Testing Java Concurrent Software for Embedded Devices

Jaroslav Kacer
jkacer@kiv.zcu.cz

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

2004-05-17

Outline

1. Introduction, Project goals
2. Related work
3. Architecture overview
4. J-Serializer
5. J-Sim
6. Case study
7. New requirements from industry (Apogee)
8. Conclusion

Introduction

- Bugs in concurrent programs are hard to be found
- Many different behaviors, some of them incorrect
- How to crash/freeze the program again in the same way?
- Java: Concurrency-related things are easy to program but the problems remain
- Embedded devices: Will my program work OK with all that connected hardware?

Project Goals

- Develop an experimental method of concurrent Java programs testing and all tools needed for it
- The environment (HW, network, ...) should also be included
- When a problem found (also throughout all testing), the program's analysis should be available

Related Work

- Theoretical methods (Petri nets, pi-calculus) unusable
- Program checkers like SPIN, LTSA usually test just a model of the program
- Some checkers (VeriSoft) test UNIX programs in C
- Java: ExitBlock, Java PathFinder 2, Bandera
- Some assumptions and special requirements
- The FIT project: Testing the TTP/C protocol

Architecture Overview (1/3)

- Two main parts executed together during testing:
  - The tested program (in a modified form)
  - Model of the environment
- The program converted automatically
- The model created manually
- Two possible approaches of binding them together

Architecture Overview (2/3)

- Key idea: Threads are run in a serialized way, just one at any moment
- Well defined points of switching, in so-called consistent states
- Principles of Java scheduling must be preserved
- But there is still much space for you to drive the program
- Application threads interleave with model's processes

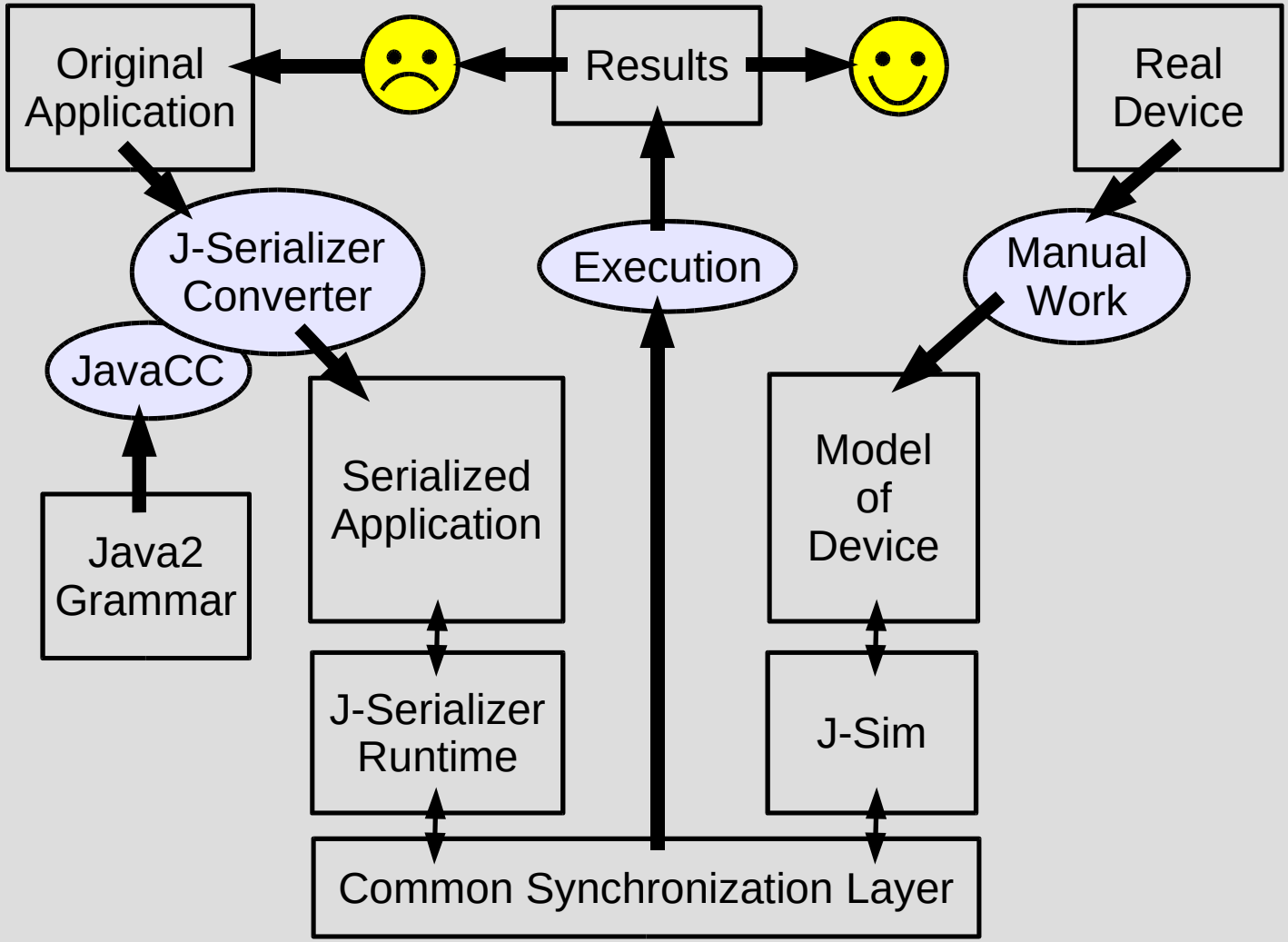
Architecture Overview (3/3)

- A communication interface exists, used by the application
  - Implemented as a Java interface
- During testing, this interface points to a class of the model
- During real operation, this interface points to a class performing HW operations
- Can be extended for network communication etc.

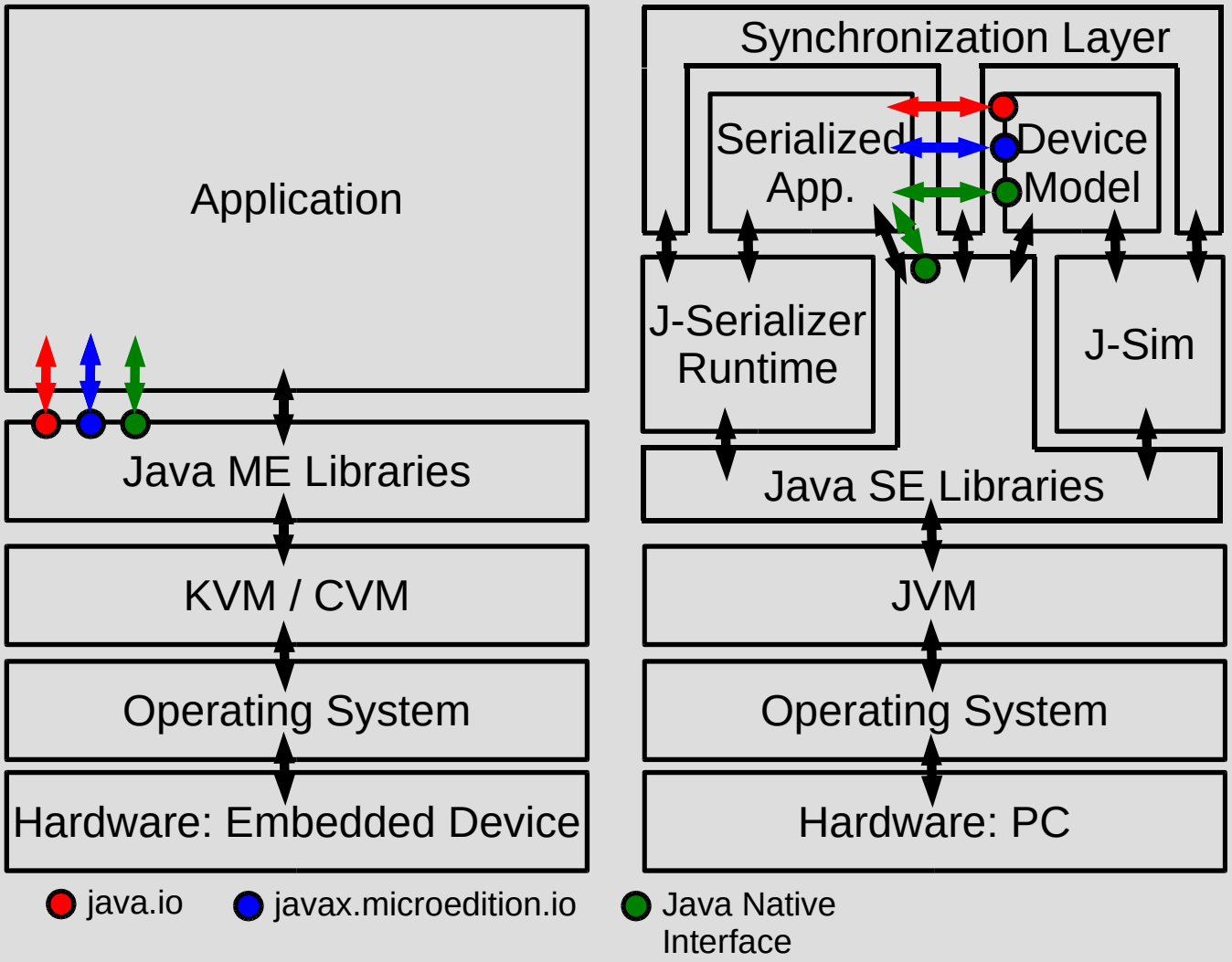
Approach 1 (1/3)

- Model on top of J-Sim
- Program on top of J-Serializer Runtime
- Synchronization necessary
  - Shared calendar
  - Either a thread of the program or a process of the model is running
- J-Serializer Runtime assures proper scheduling of the application part when its turn comes
- Model processes scheduled as usual

Approach 1 (2/3)



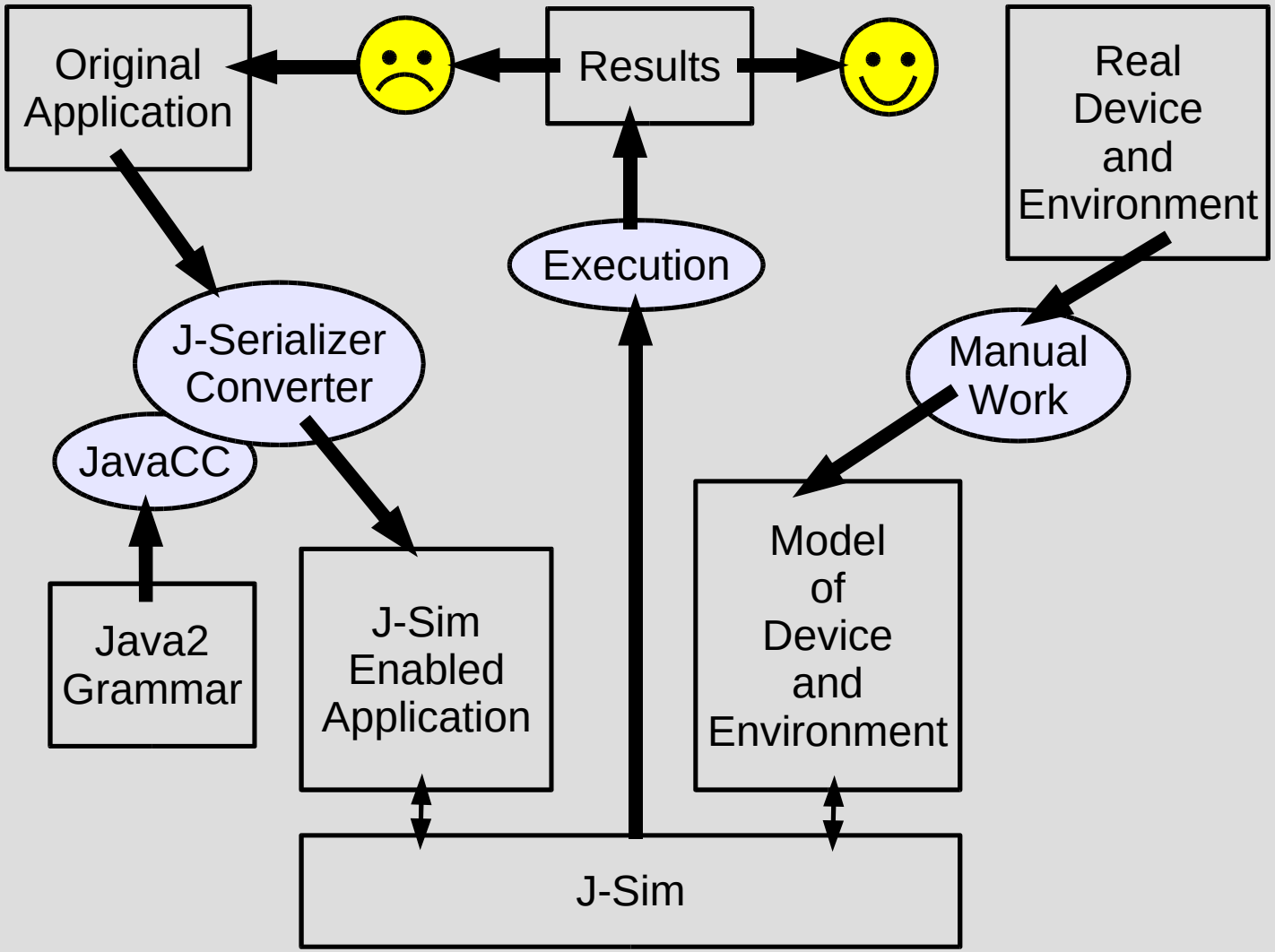
Approach 1 (3/3)



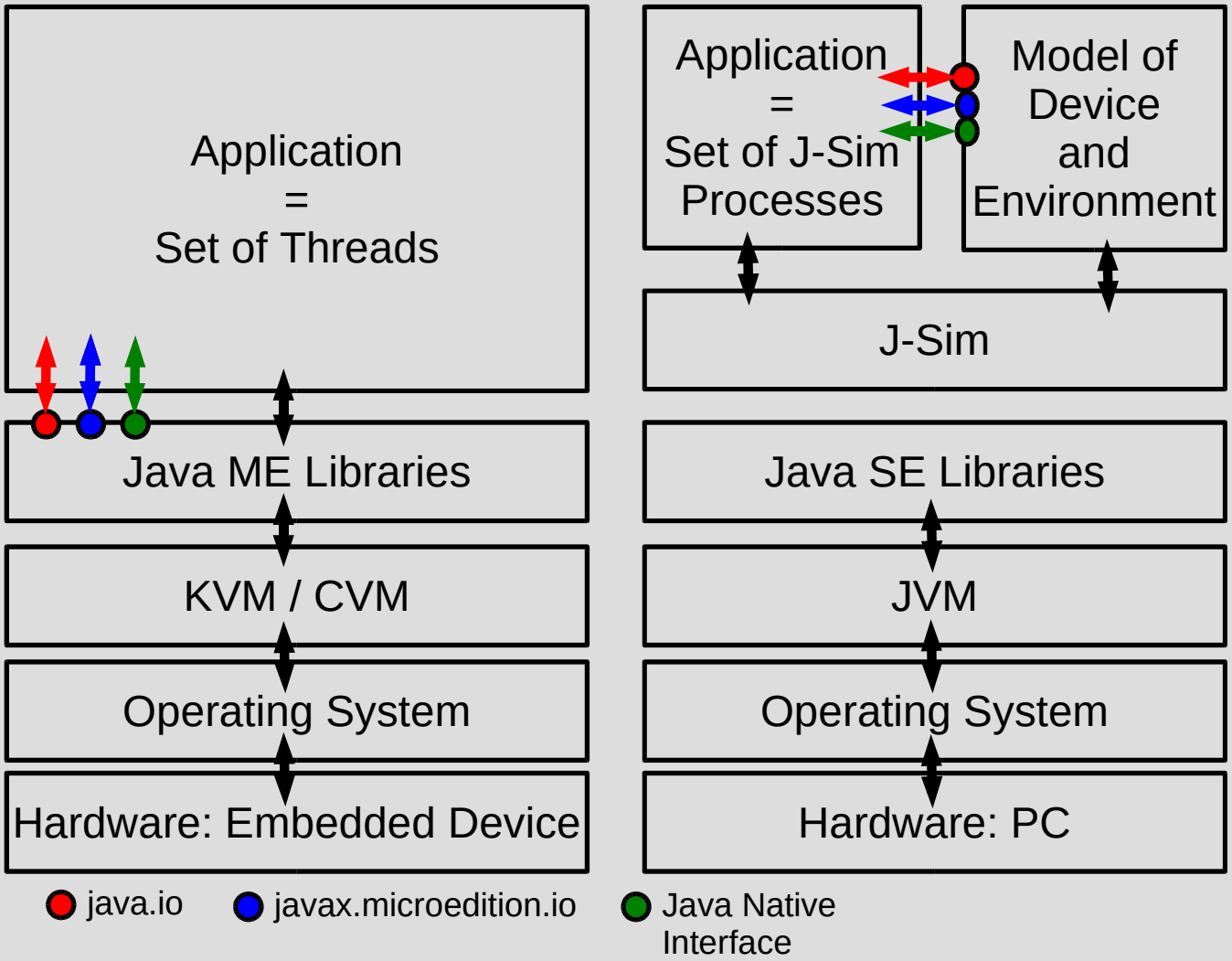
Approach 2 (1/3)

- Both model and program on top of J-Sim
- No synchronization necessary, just one simulation object
- Requires J-Sim to support simulation of a part of the JVM
  - A new package to J-Sim: jsim.jij (Java in Java)
  - Previously a part of J-Serializer Runtime
- Requires different set of rules for J-Serializer Converter
- Problems with simulation time advancement

Approach 2 (2/3)



Approach 2 (3/3)



Model of the Environment

- A set of J-Sim processes and data
- Data accessed via a public interface
- Periodic processes with discrete-time events
- Two subparts:
  - Hardware
  - Natural processes
- Model data read/set by:
  - The model itself
  - The tested application

Model of the Control Program

- A set of threads (or J-Sim processes) and data
- Accesses its own data and model data
  - Model data accessed via the common interface, the application is not aware of the model
- Key issue: Principles of scheduling
  - There can be several possible ways how to schedule serialized threads
  - They may / may not have a relation to real time

Scheduling the Threads (1/2)

- Inside the application, JVM rules must be respected
- How to choose from a set of runnable threads with the same priority?
  - Random choice
  - Round robin
  - Real time measurement (possibly with randomization or other changes)
  - User-driven selection
  - Prescribed path (hard to determine first) or sub-graph of the whole state space
  - Focus on selected threads

Scheduling the Threads (2/2)

- In all cases, the simulation time must be advanced somehow (because of the model)
- If low load is guaranteed and sleep() is used, real actions can be considered infinitely short and only sleep will influence the simulation time
- In other cases, the problem is more complicated
- Time conditions don't necessarily have direct impact on selecting the next thread (app. 1)

J-Serializer

- A tool for time serialization of threads
- Two parts
  - Converter: Converts source files into a modified form, using the Runtime
  - Runtime: Responsible for thread scheduling and switching, simulates a part of JVM functionality (Java in Java simulator): threads, locks, thread states, priorities, waiting on locks, ...
- Not freely available yet

J-Serializer Converter

- Currently subject of a MSc. thesis
- A Java application
- Able to delete/replace/add imports, class signatures, method signatures, method calls, class attributes, ...
- Rules in a XML file
- Macro expansions (e.g. for wait())
- Based on JavaCC – A Java parser generator

J-Sim

- A Java library for discrete-time process-oriented simulation
- Principles known from Simula
  - Simulation time
  - Processes and their interleaving
  - activate(), passivate(), hold(), cancel()
  - step()
  - Heads and links
- New packages in 0.3.0: random, ipc, ...
- http://www.j-sim.zcu.cz

New Features in J-Sim 0.3.0

- Building with Ant
- Simulation of inter-process communication
  - Semaphores
  - Message passing
- Independent generators of random numbers (streams), initial seed allows for repeatibility
- Package `Unsecure' for experiments with distributed simulation
- Case studies

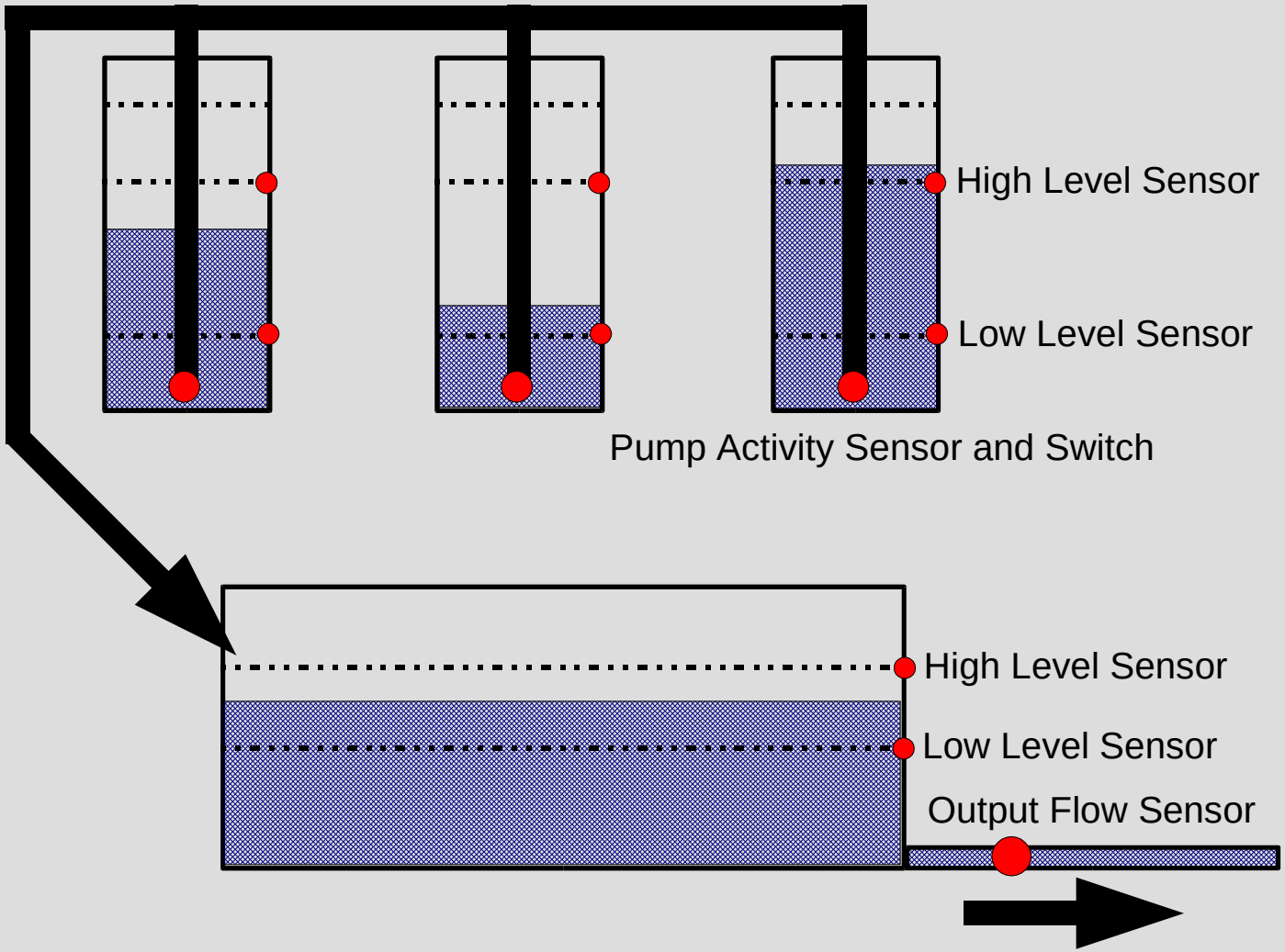
Still Missing in J-Sim

- The JiJ package
- Complete, up-to-date documentation, easily manageable (DocBook)
- Case study #1 not finished
  - Depends on JiJ
  - Network interface
  - Must be polished
  - Tech report 2004-03

Case Study (1/6)

- A system of water sources and a main water tank
- Sources: sensors for high/low level (binary), a pump with a sensor (binary) and a switch
- Main water tank: sensors for high/low level, output flow register
- The aim is to keep all levels between low and high limit, if possible

Case Study (2/6)



Case Study (3/6): The Common Control Interface

- Reading the sensors of the i-th source
- Switching the pump of i-th source on/off
- Reading the sensors of the main tank
- Reading the output flow register value

Case Study (4/6): The Control Program

- Communicates via the common interface
- Does not know what is behind the interface, either the model or real hardware or network or ...
- One control thread per source, periodically checking sensors and deciding about pump activity
- Can be blocked in the main tank's monitor when switching on: a limit of running pumps + the level in the tank may be too high

Case Study (5/6): The Model

- Classes for the sources and the tank and their HW data (sensor data)
- Periodic processes
  - Sources' water level adjustments – natural process
  - Output flow changes – people's activity
  - The main tank's level adjustments – the model must reflect pump activity
  - Sensors must be set appropriately

Case Study (6/6): Current State

- We use a reversed process to the described one: writing the “converted” application first, than rewriting it to its “original” form
- This is necessary for development of tools and for proper analysis of the method
- Included as Case Study #1 in J-Sim 0.3.0

New Requirements from Industry (1/2)

- Apogee: JVM ports for about 100 different platforms (platform = processor × OS)
- Every program must be tested on its respective platform to reach the C or B level (military customers), plus extensive documentation
- This would be just a very small part of the complete testing (and development) process
- Integration with Aphelion

New Requirements from Industry (2/2)

- Requirement:
  - Don't test only the application and the outer world
  - Test it together with lower layers: CDC, CDLC, user profiles, even OS (an RTOS)
  - Parametrization: available memory, its allocation, garbage collection, ...
  - JSR-000001: Real-Time Java – George Malek also worked on it
- So far no idea how to do it
- Any suggestions?

Conclusion

- This is an experimental method that can find incorrect behavior but cannot guarantee correct one
- The method combines thread serialization with environment modeling (so far unique for Java, afaik)
- Still worked on (J-Sim, J-Serializer, case studies, a tech report, the Apogee proposal)