioral hierarchy
    - Examples: statements->procedures->programs
  - Structural hierarchy
    - Examples: transistors->gates->processors->printed circuit boards
- Concurrency, synchronization and communication

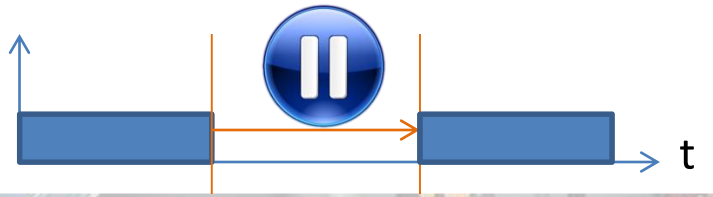
Requirements for Model & Spec. Techniques (2)

- Represent timing behavior/requirements
  - Timing is essential for embedded systems!
  - Four types of timing specs required [Burns, 1990]

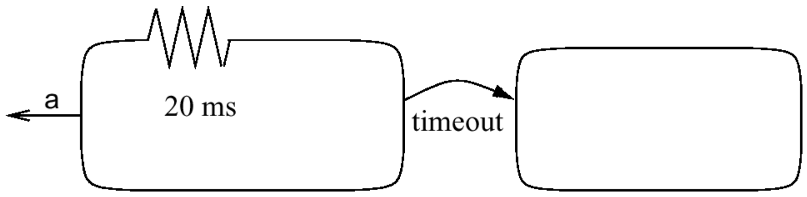
1. Measure elapsed time: Check, how much time has elapsed since last call



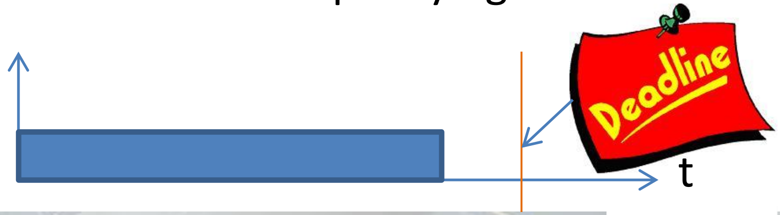
2. Means for delaying processes



3. Possibility to specify timeouts : Stay in a certain state a maximum time



4. Methods for specifying deadlines



Requirements for Model & Spec. Techniques (3)

- Represent state-oriented behavior
  - Required for reactive systems
- Represent dataflow-oriented behavior
  - Components send streams of data to each other
- No obstacles for efficient implementation

Models of Computation (MoC): Definition

- Components and an execution model for computations for each component
- Communication model for exchange of information between components
  - Shared memory
  - Message passing
  - …

There is NO model of computation that meets all specification requirements previously discussed
→ using compromises

Von Neumann Model

- An instruction set, a memory, and a program counter, is all need to execute whatever application one can dream of
  - Basically sequential execution

- Von Neumann model of computation does not match well with requirements for embedded system design
  - This model does not consider timing requirements and constraints, e.g.,
    - Timing cannot be described (instructions cannot be delayed or forced to execute at a specific time)
    - Timing deadlines cannot be specified for instructions or sequence of instructions
    - Timeouts cannot be specified for sequence of instructions
  - No way of concurrency

Thread-based Concurrency Models

"... threads as a concurrency model are a poor match for embedded systems. ... they work well only ... where best-effort scheduling policies are sufficient."

Edward Lee: Absolutely Positively on Time, IEEE Computer, July, 2005

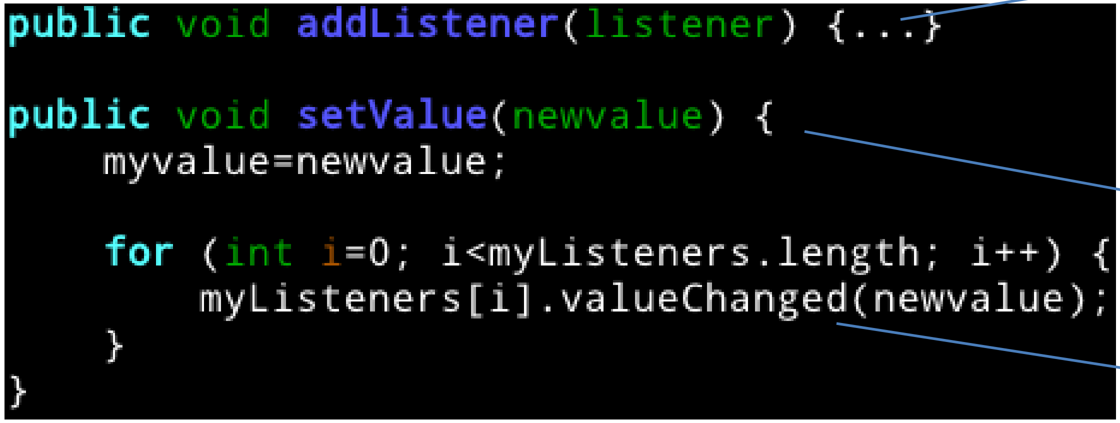
- Thread-based processing may access global variables
- We know from the theory of operating systems that
  - Access to global variables might lead to race conditions
  - To avoid these, we need to use mutual exclusion
  - Mutual exclusion may lead to deadlocks
  - Avoiding deadlocks is possible only if we accept performance penalties

An 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."

Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Addision-Wesley, 1995

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);
    }
}
```

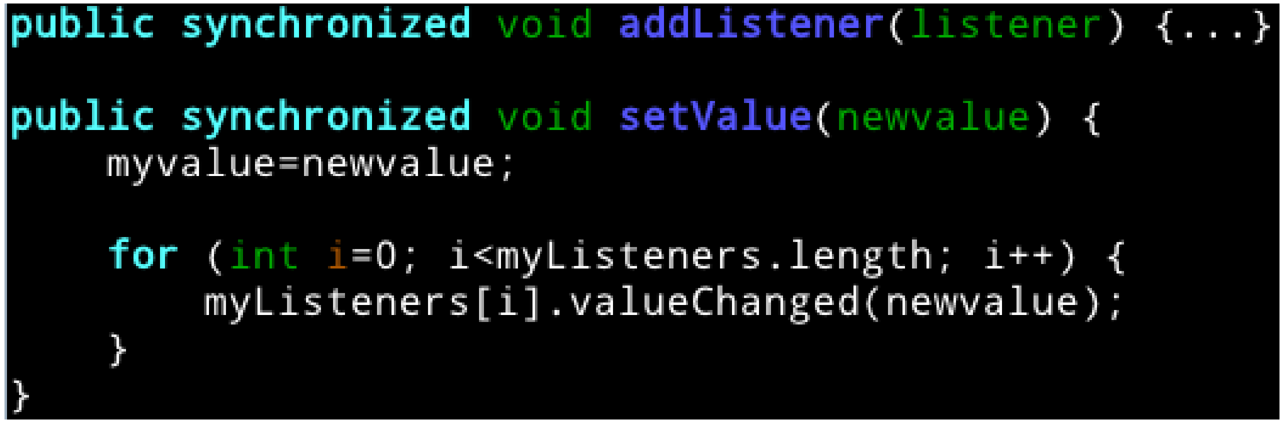
Add observer *listener* to data structure *myListeners*

Changes subject state

Changes observer state

What happens in a multi-threaded context?

Observer Pattern with Mutexes

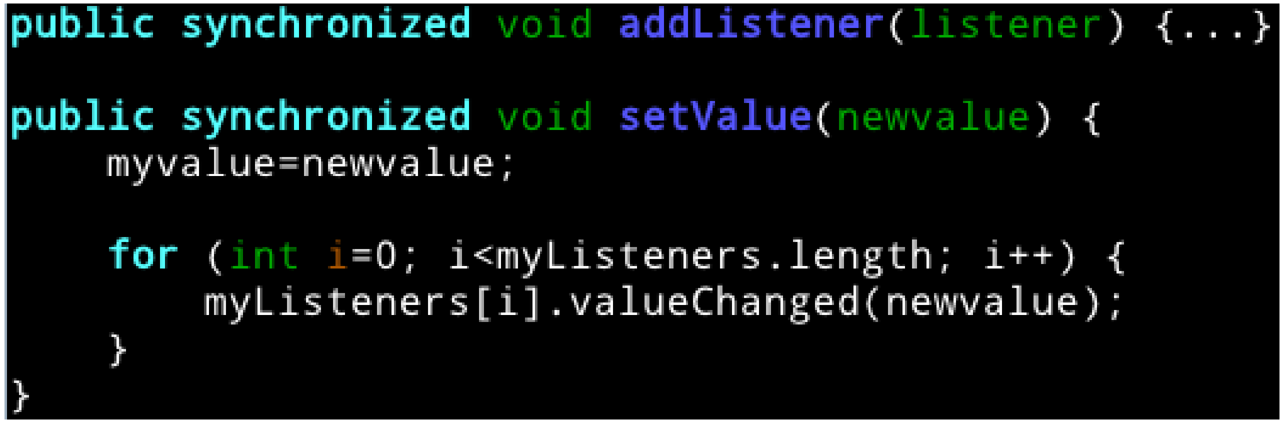


```
public synchronized void addListener(listener) {...}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;

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

Resolves race condition between addListener and setValue

Javasoft recommends against this! What’s wrong with it?

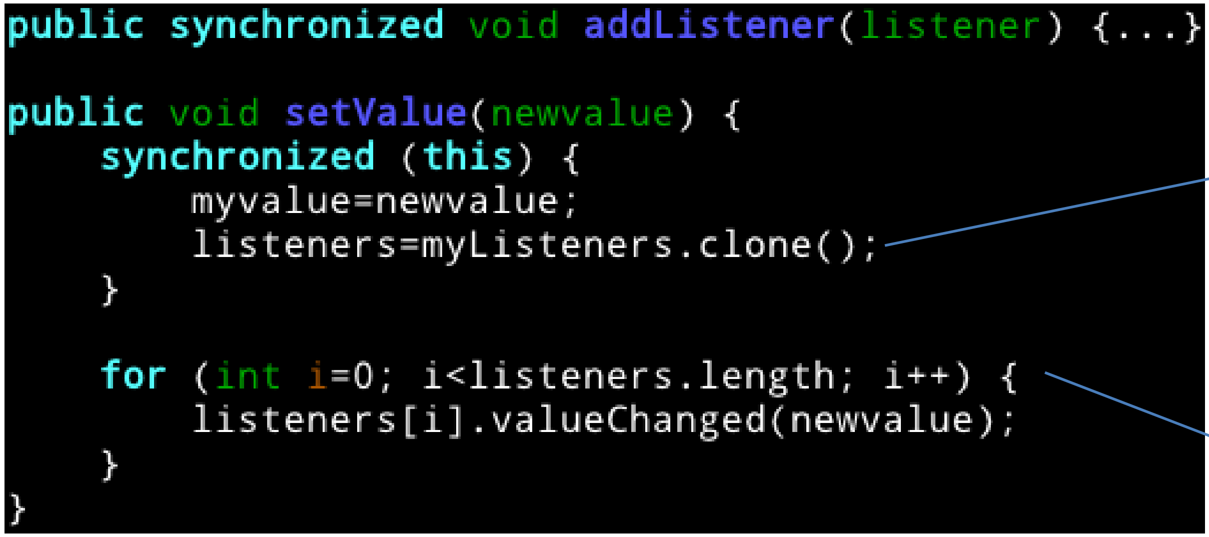
Mutexes are Minefields



```
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 (independent) object and stall. If the holder of that lock calls addListener(): deadlock!

Simple observer pattern gets complicated



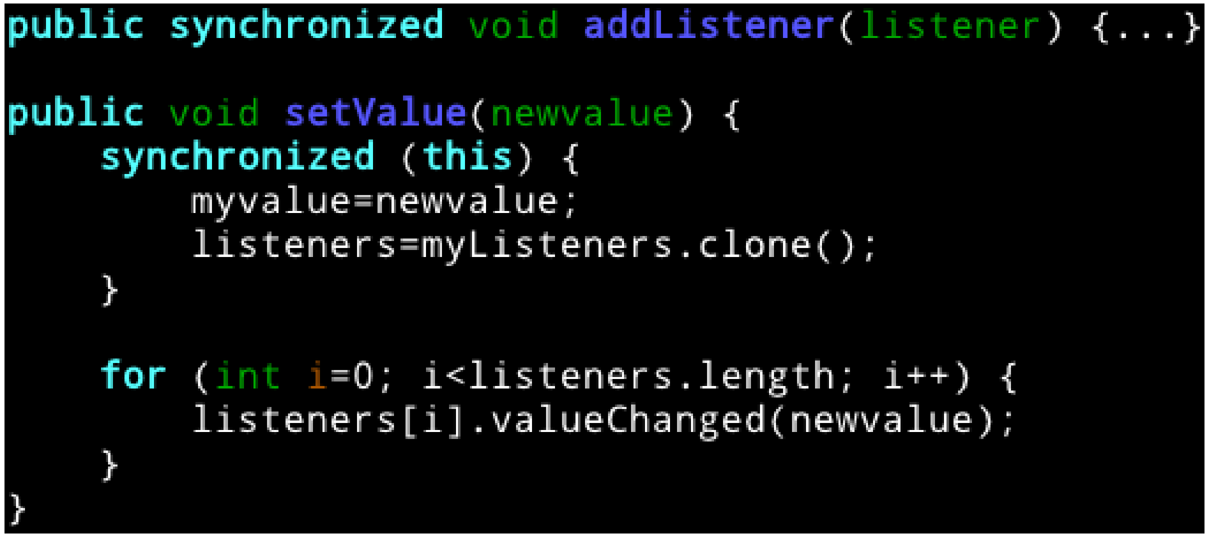
```
public synchronized void addListener(listener) {...}

public void setValue(newvalue) {
    synchronized (this) {
        myvalue=newvalue;
        listeners=myListeners.clone();
    }

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

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

notify each listener outside of the synchronized block to avoid deadlock

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. Listeners may finally have different values.

A Replicable Simple Example

```java
class A {
    synchronized void methodA(B b)  {
        b.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last()");
    }
}

class B {
    synchronized void methodB(A a) {
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside B.last()");
    }
}
```

```java
class Deadlock implements Runnable {
    A a = new A();
    B b = new B();

    // Constructor
    Deadlock() {
        Thread t = new Thread(this);
        int count = 20000;

        t.start();
        while (count-->0);
        a.methodA(b);
    }

    public void run() {
        b.methodB(a);
    }

    public static void main(String args[] ) {
        new Deadlock();
    }
}
```

```bash
#!/bin/bash

for (( c=1; c<=100; c++ ))
do
   echo "$c times"
   java Deadlock
done
```

Another latest article:
http://javaeesupportpatterns.blogspot.ca/2013/01/java-concurrency-hidden-thread-deadlocks.html

Problems with Thread-Based Concurrency

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

© Ed. Lee, Berkeley
Artemis Conference
Graz, 2007

- Search for non-thread-based models: which are the requirements for appropriate specification techniques?

The Bottom Line Is

When specifying and designing Embedded Systems we should search for and use NON-thread-based, NON-von-Neumann Models of Computation.

- Finding appropriate model to capture an embedded system's behavior is an important step
  - Model shapes the way we think of the system
  - For control-dominated and reactive systems
    - State-based models are appropriate, monitor control inputs and set control outputs
  - For data-dominated systems
    - Dataflow models are appropriate, transform input data streams to output data streams