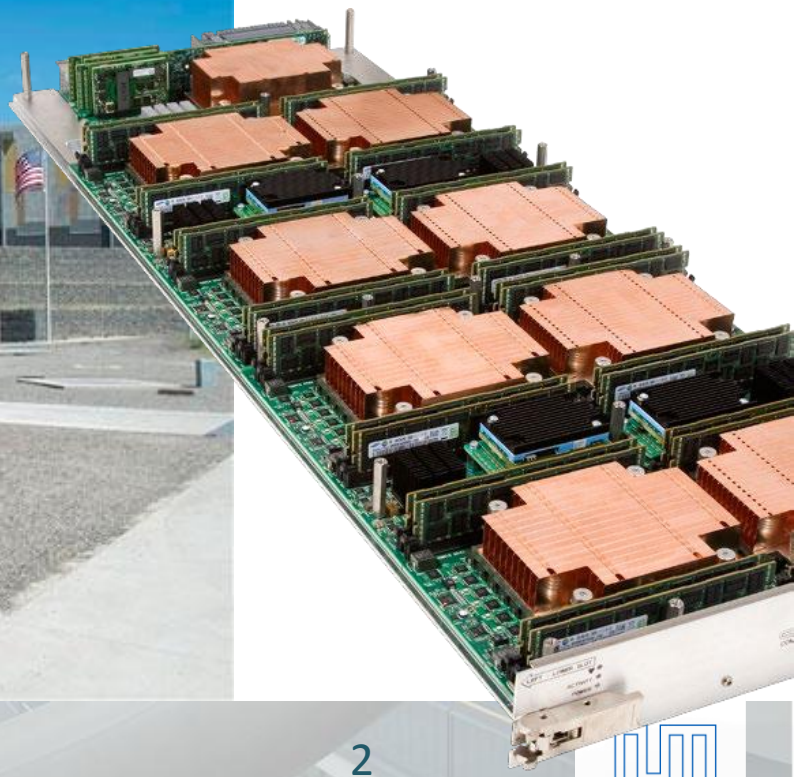
# Specification and Modeling

**Kai Huang**

# News

- 70M French phones/month,  60M Spain phones/month, …
- Cray XC30 series (64-bit Intel® Xeon® processor E5 family)
- The completed facility is expected to require 65 megawatts, costing about $40 million per year

# 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.

> Specification: 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?

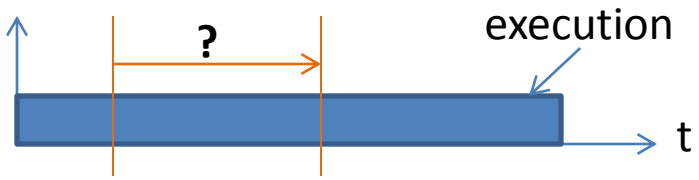
# Requirements for Model & Spec. Techniques (1)

- 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

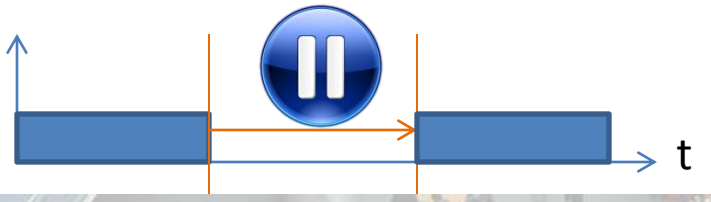
# Requirements for Model & Spec. Techniques (2)

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

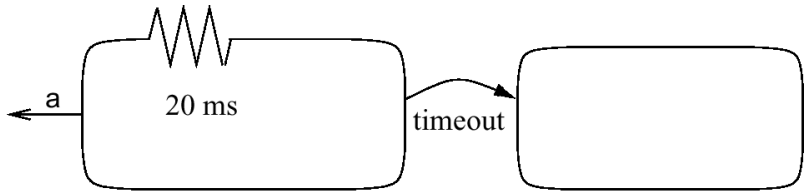
1. Measure elapsed time: Check, how much time has elapsed since last call



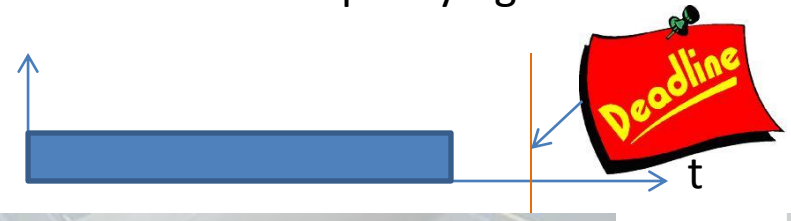

2. Means for delaying processes




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

```
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()");
    }
}
```

```
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();
    }
}
```

```
#!/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

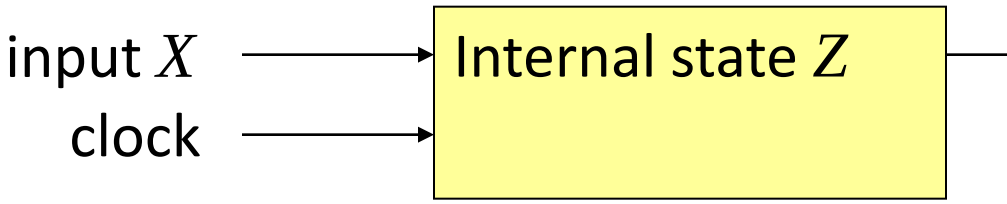
When specifying and designing Embedded Systems we should search for and use NON-thread-based, NON-von-Neumann Models of Computation.

- 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

# Outline

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

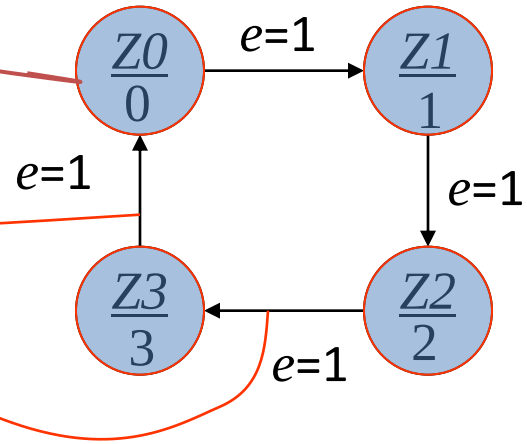
# Classical Automata




Next state $Z^+$ computed by function $\delta$
Output computed by function $\lambda$

Moore + Mealy automata=finite state machines (FSMs)

- Moore-automata:
  $Y = \lambda\,(Z)$; $Z^+ = \delta\,(X, Z)$
- Mealy-automata
  $Y = \lambda\,(X, Z)$; $Z^+ = \delta\,(X, Z)$

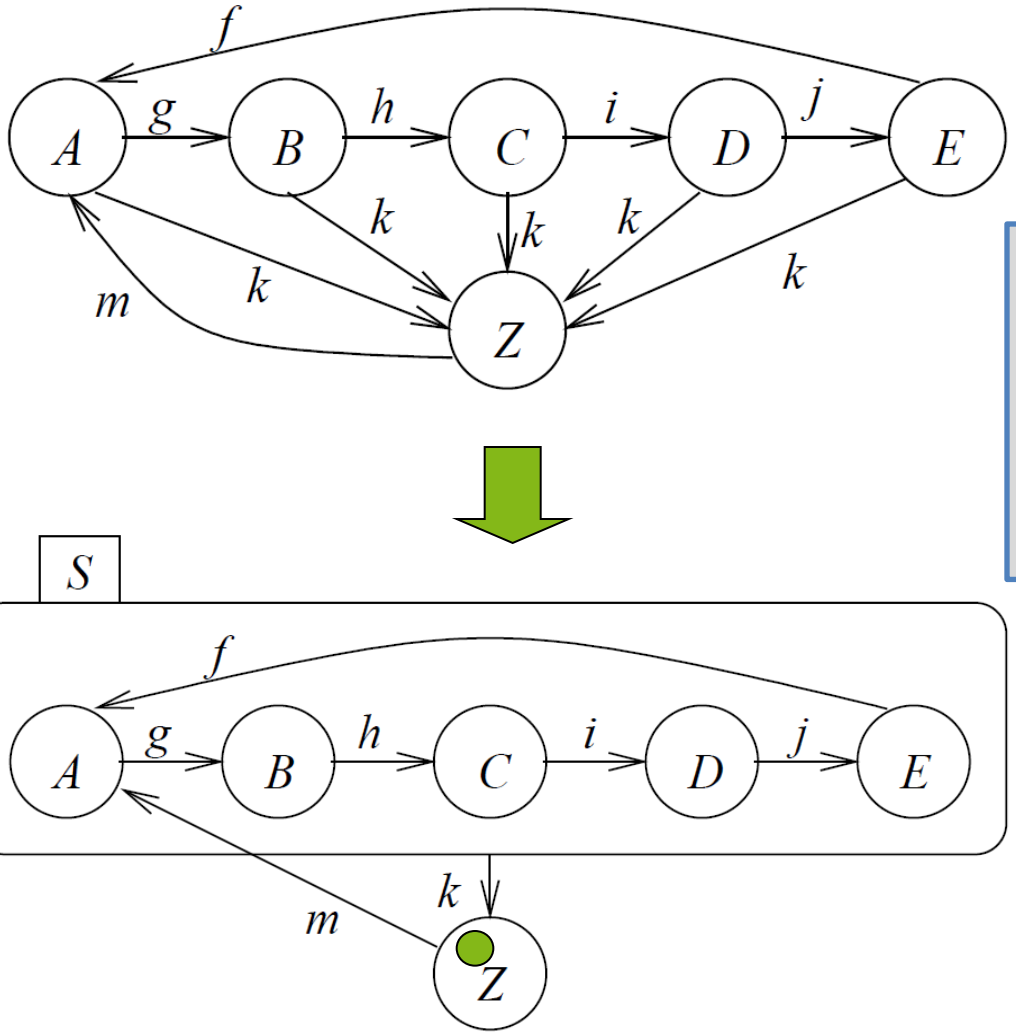


# StateCharts

Classical automata NOT useful for complex systems (complex graphs cannot be understood by humans).

→ Introduction of hierarchy→StateCharts [Harel, 1987]

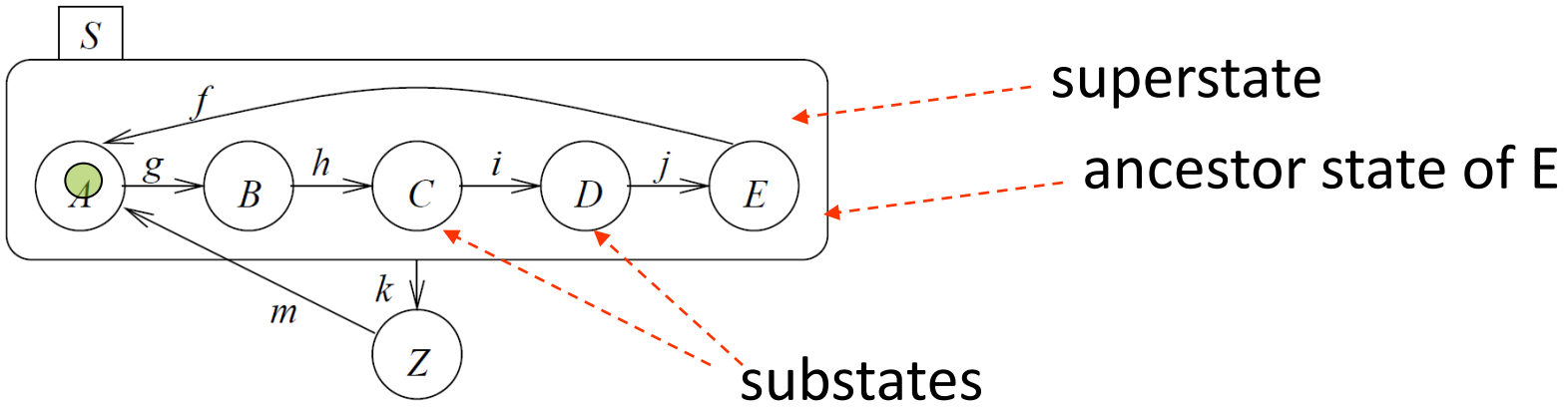
# Introducing Hierarchy




FSM will be **in** exactly one of the substates of S if S is **active**
(either in A or in B or ..)

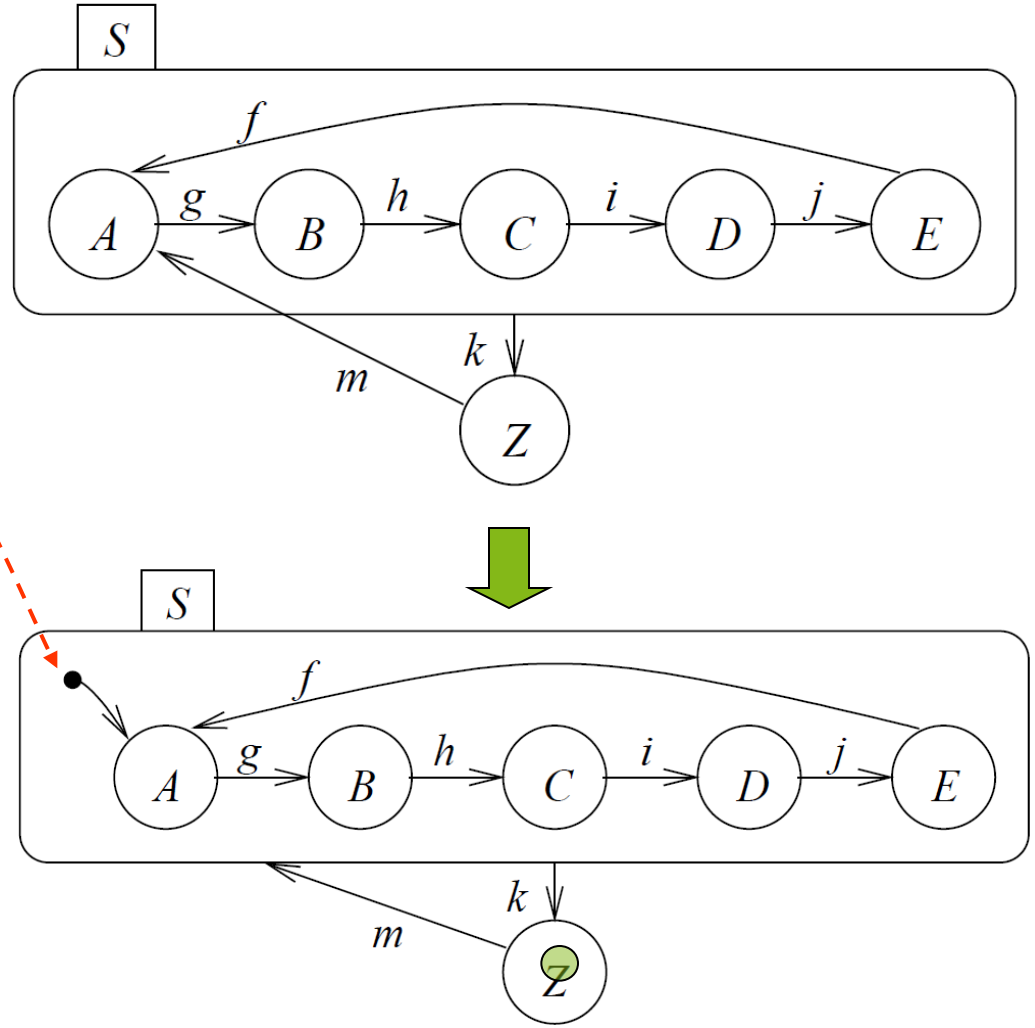
# Definitions

- Current states of FSMs are also called **active** states.
- States which are not composed of other states are called **basic states.**
- States containing other states are called **super-states**.
- For each basic state s, the super-states containing s are called **ancestor states**
- Super-states $S$ are called **OR-super-states**, if exactly one of the sub-states of $S$ is active whenever $S$ is active.

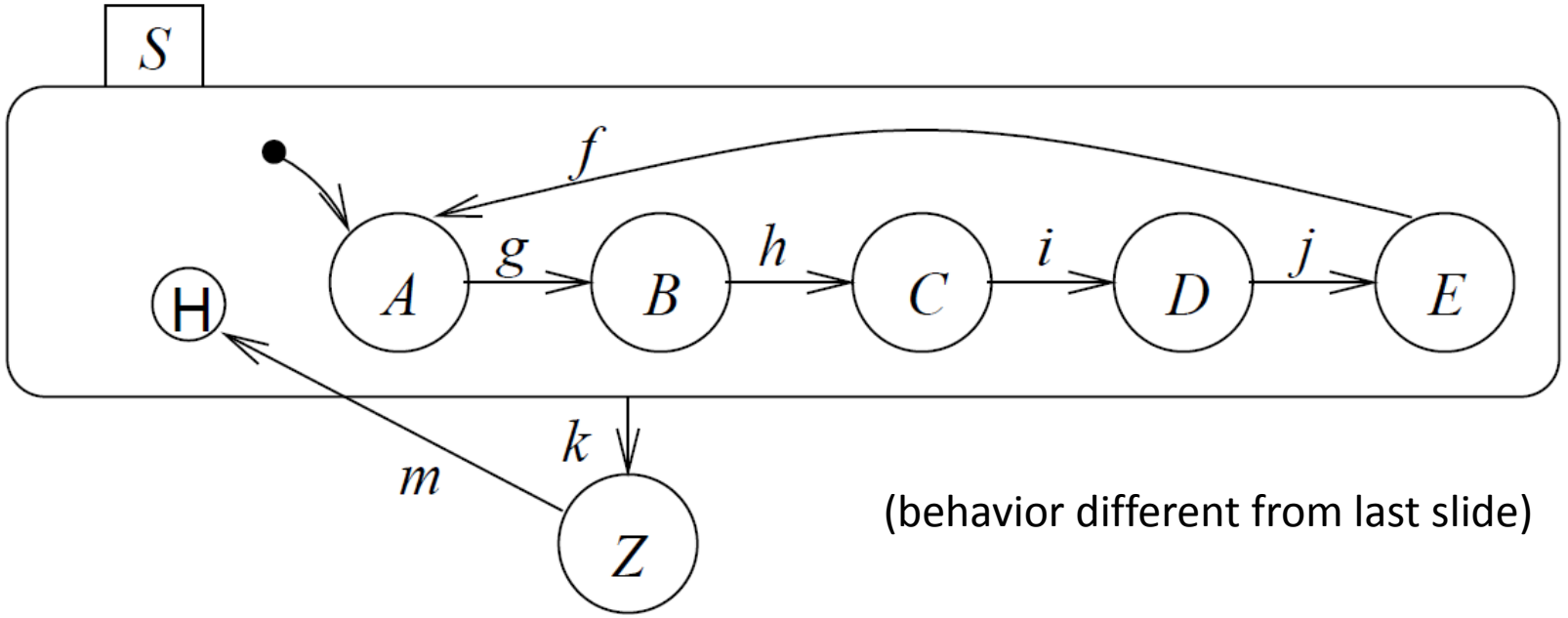


# Default State Mechanism

- Try to hide internal structure from outside world!
  - Default state
- Filled circle indicates sub-state entered whenever super-state is entered.
- Not a state by itself!



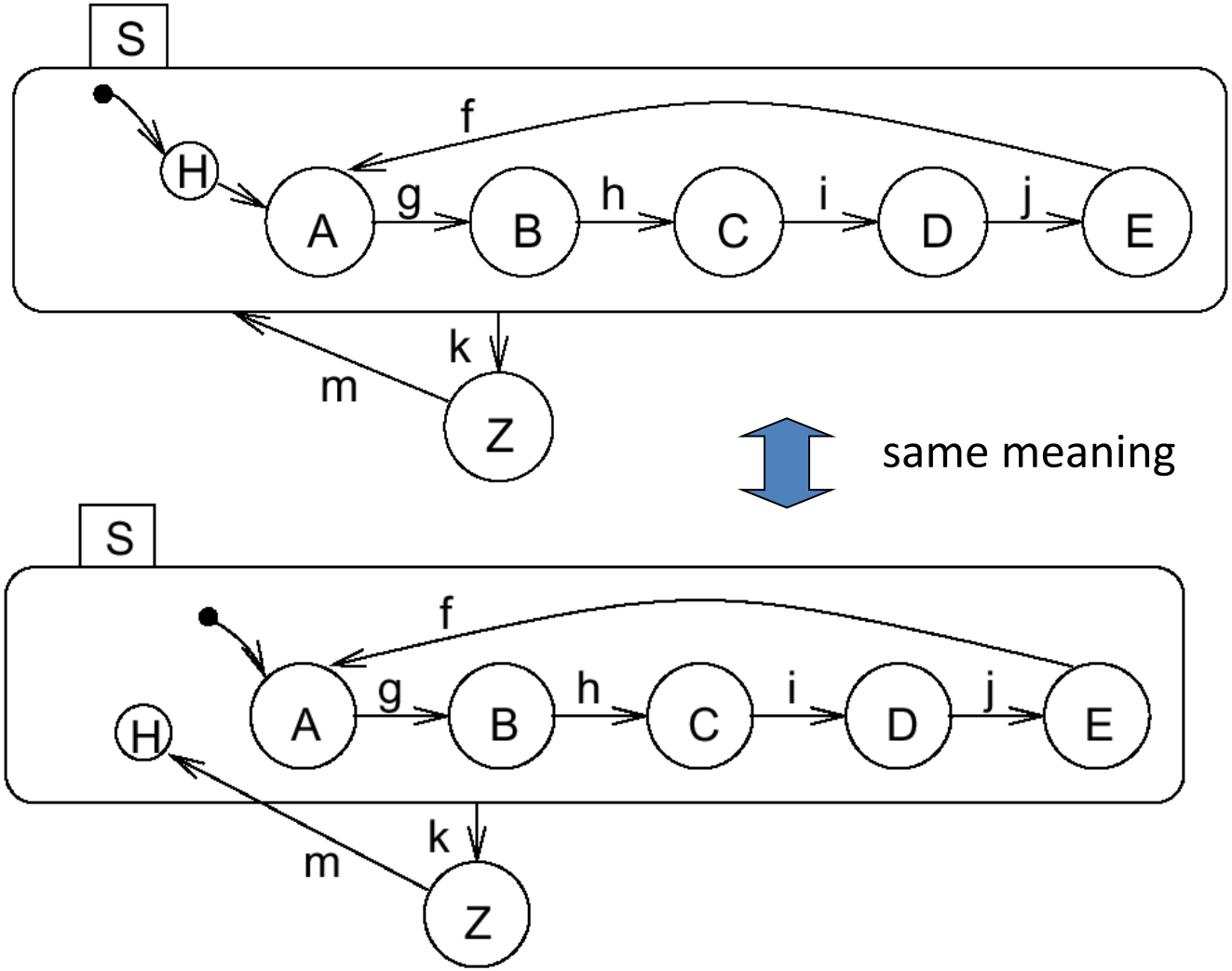
# History Mechanism




(behavior different from last slide)

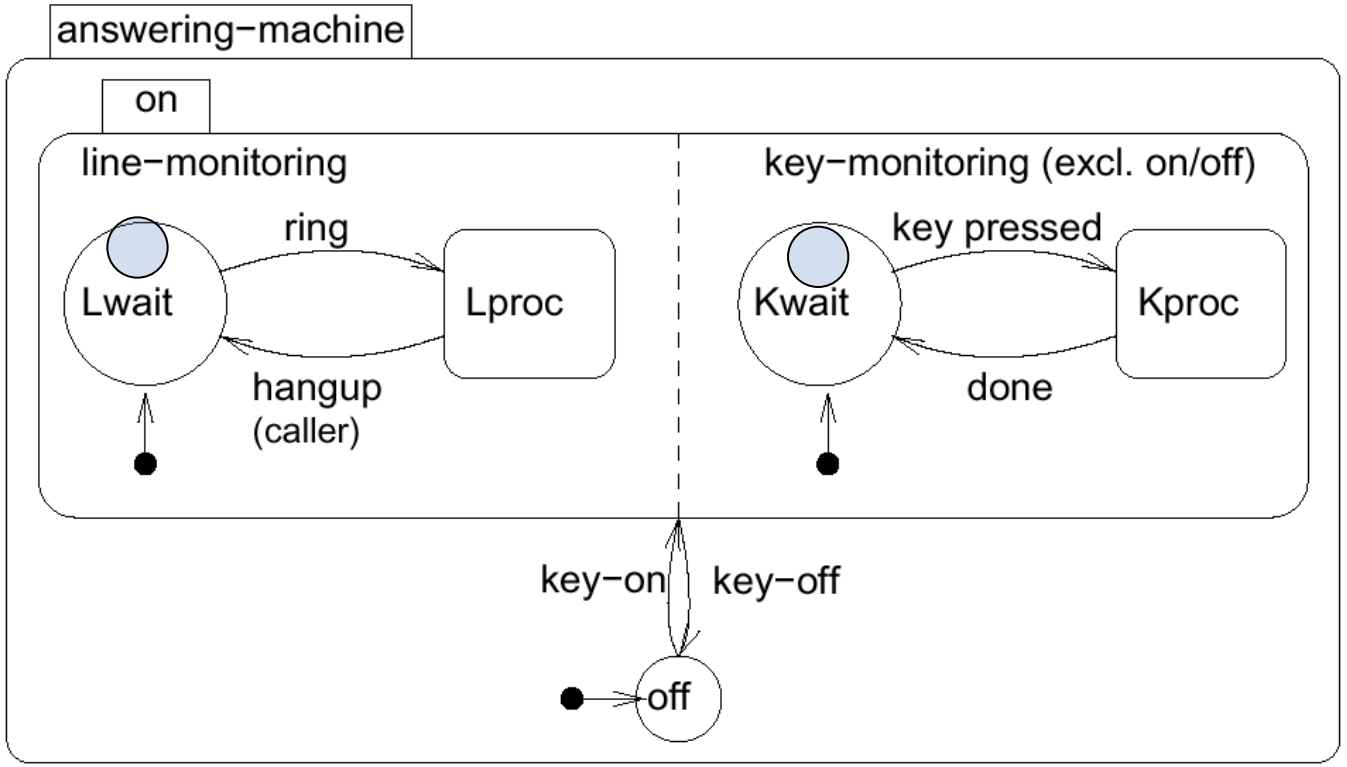
- For input m, S enters the state it was in before S was left (can be A, B, C, D, or E). If S is entered for the very first time, the default mechanism applies.
- History and default mechanisms can be used hierarchically.

# Combining History and Default State

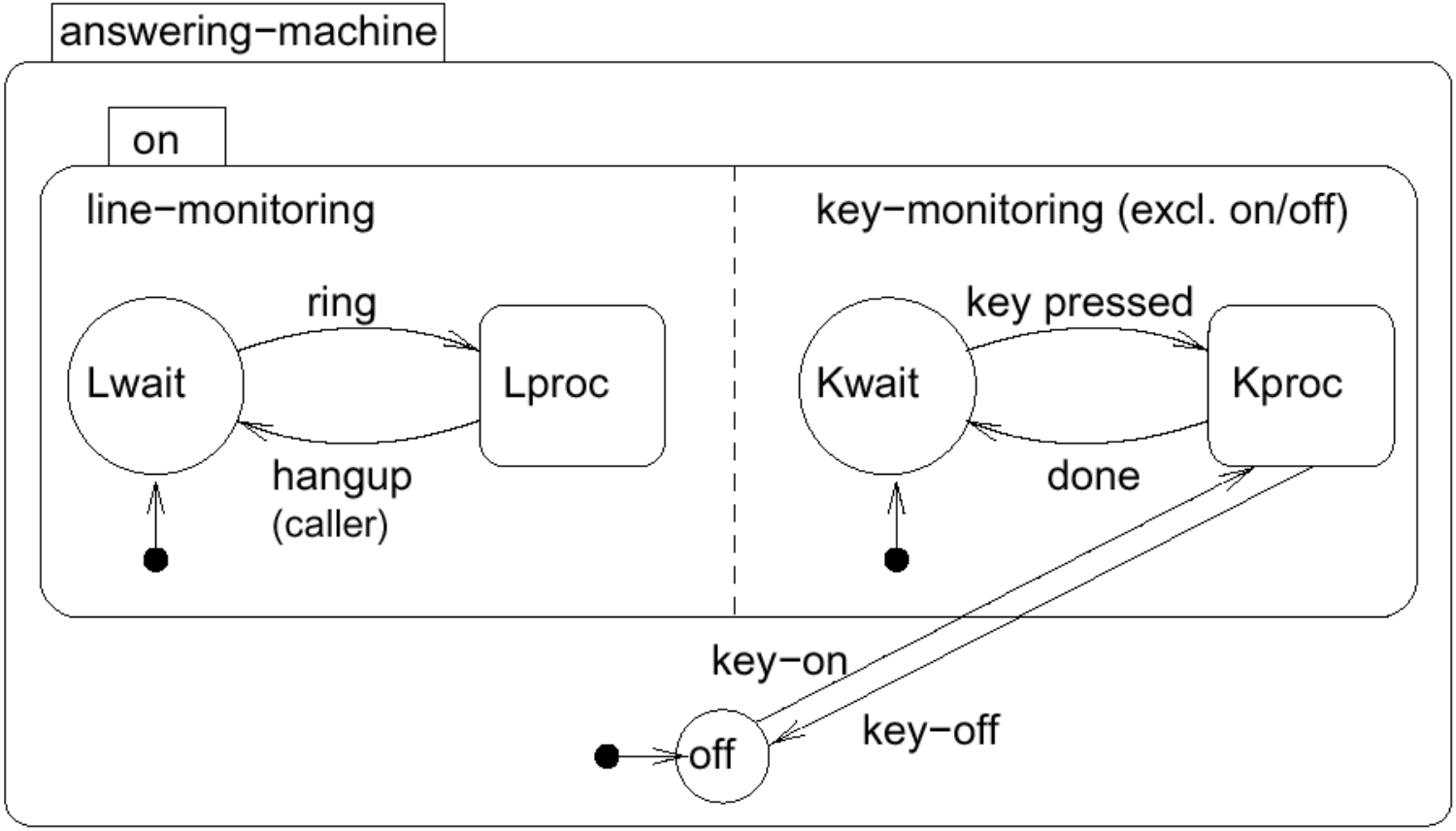


# Concurrency

- Convenient ways of describing concurrency are required.
- **AND-super-states**: FSM is in **all** (immediate) sub-states of a super-state; Example:

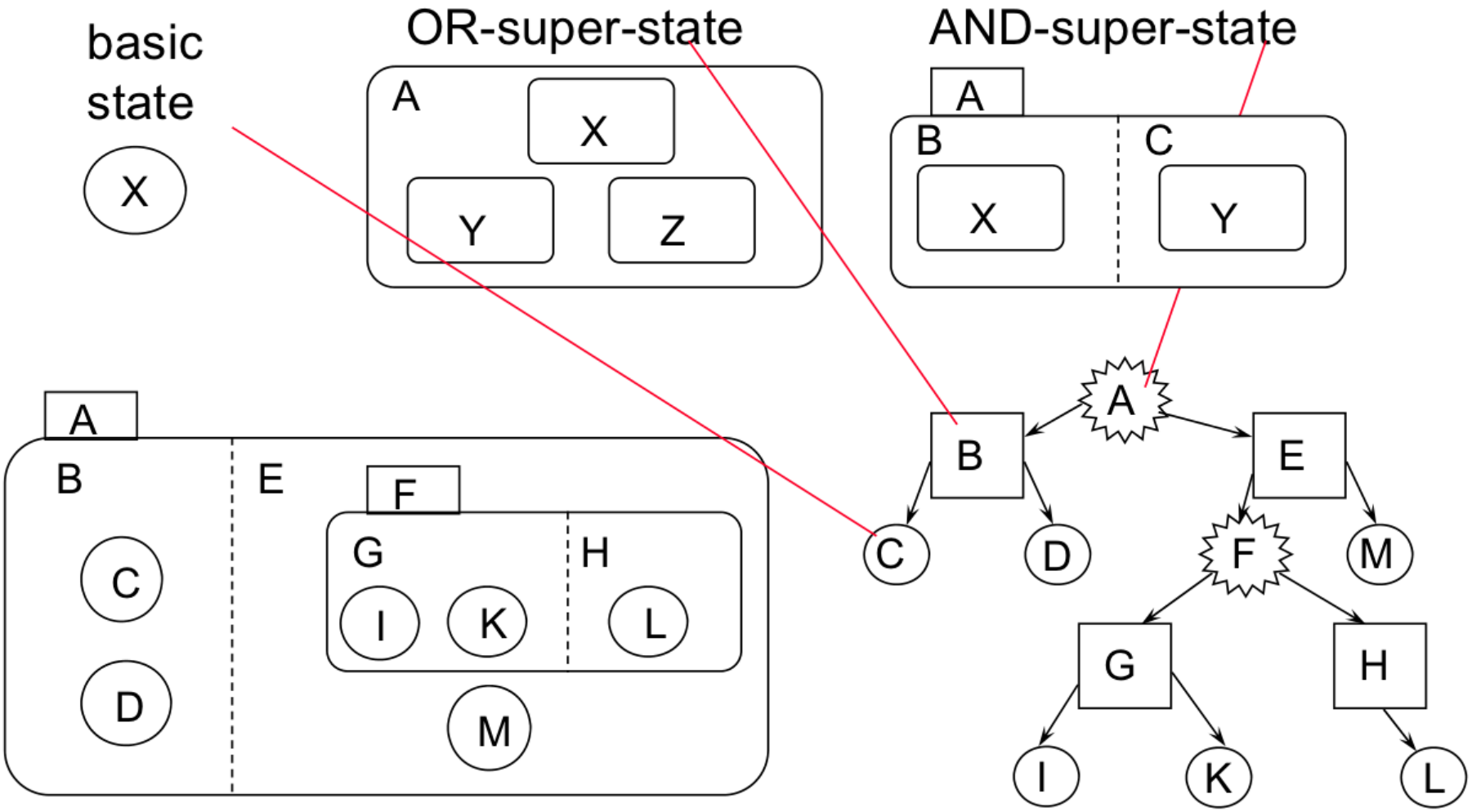


# Entering and Leaving AND-super-states



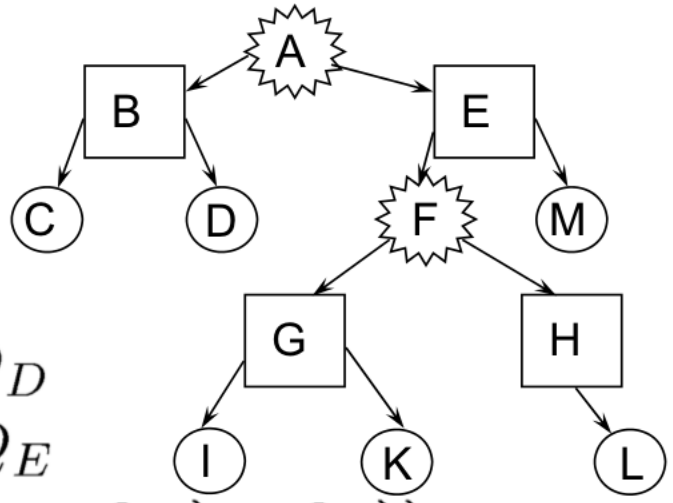

- Line-monitoring and key-monitoring are entered and left, when service switch is operated.

# Tree Representation of State Sets



# Computation of State Sets

- Computation of state sets by traversing the tree from leaves to root:
  - basic states: state set = state
  - OR-super-states: state set = union of children
  - AND-super-states: state set = Cartesian product of children




$$Q_H = Q_L,\ Q_G = Q_I \cup Q_K$$
$$Q_F = Q_G \times Q_H,\ Q_B = Q_C \cup Q_D$$
$$Q_E = Q_F \cup Q_M,\ Q_A = Q_B \times Q_E$$
$$Q_A = (Q_C \cup Q_D) \times (Q_M \cup ((Q_I \cup Q_K) \times Q_L))$$

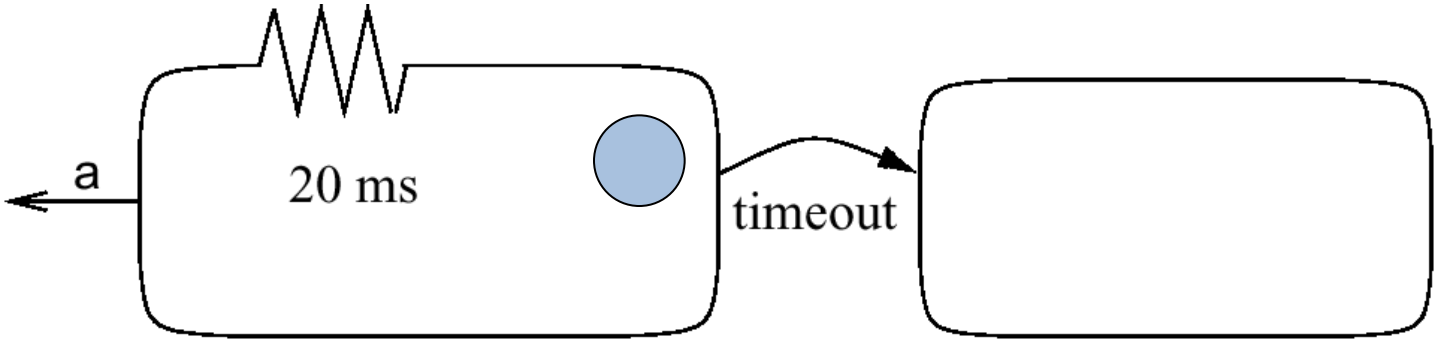
# Types of States

- In StateCharts, states are either
  - Basic states, or
  - AND-super-states, or
  - OR-super-states.

- Sable state: there are no generated events and no enabled compound transition or static action

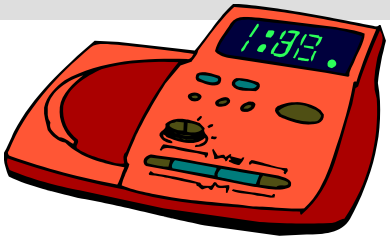
# Timers

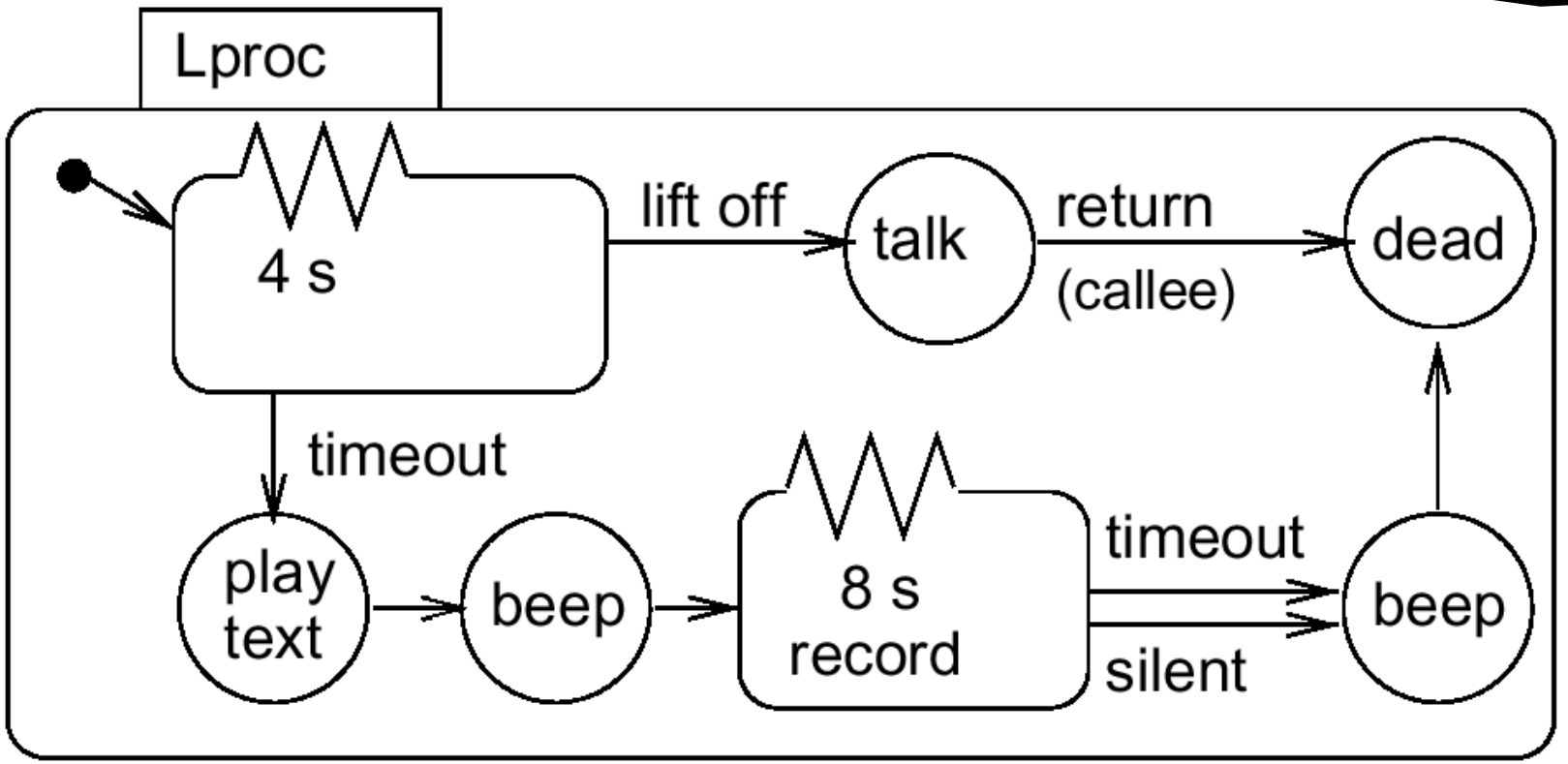
- Since time needs to be modeled in embedded systems, timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.




If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

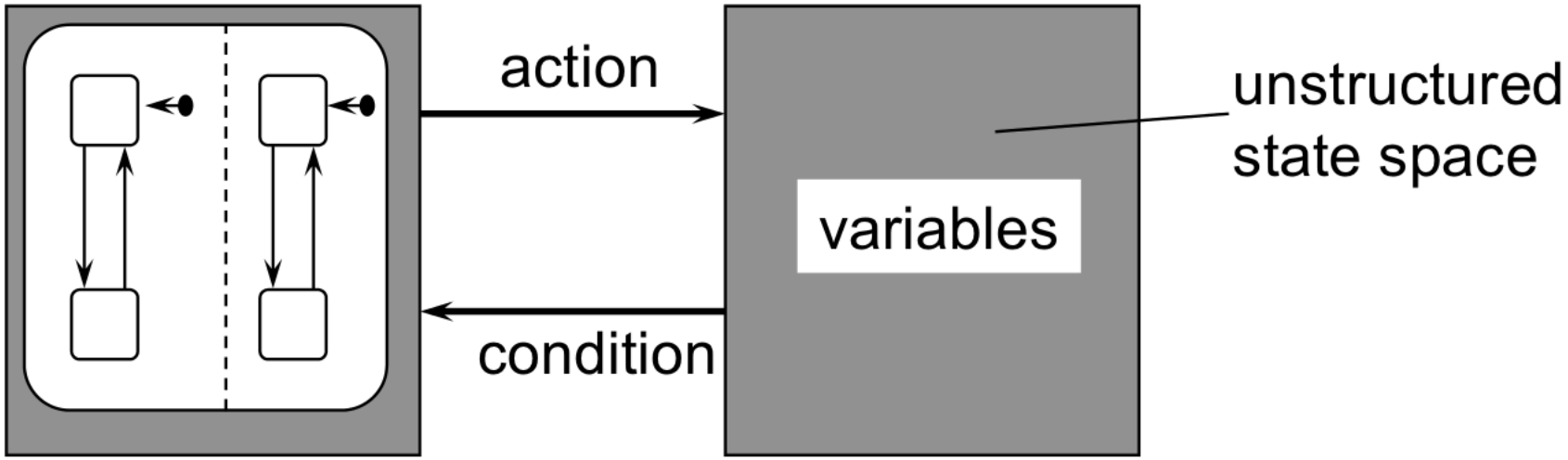
# Using Timers in an Answering Machine



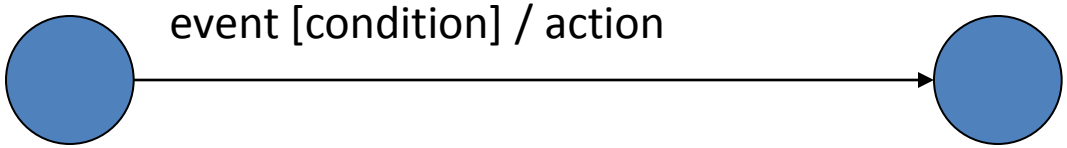


# Representation of Computations

- Besides states, arbitrary many other variables can be defined. This way, not all states of the system are modeled explicitly.
- These variables can be changed as a result of a state transition ("action"). State transitions can be dependent on these variables ("condition" ).



# General Form of Edge Labels



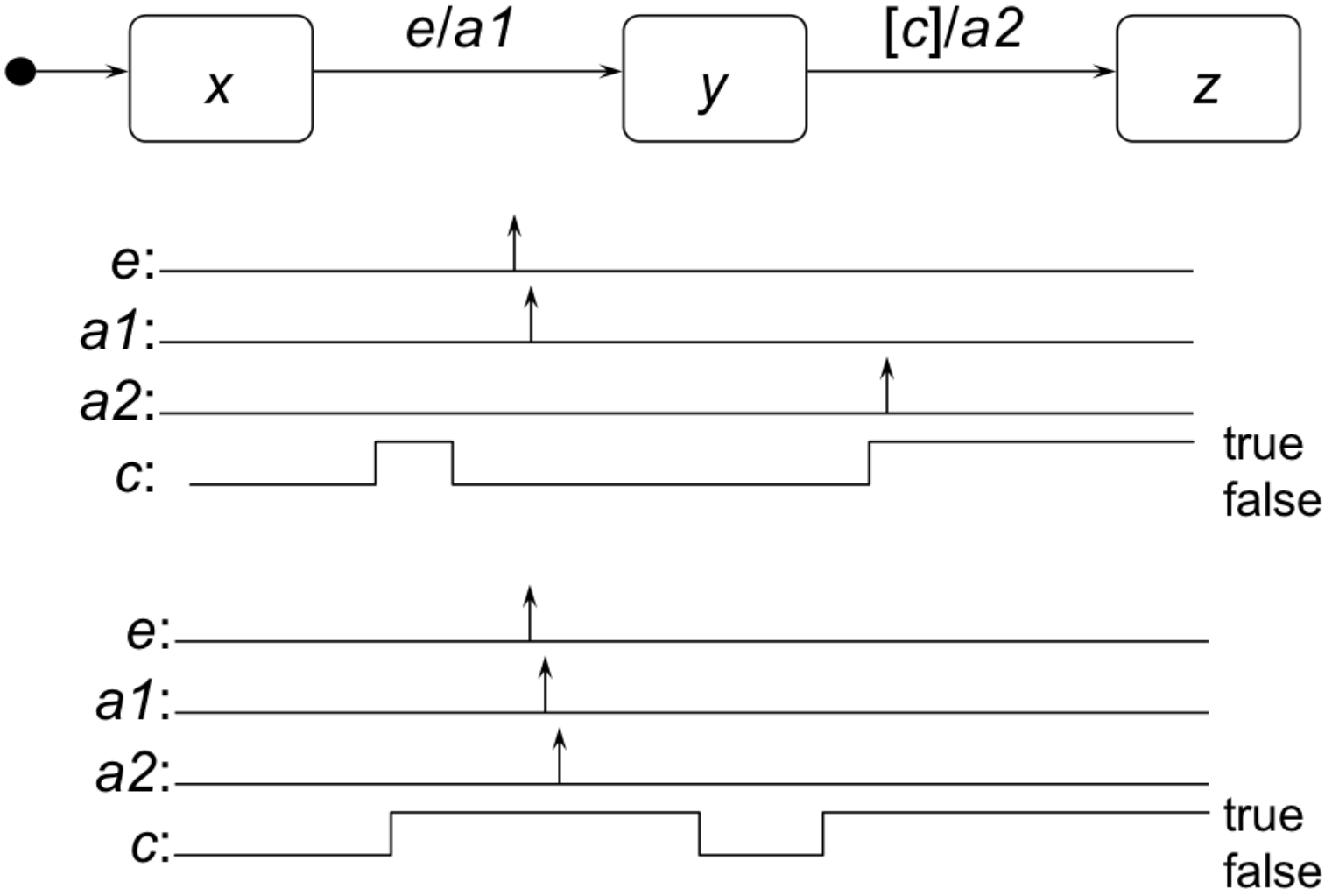

- Events:
  - Exist only until the next evaluation of the model
  - Can be either internally or externally generated
- Conditions:
  - Refer to values of variables that keep their value until they are reassigned
- Actions:
  - Can either be assignments for variables or creation of events

- Example:
  - service-off [a <= 7] / service:=0

# Events and Actions

- "event" can be composed of several events:
  - (e1 and e2) : event that corresponds to the simultaneous occurrence of e1 and e2.
  - (e1 or e2) : event that corresponds to the occurrence of either e1 or e2, or both.
  - (not e) : event that corresponds to the absence of event e.
- "action" can also be composed:
  - (a1; a2) : actions a1 and a2 are executed in parallel.
- All events, states and actions are globally visible.

# Example



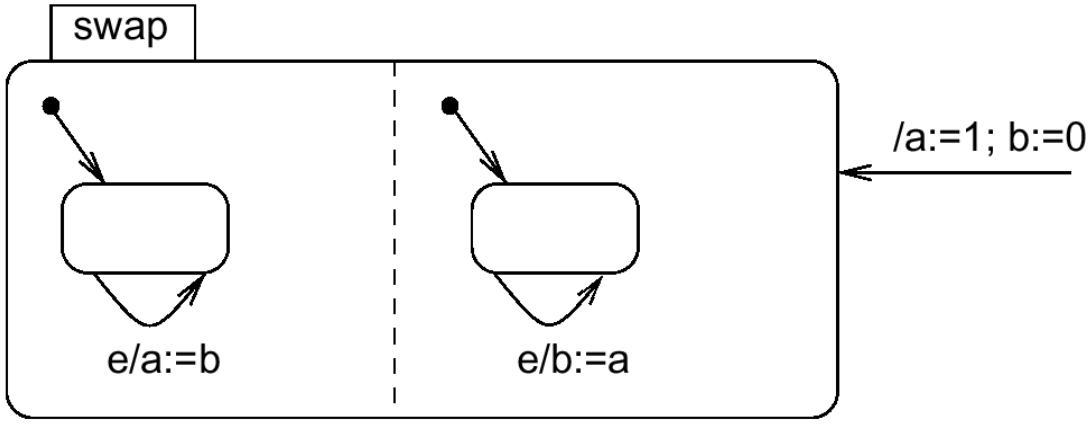
# The StateCharts Simulation Phases

- How are edge labels evaluated?

- Three phases:
  1. Evaluate effect of external changes on events and conditions
  2. Compute set of transitions to be made in the current step and right hand sides of assignments
  3. Activate transitions, assign new values to variables

Separation into phases 2 and 3 guarantees deterministic and reproducible behavior

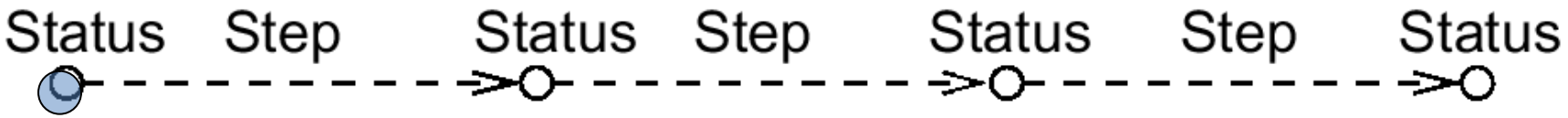
# Example




- In phase 2, variables a and b are assigned to temporary variables. In phase 3, these are assigned to a and b. As a result, variables a and b are swapped.
- In a single phase environment, executing the left state first would assign the old value of b (=0) to a and b. Executing the right state first would assign the old value of a (=1) to a and b. The execution would be non-deterministic.

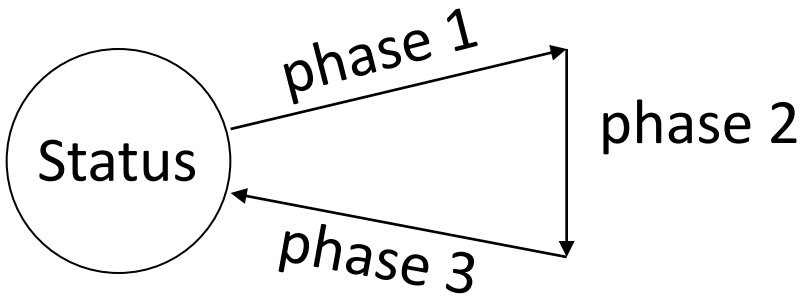
# Steps

- Execution of a StateMate model consists of a sequence of (status, step) pairs



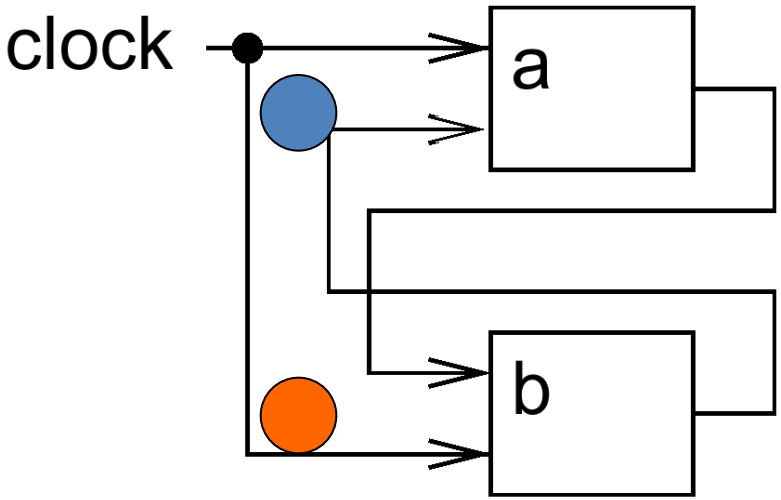

Status= values of all variables + set of events + current time

Step = execution of the three phases (StateMate semantics)




Other implementations of StateCharts do not have these 3 phases (and hence are nondeterministic)!

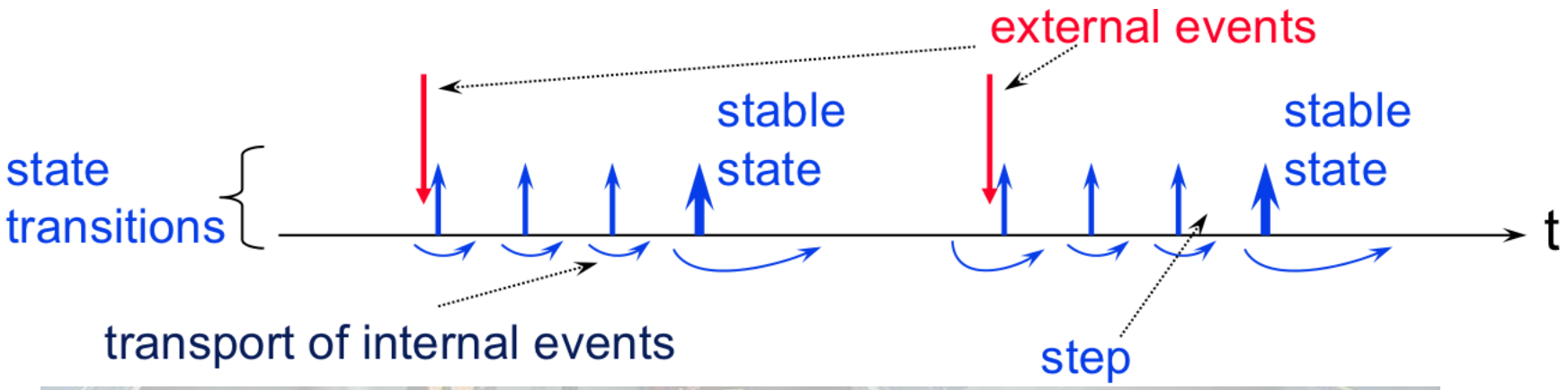
# Reflects Model of Clocked Hardware



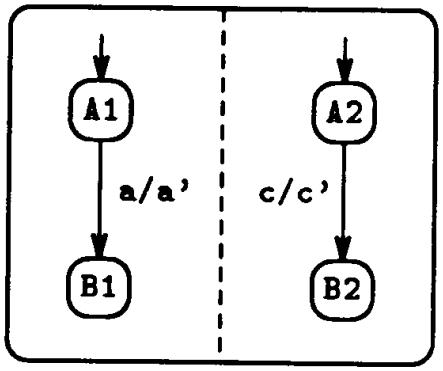

- In an actual clocked (synchronous) hardware system, both registers would be swapped as well.
- Same separation into phases found in other languages as well, especially those that are intended to model hardware.

# More on Semantics of StateCharts

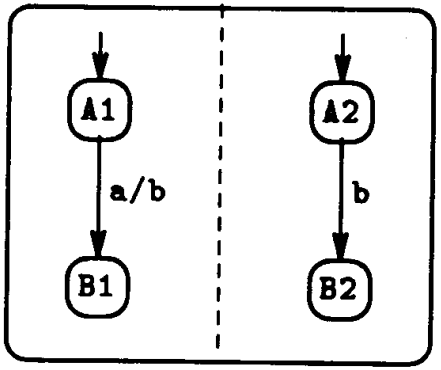
- Unfortunately, there are several time-semantics of StateCharts in use. This is another possibility:
  - A step is executed in arbitrarily small time.
  - Internal (generated) events exist only within the next step.
  - Difference: External events can only be detected after a stable state has been reached.



# Examples



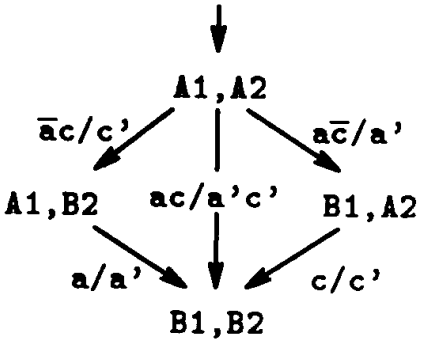

(a)



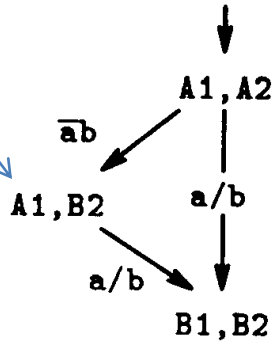

(b)

state diagram:

stable states

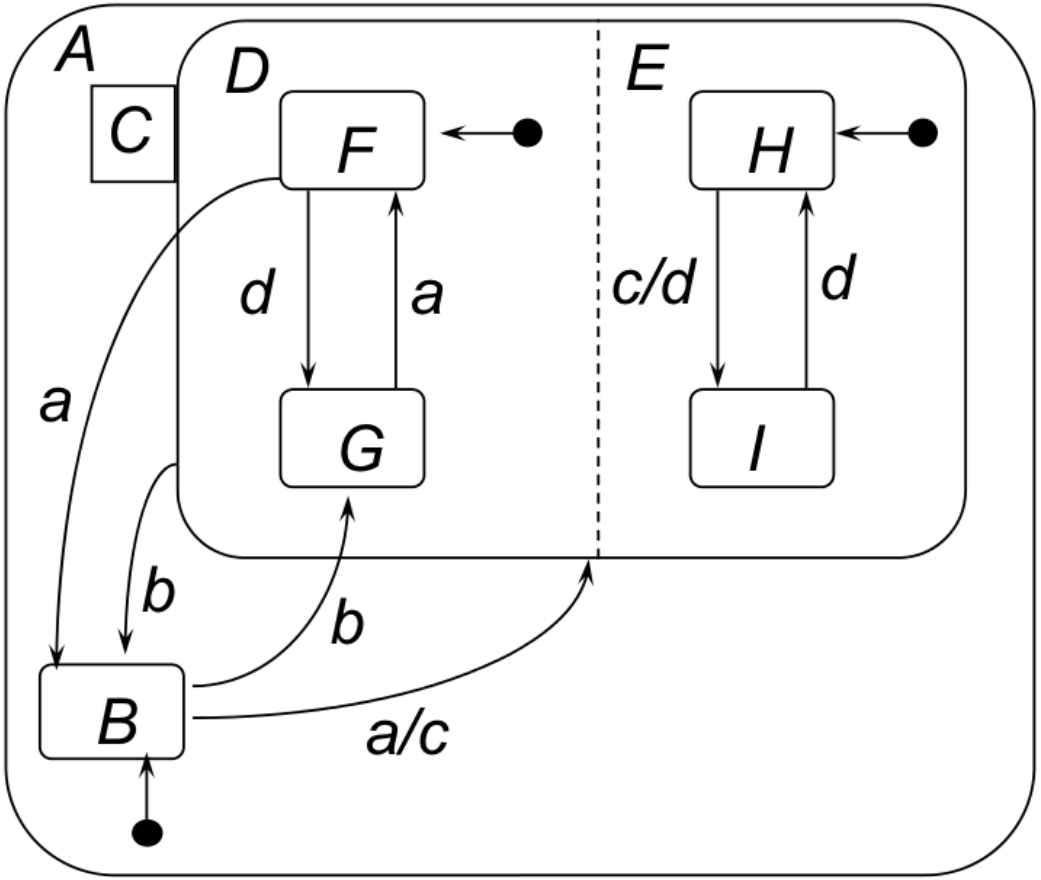



(a)

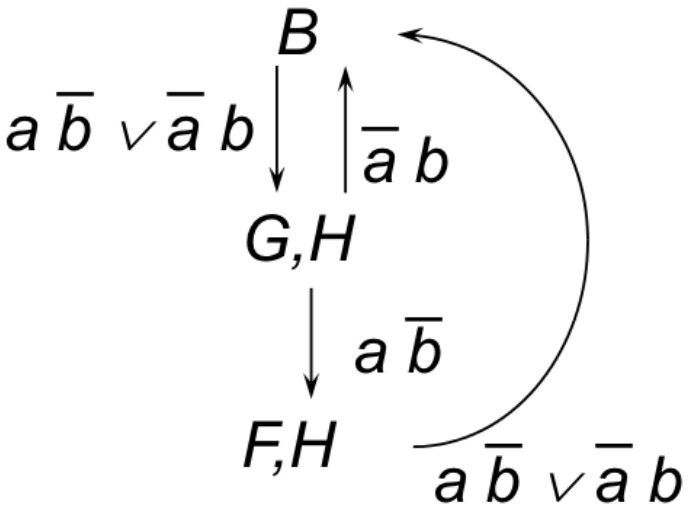



(b)

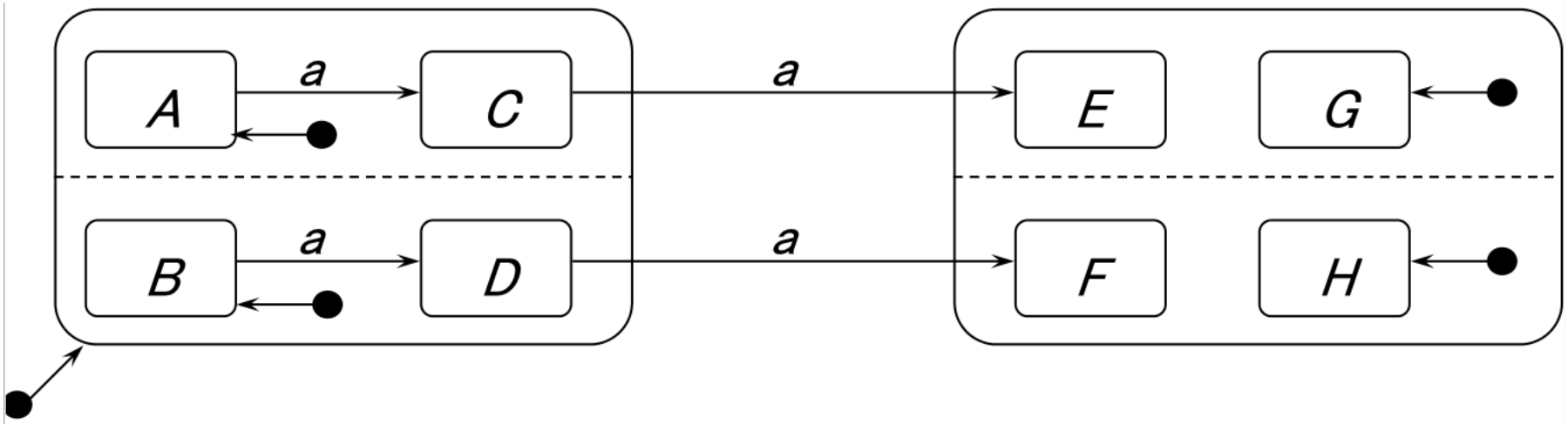
# Example



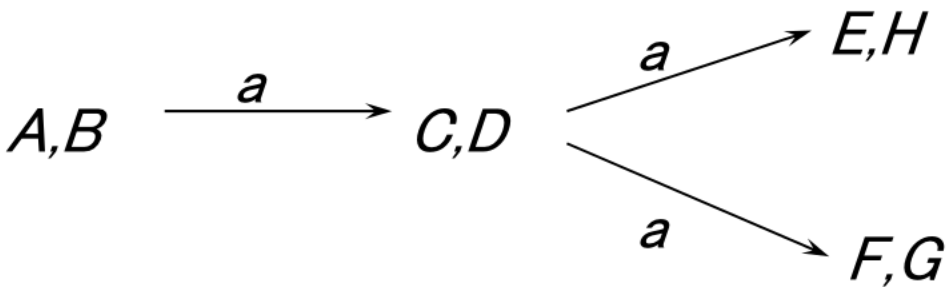

state diagram (only stable states are represented, only a and b are external):



# Example




state diagram:




Nondeterministic !

# Evaluation of StateCharts (1)

- Pros:
  - Hierarchy allows arbitrary nesting of AND- and OR-super states.
  - Semantics defined in a follow-up paper to original paper.
  - Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
  - Available “back-ends“ translate StateCharts into C or VHDL, thus enabling software or hardware implementations.

# Evaluation of StateCharts (2)

- Cons:
  - Generated C programs frequently inefficient,
  - Not useful for distributed applications,
  - No description of non-functional behavior,
  - No object-orientation,
  - No description of structural hierarchy.

To be continue