JiJ Simulator – Testing Java Concurrent Software

Jaroslav Kačer
jkacer@kiv.zcu.cz

University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

DSS and MFF Seminar
2004-10-22

Presentation Outline

1. Project goals
2. Main principles
3. Development process
4. Architecture overview
5. Case study
6. Tools
7. Conclusion

References

- My presentation from the last joint seminar of DSS and MFF in April
  - Quite many things done since then

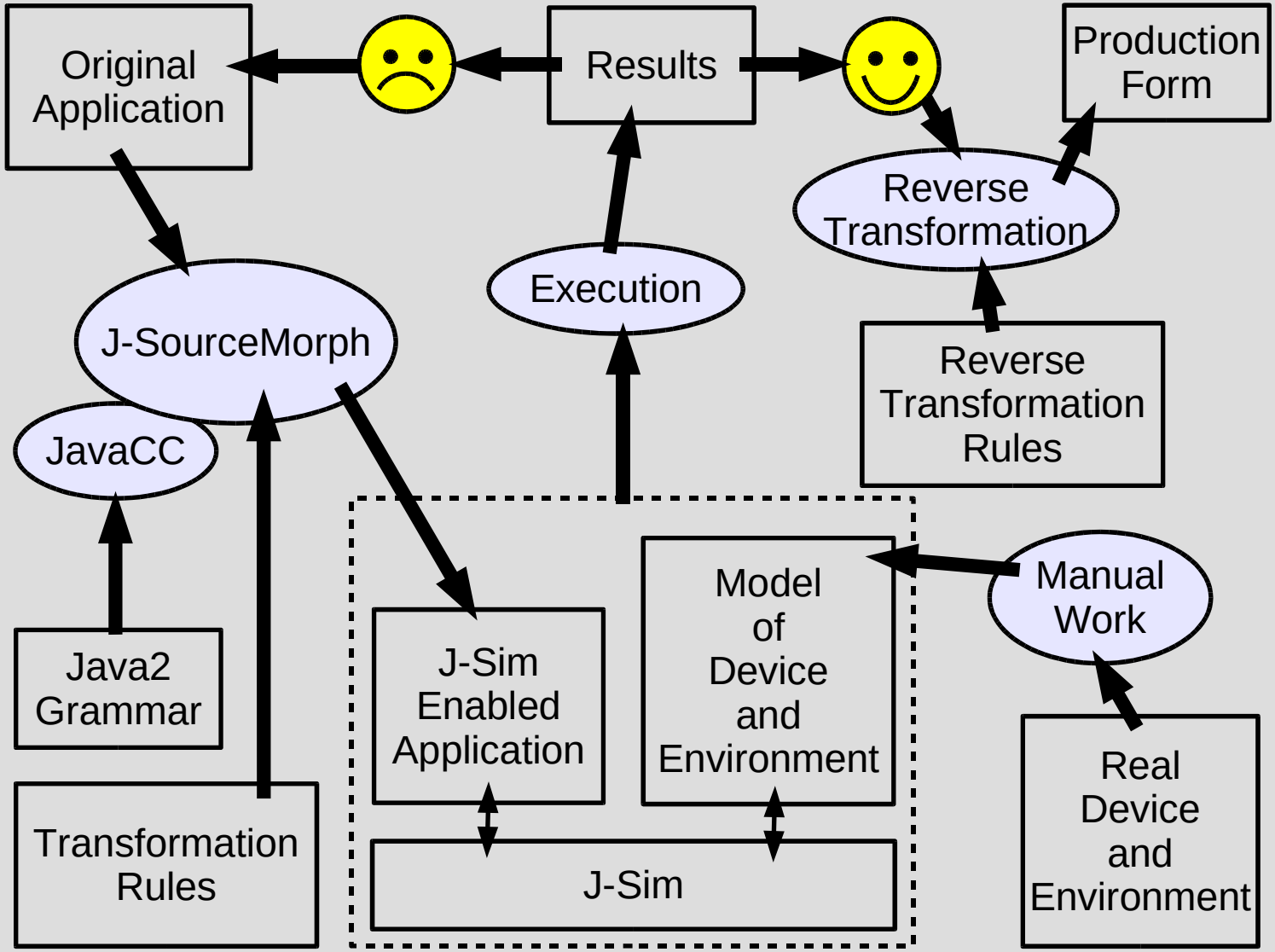
Project Goals

- Verification of Java concurrent programs for embedded devices – usually control apps
- Verification by simulation – no formal proof of correctness
- Don't test the program alone, test it in the environment it is supposed to work
- Run tests on PC, not the target platform – it may not even be available yet

Main Principles

- The program is tested by its own execution
- A modified version of the program is tested, not the original
- A model of the environment must be created, executed together with the program
- Easy transition from the original to the testing version and then (after verification) back to the production form

Development Process



Architecture Overview (1/5)

- Main software parts:
  - Control program – 2 versions: real and simulation version
  - Model of the environment – simulation version only, written by hand
  - Communication interface – 2 versions = 2 Java classes implementing the same Java interface

Architecture Overview (2/5) Communication Interface

- A set of method declarations and implementations
- Should cover all interaction of the control program with the environment = mainly I/O
- 1 Java interface with 2 different implementations (Java classes):
  - Production version comm. int. uses JNI (C or assembly inside) to communicate with real hardware
  - Simulation version comm. int. reads/writes from/to data of the environment model

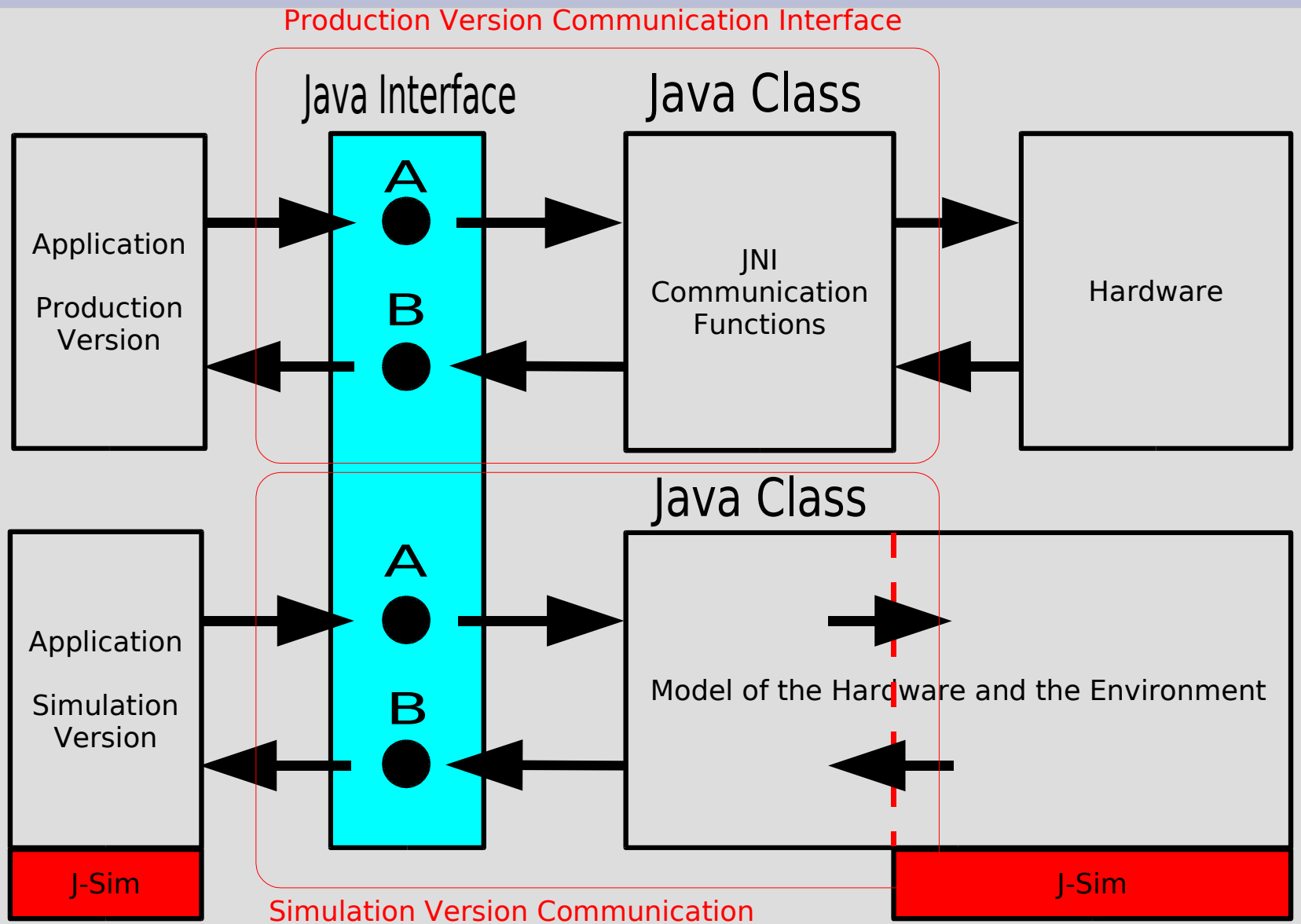
Architecture Overview (3/5) Model of the Environment

- Data + periodic processes + model implementation of the communication interface
- Can represent HW devices, nature, people's behavior, ... = anything outside the control program
- Running on top of J-Sim: Processes are instances of JSimProcess
  - Classic discrete-time process-oriented simulation
  - Simula-like scheduling

Architecture Overview (4/5) Control Program

- A multithreaded Java program, threads are periodic
- Uses monitors → synchronization, wait(), notify(), etc.
- 2 versions
  - Production version – runs “as is”
  - Simulation version – runs on top of J-Sim JiJ simulator, thread execution is managed; all threading-specific actions are simulated, modified scheduling

Architecture Overview (5/5)



Model of Computation (1/6)

- Threads enter so-called consistent states
- A consistent state = a point where both the thread's local data and monitor data are “stable”
  - Beginning/end of a synchronization block
  - wait()
  - Some other points where threads become non-runnable
- All threads are in consistent state → The program is in a consistent state

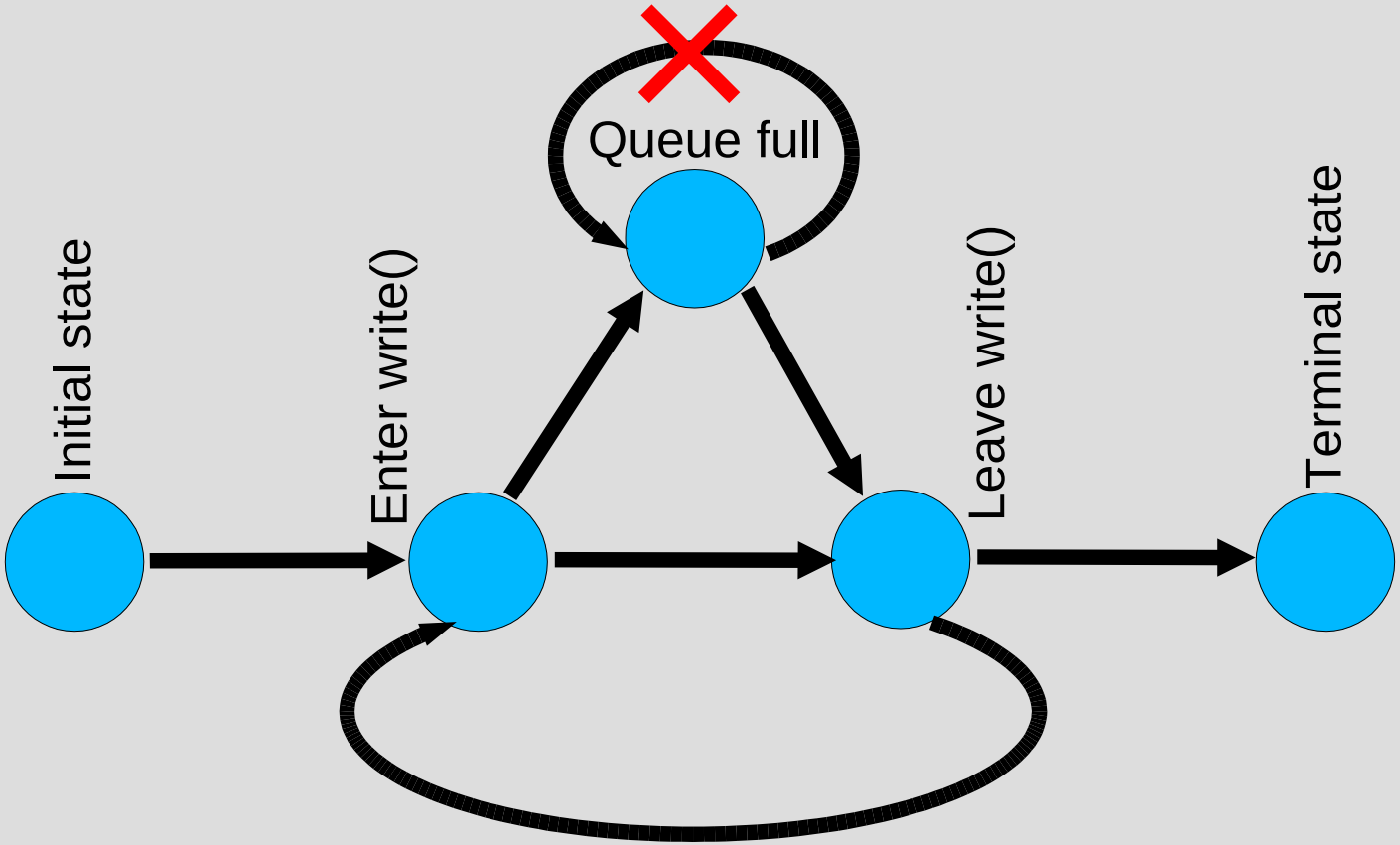
Model of Computation (2/6)

- Main principles of the simulation:
  - Just one thread running
  - Switching in program consistent states only
  - Switching can be unfluenced/controlled by the user
  - In a consistent state, a “snapshot” of the program can be taken and analyzed
- Simulation execution: The program goes through a sequence of consistent states
- In every CS, the next thread to run must be selected – Different strategies possible

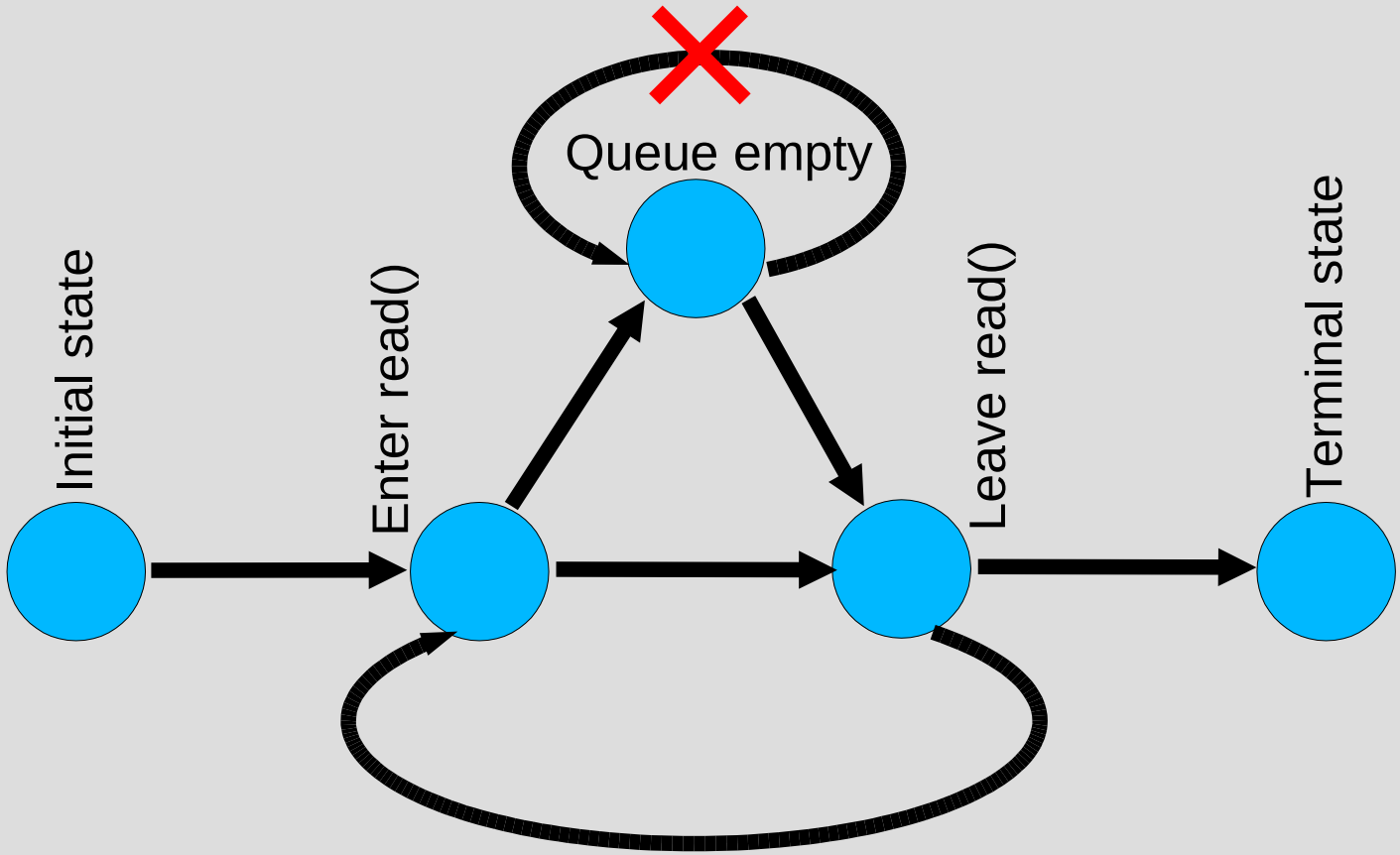
Model of Computation (3/6)
An Example

- The well-known producer-consumer problem
- 2 threads, a shared resource with synchronized access: read(), write()
- Threads may block inside monitor methods
- A thread may finish if a special value is generated/read
- Every thread has 5 consistent states, the program has 5×5 consistent states
- 2-D state space, one dimension per thread

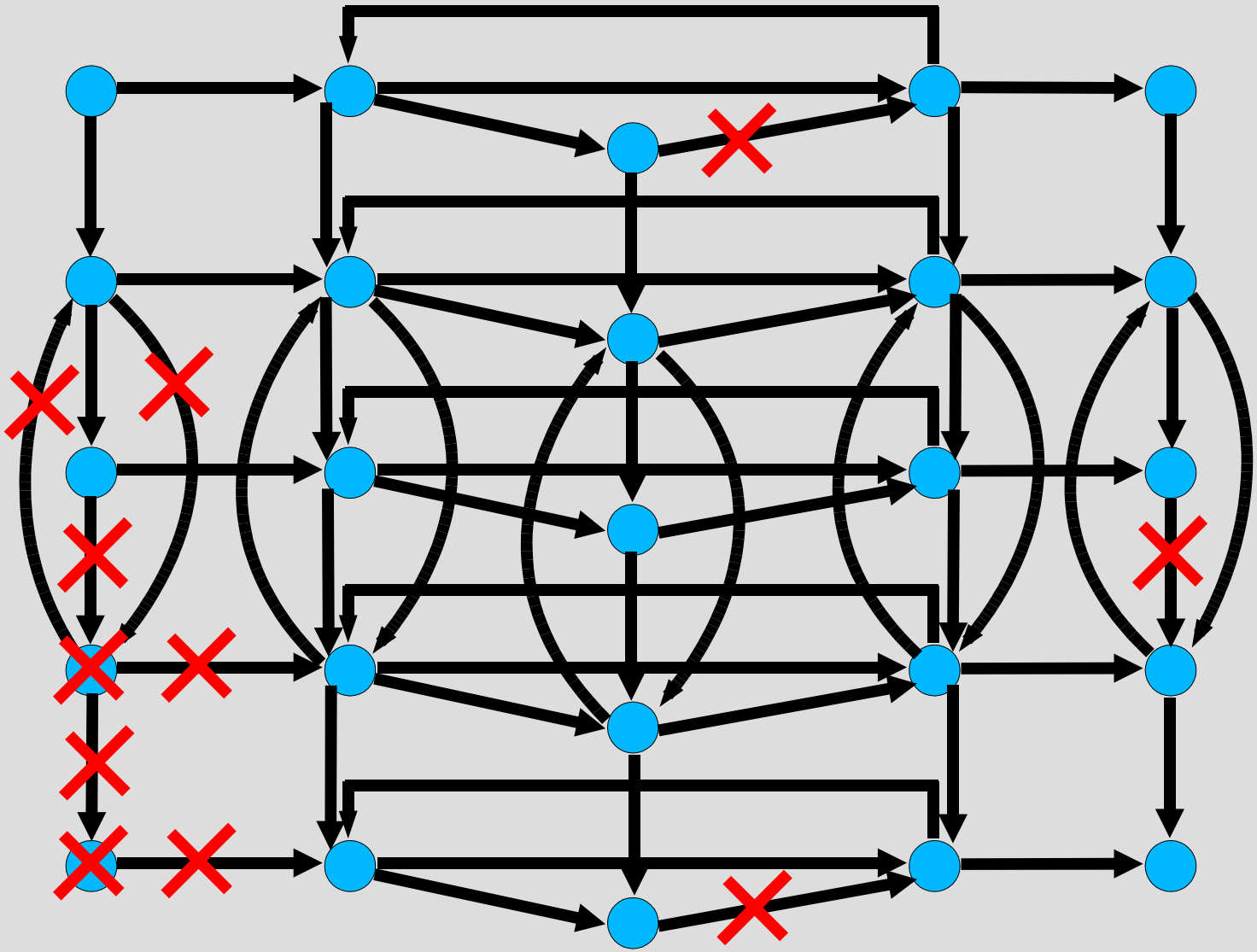
Model of Computation (4/6)
Producer State Space



Model of Computation (5/6)
Consumer State Space



Model of Computation (6/6)
Program State Space



Verification Procedure

- Application dependent
- 2 basic approaches:
  - Invariant checking in every consistent state
  - Behavioral protocols:
    - A sequence of operations on the control interface
    - Will be done at MFF (???)
- The more different computation paths explored, the bigger the chance of a discovered bug

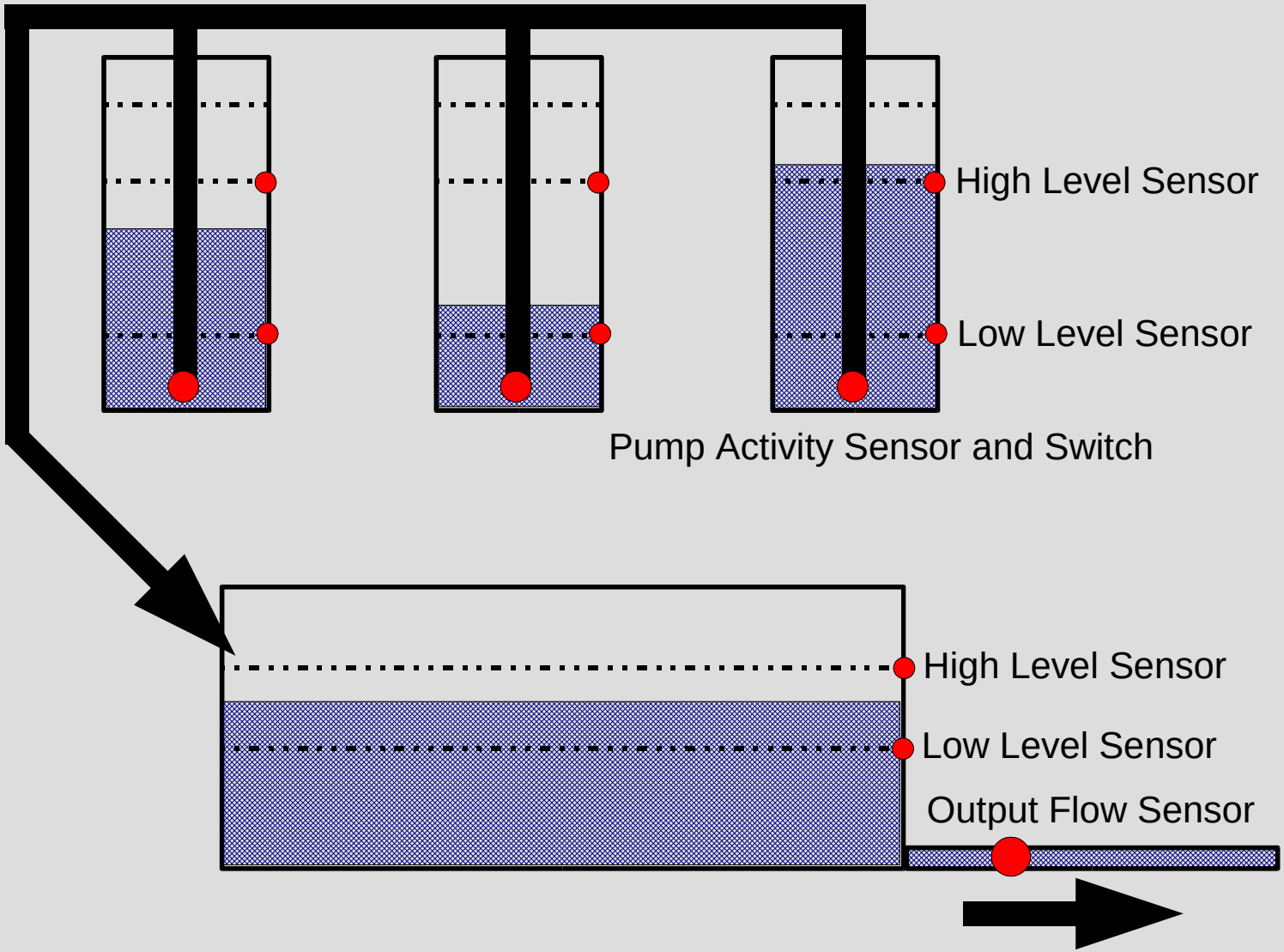
Case Study (1/8)

- A program controlling N water pumps
- N water sources with 3 sensors, 1 pump
  - Low-level sensor
  - High-level sensor
  - Pump activity sensor
  - Pump switch
- 1 main tank with 3 sensors, no pump
  - Low-level sensor
  - High-level sensor
  - Output flow sensor

Case Study (2/8)

- You control:
  - The state of every pump: ON/OFF
- You don't control:
  - Water consumption – Depends on people
  - Water coming to the sources – Natural process
- Your task: Keep the level in all water sources and in the main tank between the low-level and high-level limits

Case Study (3/8)



Case Study (4/8)
Control Interface

- readSourceLevelLow(int)
- readSourceLevelHigh(int)
- readPump(int)
- setPump(int,boolean)
- readWaterStationLevelLow()
- readWaterStationLevelHigh()

Case Study (5/8)
Model of the Environment

- Implementation of the control interface
- Environment:
  - Water level of every source
  - A process updating water level in every source
  - Water level of the main tank
  - Water consumption by people – a process
  - A process updating water level in the main tank
- Hardware:
  - Sensor data

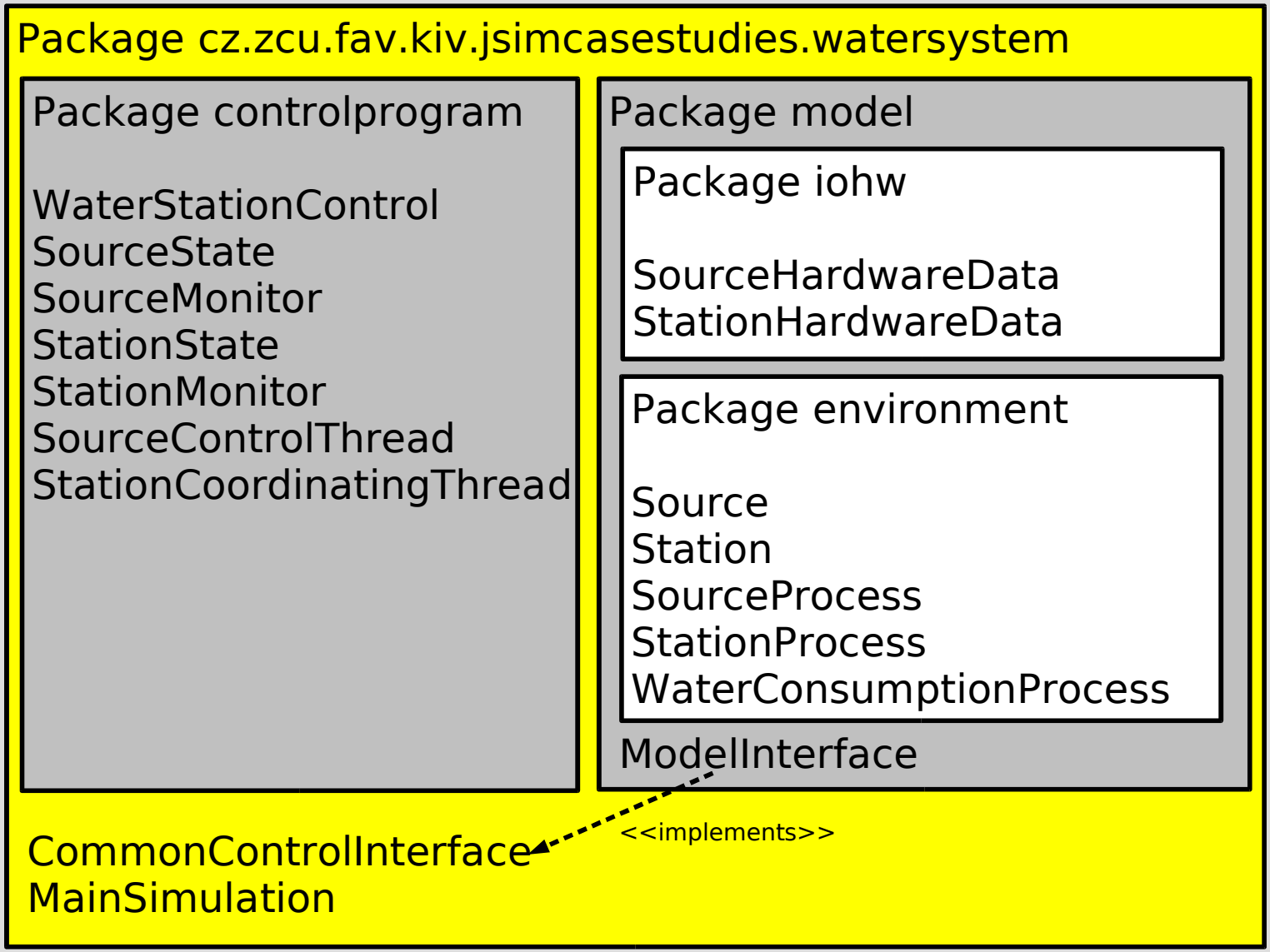
Case Study (6/8)
Control Program

- N infinite control threads, 1 per source
- Monitors reporting state of all 3 sensors at once
- A monitor of the main tank
  - At most K (K<N) pumps can be switched on
  - The level must not get above the upper limit
  - Control threads may get blocked here when trying to switch the pump on
- Switching on/off: A built-in hysteresis, the level must get to the upper/lower limit before the pump is switched on/of

Case Study (7/8) Invariants

- The number of running pumps must not exceed K
- The main tank's level is at the high-level limit → No pump can be running
- The I^{th} source level is below the low-level limit → The I^{th} pump must be switched off
- The tank's level is below the low limit and at least K sources are above the high limit → K pumps should be switched on

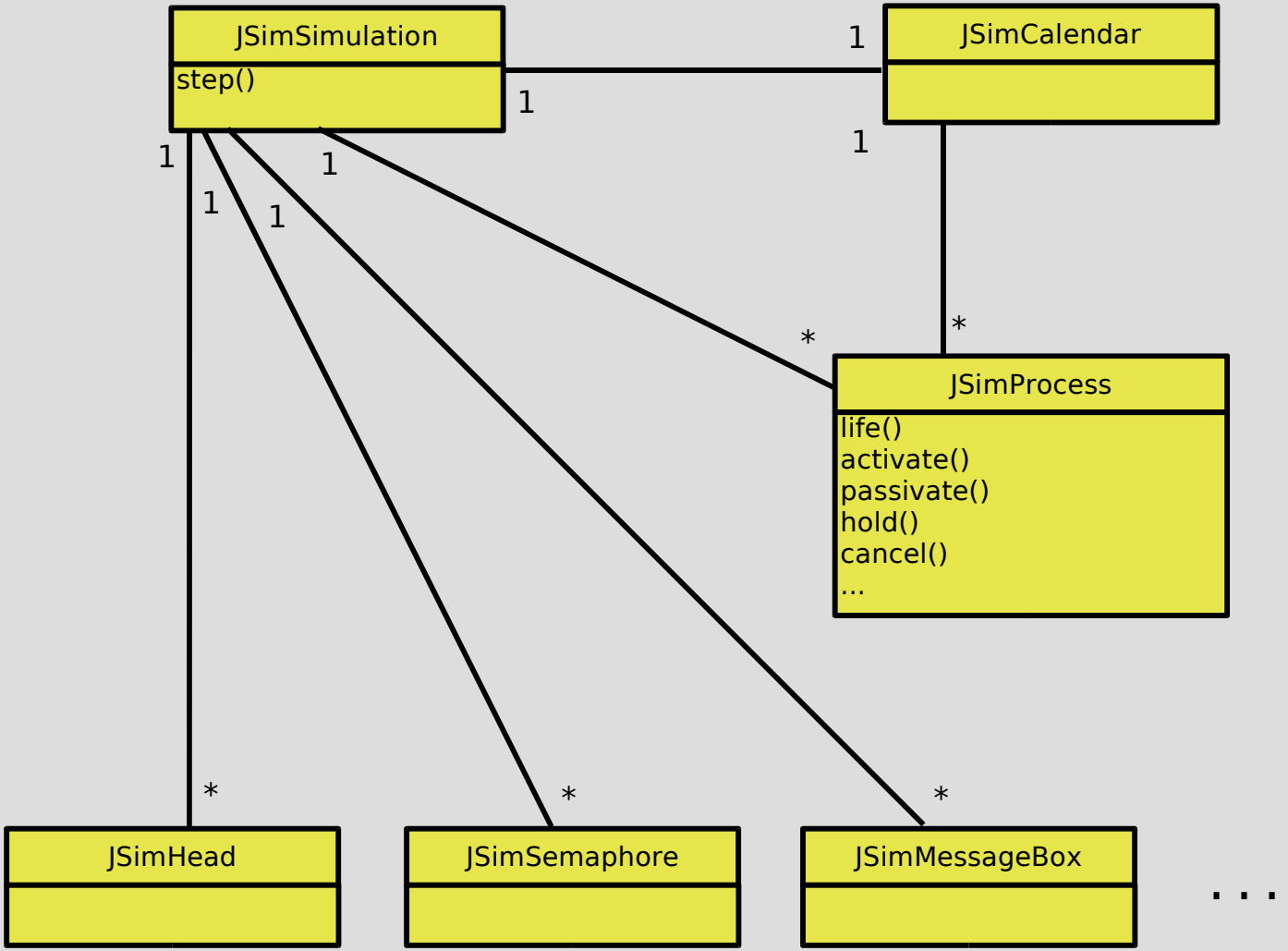
Case Study (8/8) – Structure of Source Code



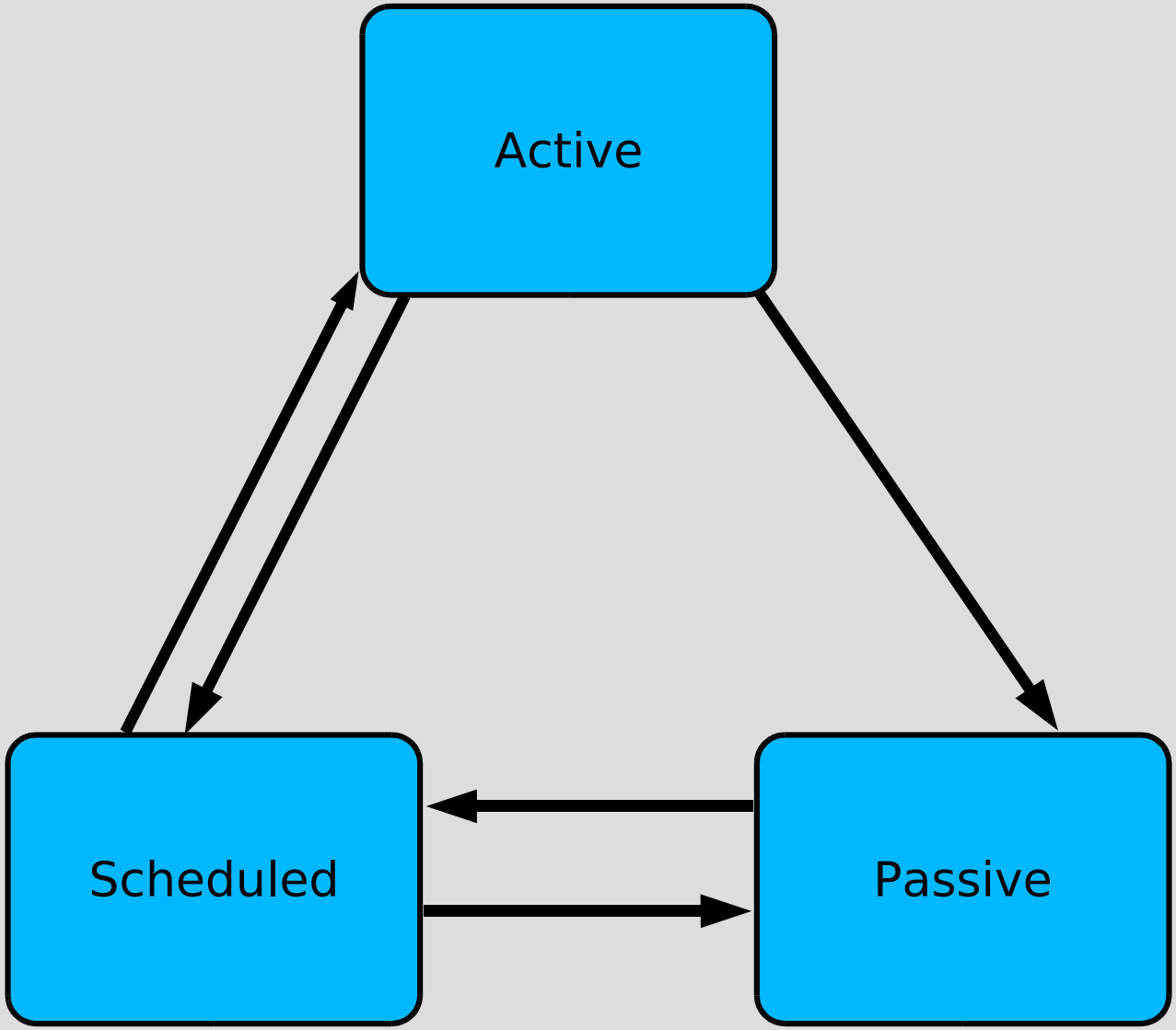
Tools (1/8)
J-Sim

- Discrete-time process-oriented simulation library
- “Java in Java” package for simulation of Java multithreading
  - Synchronization (locking)
  - wait(), notify(), notifyAll()
  - java.lang.Thread methods: sleep(), join(), ...
- Execution of a Java program simulation can be interleaved with classic J-Sim processes
  - Java threads share time of the processor
  - Classic processes run “in parallel” in zero time

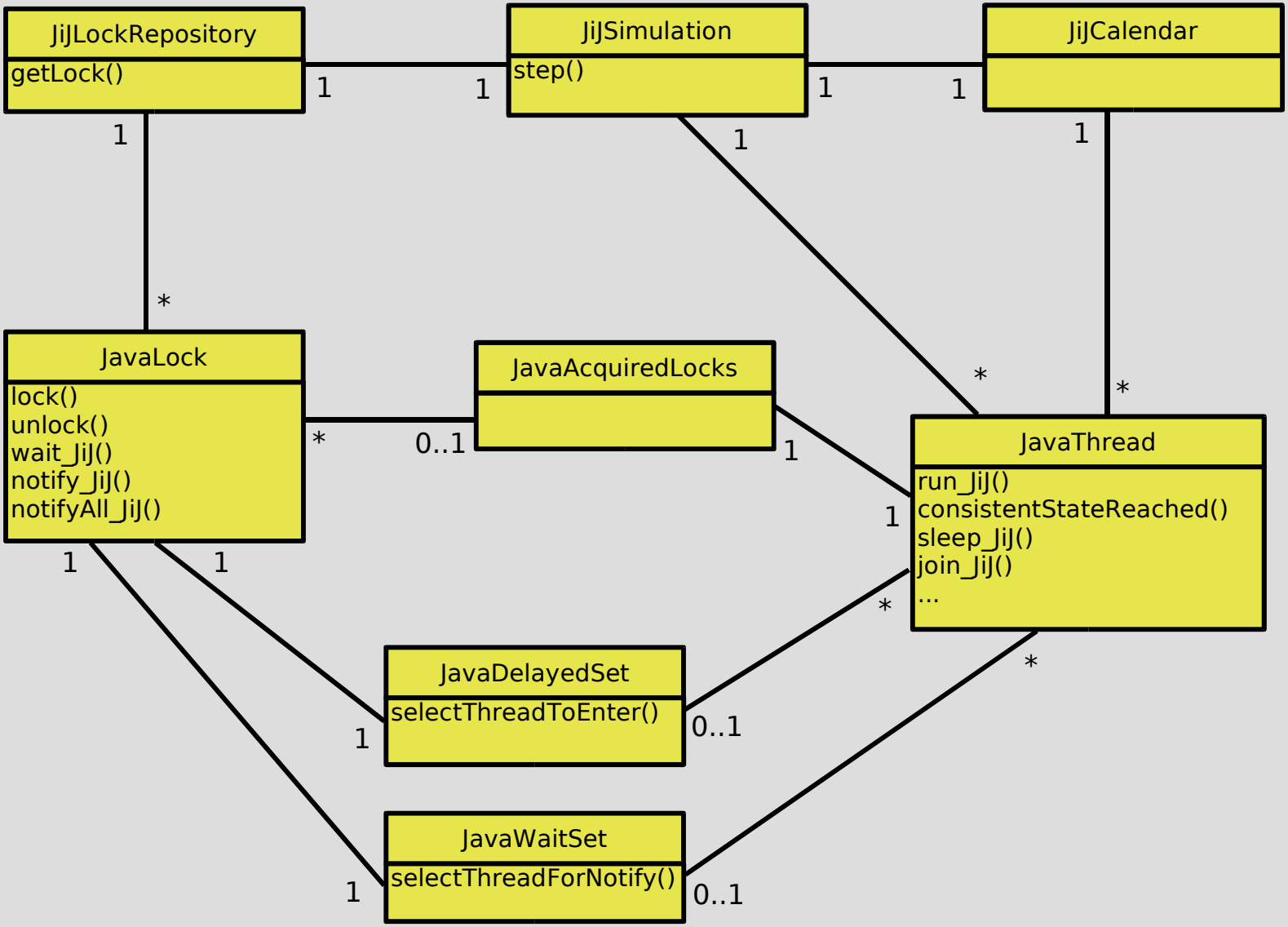
Tools (2/8)
J-Sim Classes



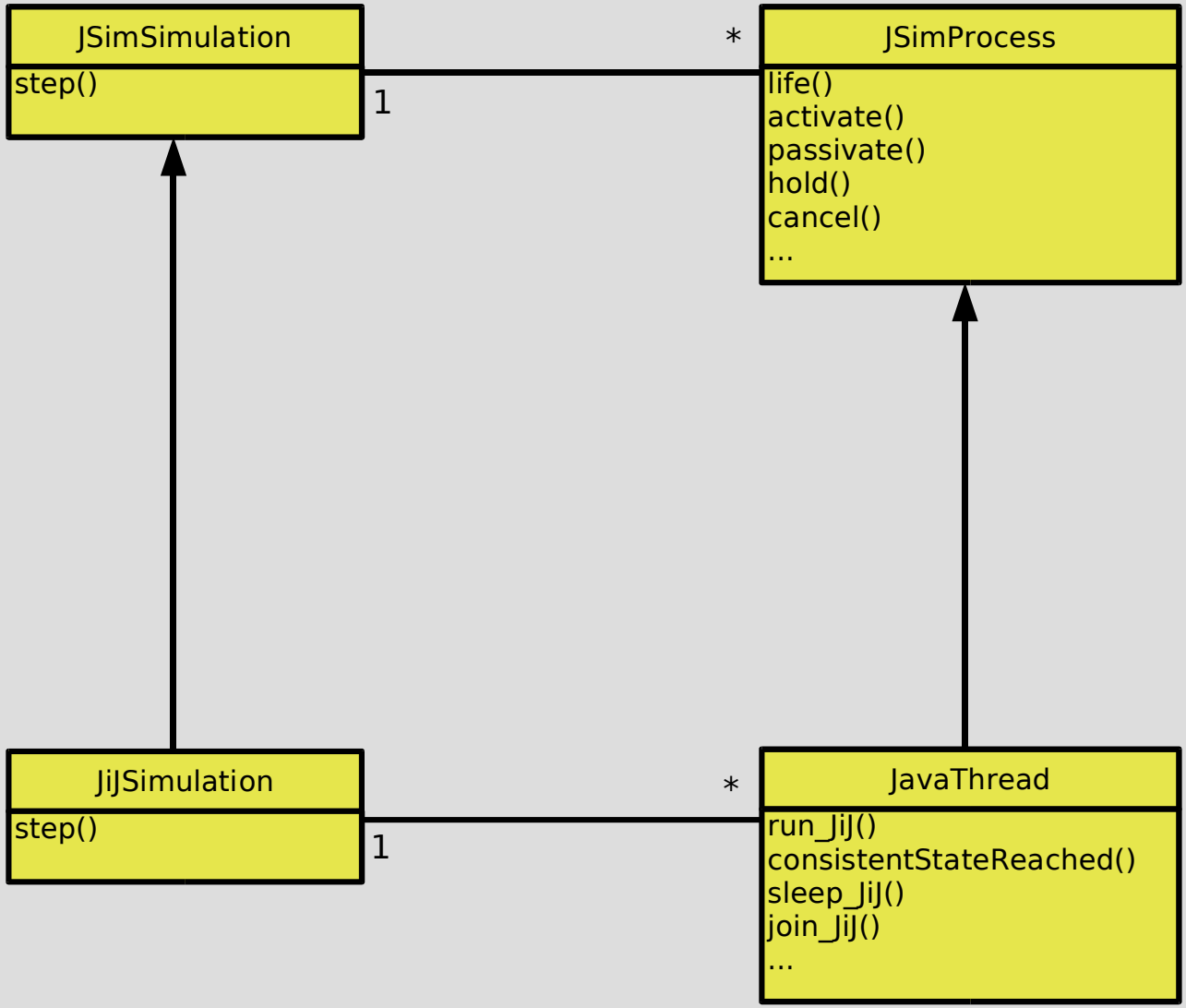
Tools (3/8)
J-Sim Process States



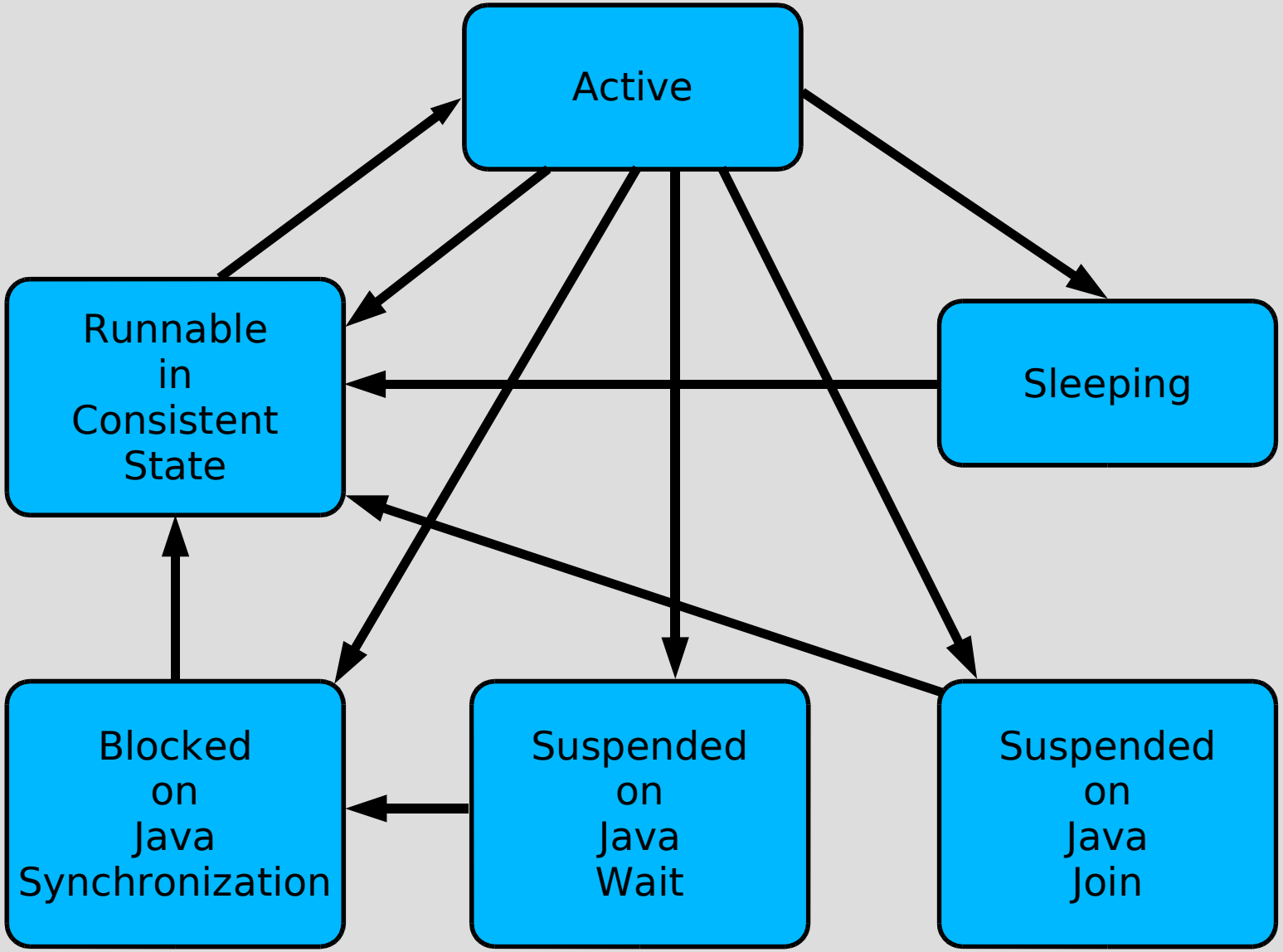
Tools (4/8)
JiJ Classes



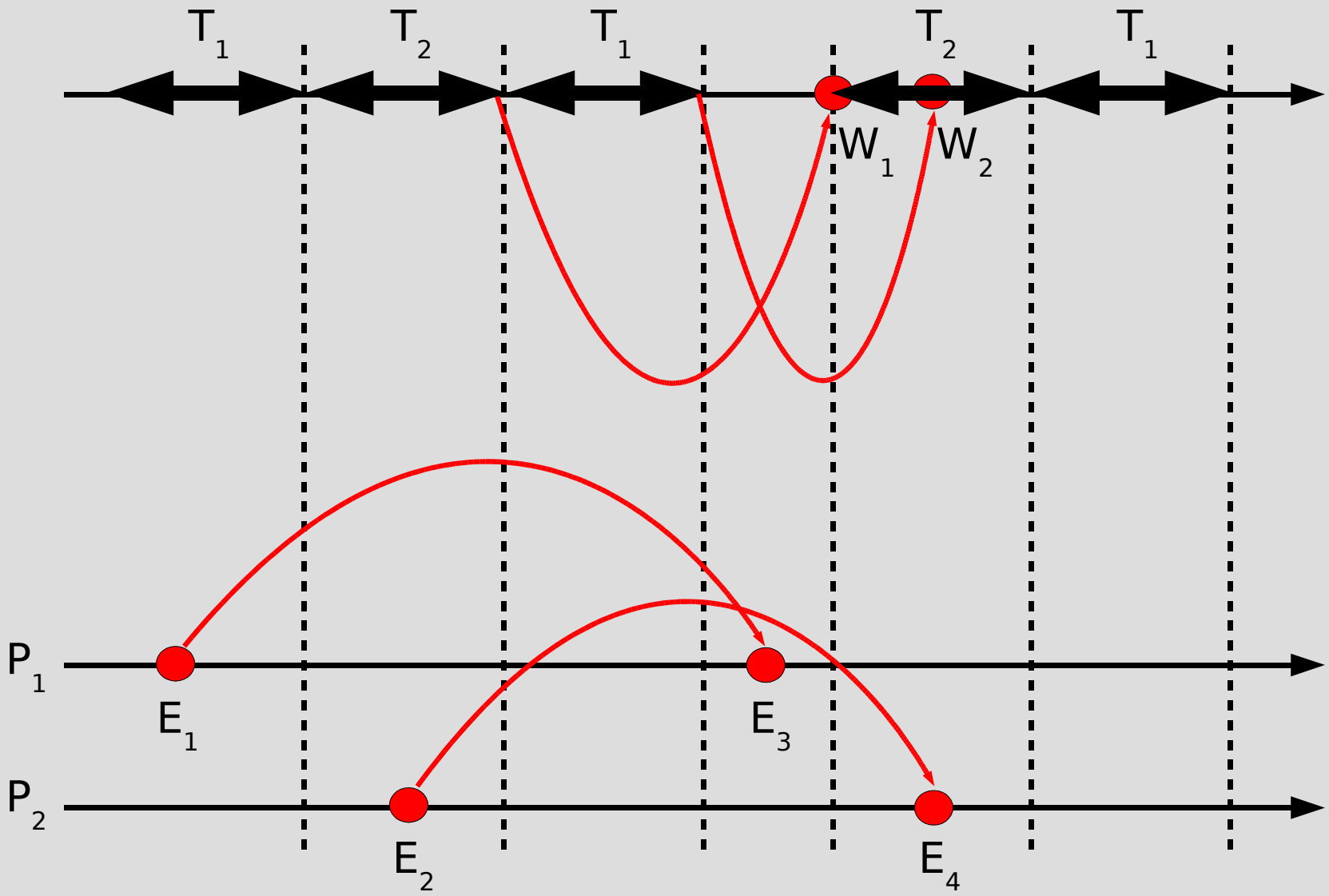
Tools (5/8)
Relation between J-Sim and JiJ



Tools (6/8)
JiJ JavaThread States



Tools (7/8)
Joint Scheduling



Tools (8/8)
J-SourceMorph

- A conversion utility
- Input: Java sources; Output: Java sources
- Based on JavaCC – a parser generator
- Parametrized by an XML file with transformation rules (not ready for this task)
- Can change/add/remove:
  - Imports
  - Attributes
  - Methods, method headers
  - Method calls
  - Classes, superclass, implemented interfaces

Availability & References

- J-Sim v. 0.4.0
  - The JiJ package
  - Case study using JiJ
  - http://www.j-sim.zcu.cz
- J-SourceMorph
  - http://www.j-sourcemorph.zcu.cz
  - XML transformation rules not ready yet
- See Technical Report 2004-03
  - http://www.kiv.zcu.cz/publications/techreports.php

Conclusion & Future Work

- Future plans:
  - All java.lang.Thread functionality should be simulated
  - Transformation rules for J-SourceMorph
  - Support for RT-Java (JSR-001)
  - RMA – Rate Monotonic Analysis
    - Do all threads meet their deadlines?
- Results:
  - “Mixed” simulation works! Invariants are sometimes broken for a short time – expected behavior.