

JAVA FOR SAFETY CRITICAL EMBEDDED HARD-REAL-TIME SYSTEMS

Bent Thomsen

Aalborg University

A typical safety critical embedded hard-real-time program

Cruise control:

Loop every X microseconds

- Read the sensors;

- Compute speed;

- if speed too high

 - Compute pressure for brake pedal;

- if speed too low

 - Compute pressure for accelerator;

- Transmit the outputs to actuators;

- wait for next period;

How hard can it be to program such systems?

Apparently hard enough

- Toyota's Accelerator Problem Probably Caused by Embedded Software Bugs
- Software Bug Causes Toyota Recall of Almost Half a Million New Hybrid Cars
- BMW recall: The company will replace defective high-pressure fuel pump and update software in 150,000 vehicles.

Some examples

- The Ariane 5 satellite launcher malfunction
 - caused by a faulty software exception routine resulting from a bad 64-bit floating point to 16-bit integer conversion
- LA Air Traffic control system shutdown (2004)
 - Caused by count down timer reaching zero
- Airbus A330 nose-diving twice while at cruising altitude (2001)
 - 39 injured, 12 seriously. Problem never found

A hard real-time problem

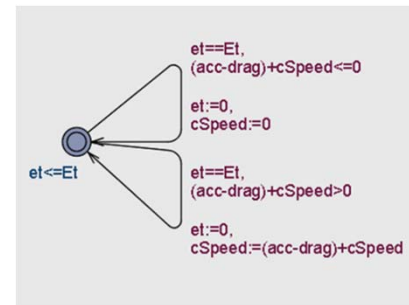
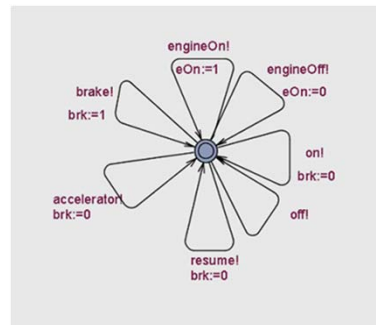


Embedded Systems

- Over 90% of all microprocessors are used for real-time and embedded systems
 - Market growing 10% year on year
- Usually programmed in C or Assembler
 - Hard, error prone, work
 - But preferred choice
 - Close to hardware
 - No real alternatives Well ... ADA – 10th on the list of most wanted skills
 - Difficult to find new skilled programmers
 - Jackson Structured Development (1975) still widely used
 - EE Times calling for re-introducing C programming at US Uni

Model Driven Development

- Develop Model of System
- Verify desirable properties
- Generate Code from Model



- But ..
 - Many finds developing models harder than programming
 - Often some parts have to be programmed anyhow
 - Model and code have tendency to drift apart

We need to look for other languages

- The number of embedded systems is growing
- More functionality in each system is required
- More reliable systems are needed
- Time to market is getting shorter
- Increase productivity
 - Software engineering practices (OOA&D) – 10%
 - Tools (IDEs, analyzers and verifiers) – 10%
 - New Languages -700%
 - 200%-300% in embedded systems programming (Atego)

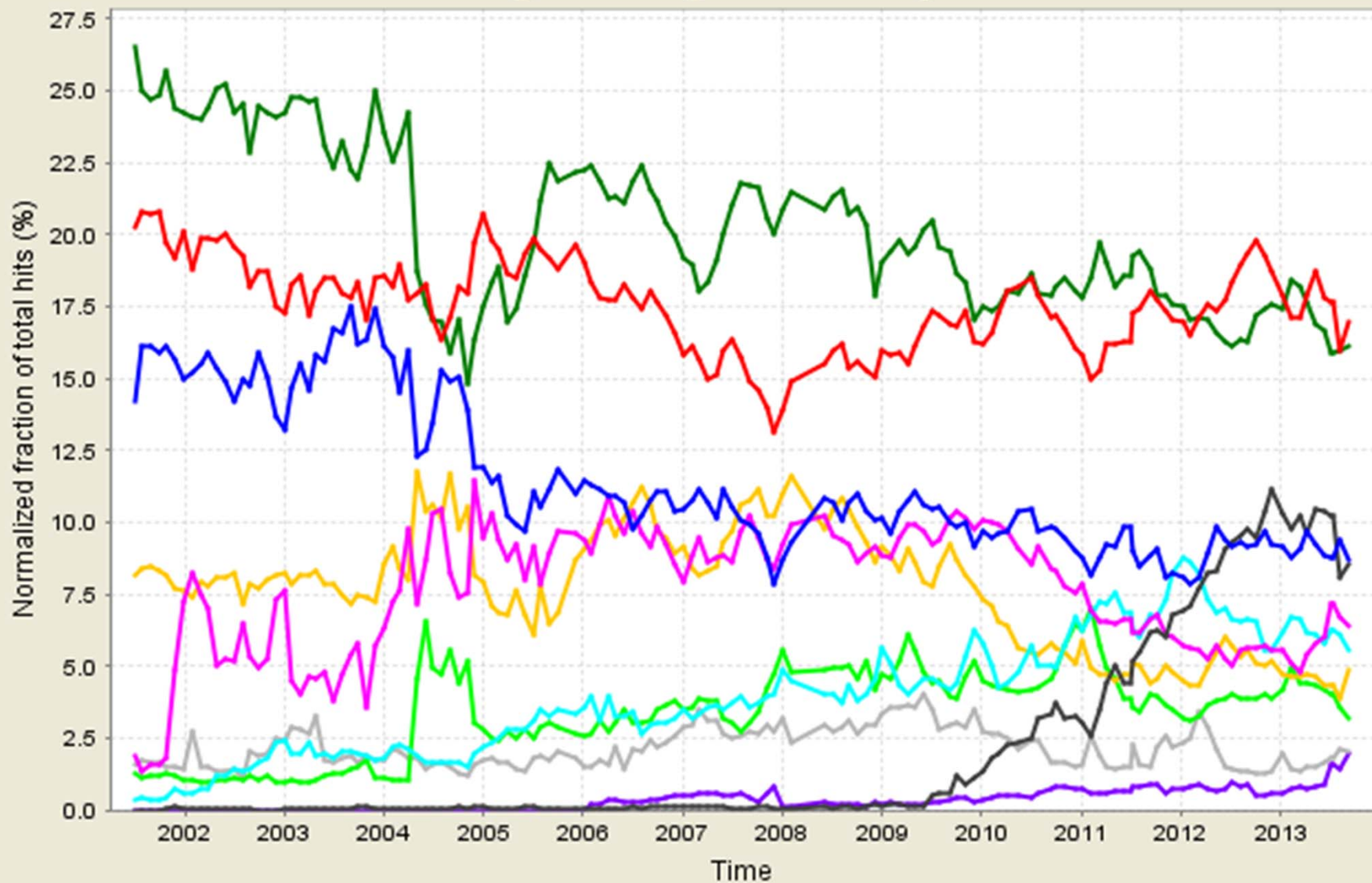
Java

- Most popular programming language ever !
 - In 2005 Sun estimated 4.5 million Java programmers
 - In 2010 Oracle estimated 9 million Java programmers
 - 61% of all programmers are Java programmers
- Originally designed for setop-boxes
- But propelled to popularity by the internet



<http://jaxenter.com/how-many-java-developers-are-there-10462.html>

TIOBE Programming Community Index



C C++ PHP (Visual) Basic JavaScript
Java Objective-C C# Python Transact-SQL

Advantage of Java over C and C++

- Clean syntax and (relative) clean semantics
- No preprocessor
- Wide range of tool support
- Single dispatch style OOP
- Strong, extendible type system
- Better support for separating subtyping and reuse via interfaces and single inheritance
- No explicit pointer manipulation
- Pointer safe deallocation
- Built-in Concurrency model
- Portability via JVM (write once, run anywhere)

Embedded hard real-time safety-critical systems

- Nuclear Power plants, car-control systems, aeroplanes etc.
- Embedded Systems
 - Limited Processor power
 - Limited memory
 - Resources matter!
- Hard real-time systems
 - Timeliness
- Safety-critical systems
 - Functional correctness
- Grundfos pumps and SKOV pig farm air conditions
- Aalborg Industries (ship boilers) and Therma (aero, defence)
- GomSpace and NASA

What is the problem with Java?

- Unpredictable performance
 - Memory
 - Garbage collected heap
 - Control and data flow
 - Dynamic class loading
 - Recursion
 - Unbounded loops
 - Dynamic dispatch
 - Scheduling
 - Lack high resolution time
- JVM
 - Good for portability – bad for predicatbility

Observation

There is essentially only one way to get a more predictable language:

- namely to select a set of features which makes it controllable.
- Which implies that a set of features can be deselected as well

Real-Time Java Profiles

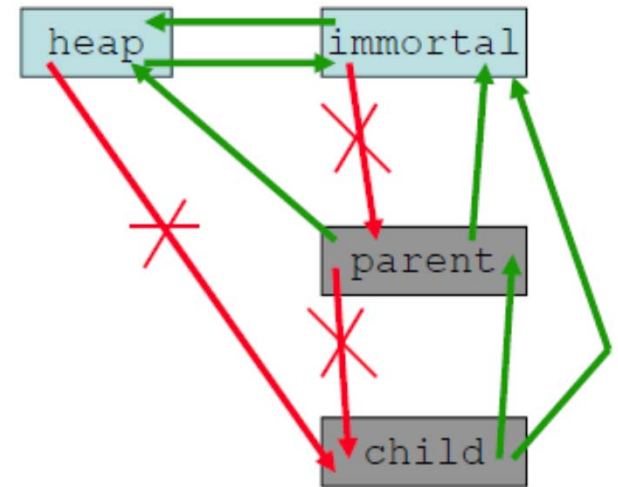
- RTSJ (JSR 001)
 - The Real-Time Specification for Java
 - An attempt to cover everything
 - too complex and dynamic
 - Not suitable for high integrity systems
- Safety-Critical Java (draft) (JSR 302)
 - Subset of RTSJ
 - Focus on simplicity, analysability, and certification
 - No garbage collection: Scoped memory
 - Missions and Handlers (and some threads)
 - Implementation: sub-classes of RTSJ
- Predictable Java
 - Super classes for RTSJ
 - Simple structure
 - Inspiration for SCJ

Real-Time Specification for Java (RTSJ)

- Java Community Standard (JSR 1, JSR 282)
 - Started in 1998
 - January 2002 – RTSJ 1.0 Accepted by JSP
 - Spring 2005 – RTSJ 1.0.1 released
 - Summer 2006 – RTSJ 1.0.2 initiated
 - March 2009 Early draft of RTSJ version 1.1 now called JSR 282.
- Most common for real-time Java applications
 - Especially on Wall Street
- New Thread model: NoHeapRealtimeThread
 - Never interrupted by Garbage Collector
 - Threads may not access Heap Objects
 - Extends Java's 10 priority levels to 28

RTSJ Overview

- Clear definition of scheduler
- Priority inheritance protocol
- NoHeapRealtimeThread
- BoundAsyncEventHandler
- Scoped memory to avoid GC
- Low-level access through raw memory
- High resolution time and timer
- Originally targeted at larger systems
 - implementation from Sun requires a dual UltraSparc III or higher with 512 MB memory and the Solaris 10 operating system



RTSJ Guiding Principles

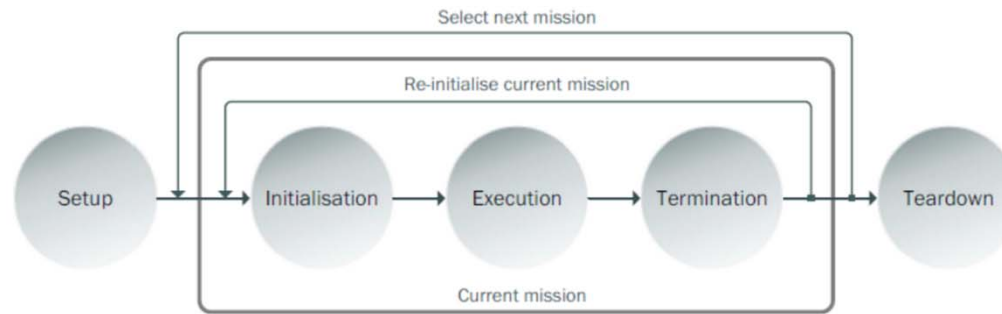
- Backward compatibility to standard Java
 - No Syntactic extension
 - Write Once, Run Anywhere
 - Reflected current real-time practice anno 1998
 - Allow implementation flexibility
-
- Does not address certification of Safety Critical applications

Safety-Critical Java (SCJ)

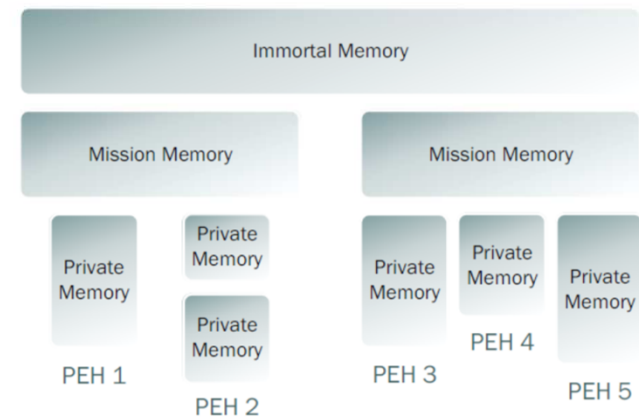
- Java Specification Request 302
- Aims for DO178B, Level A
- Three Compliance Points (Levels 0, 1, 2)
 - Level 0 provides a cyclic executive (single thread), no wait/notify
 - Level 1 provides a single mission with multiple schedulable objects,
 - Level 2 provides nested missions with (limited) nested scopes
- More worst case analysis friendly
- Restricted subset of RTSJ

SCJ

- Only RealtimeThreads are allowed
- Notions of missions and handlers



- No heap objects/ no GC
- Restricted use of scopes

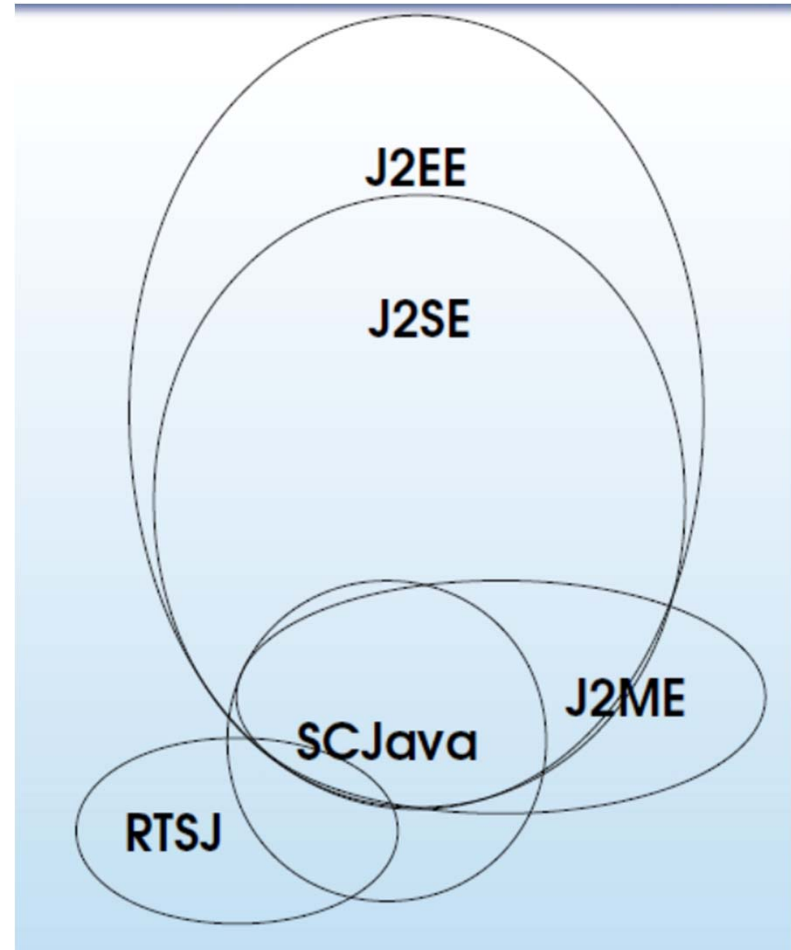


Predictable Java (PJ)

- Predictable Java intended as guidance/ideas for SCJ
- JSR-302 uses inheritance for limitation
 - Lots of @SCJAllowed annotations everywhere
- RTSJ would be a specialisation of a smaller profile
- PJ suggests to use inheritance for specialisation
 - Generalisation of RTSJ
- Missions are first-class handlers
 - Scoped memory belonging to the mission
 - No need for immortal memory known from RTSJ and SCJ.
 - Simplifies memory hierarchy
 - Programs are more Java like

Many variants of Java

- J2EE
 - J2SE & enterprise extensions
- J2SE
 - Standard Java
- J2ME
 - Subset of J2SE & additional classes
- RTSJ
 - Add on to J2EE, J2SE, or J2ME for realtime
- SCJava
 - Subset of RTSJ, subset of J2SE, & additional classes



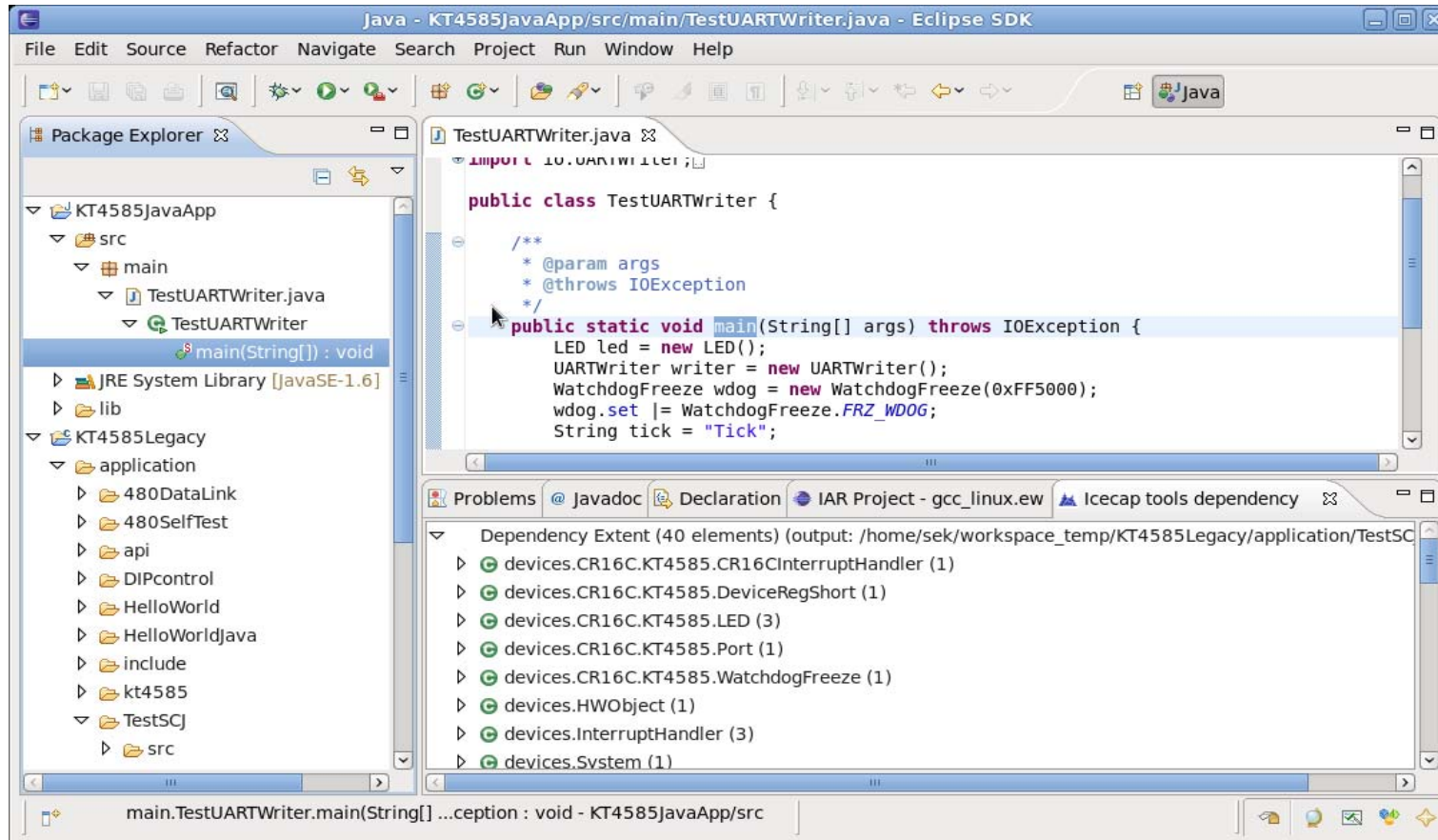
Predicatble JVM

- JOP
 - Java Optimized Processor
 - JVM in Hardware (FPGA)
- HVM
 - targeted at devices with 256 kB flash and 8kB of RAM
 - Interpreted or AOT compiling
 - 1st level interrupt handlers in Java
 - Runs on ATmega2560, CR16C, ARM7, ARM9 and x86
- JamaicaVM
 - Industrial strength real-time JVM from Aicas
 - Enroute for Certification for use in Airplanes and Cars



The HVM

Java-to-C compiler with an embedded interpreter



Java look-and-feel for low-end embedded devices

Support incremental move from C to Java

Features

- Execution on the bare metal
- First level interrupt handling & Hardware Objects
- Hybrid execution style (interpretation + AOT)
- Program specialization
 - * Classes & methods
 - * Interpreter
- Native variable support
- Portability
 - * No external dependencies
 - * Strict ANSI-C
- Process switching & scoped memory

The Predictable Real-time HVM

- Time predictable implementations of Interpreter loop and each bytecode

```
1 static int32 methodInterpreter(const
    MethodInfo* method, int32* fp) {
2     unsigned char *method_code;
3     int32* sp;
4     const MethodInfo* methodInfo;
5
6     start: method_code = (unsigned char *)
        pgm_read_pointer(&method->code, unsigned
            char**);
7     sp = &fp[pgm_read_word(&method->maxLocals)
        +2];
8
9     loop: while (1) {
10         unsigned char code = pgm_read_byte(
            method_code);
11         switch (code) {
12             case ICONST_0_OPCODE:
13                 //ICONST_X Java Bytecodes
14             case ICONST_5_OPCODE:
15                 *sp++ = code - ICONST_0_OPCODE;
16                 method_code++;
17                 continue;
18             case FCONST_0_OPCODE:
19                 //Remaining Java Bytecode impl...
20             }
21         }
22 }
```

What about Time Analysis?

Utilisation-Based Analysis

- A simple **sufficient but not necessary** schedulability test exists

$$U \equiv \sum_{i=1}^N \frac{C_i}{T_i} \leq N (2^{1/N} - 1)$$

$$U \leq 0.69 \text{ as } N \rightarrow \infty$$

Where C is WCET and T is period

41

Response Time Equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Where $hp(i)$ is the set of tasks with priority higher than task i

Solve by forming a recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

The set of values $w_i^0, w_i^1, w_i^2, \dots, w_i^n, \dots$ is monotonically non decreasing
When $w_i^n = w_i^{n+1}$ the solution to the equation has been found, w_i^0 must not be greater than R_i (e.g. 0 or C_i)

42

- Traditional approaches to analysis of RT systems are hard and conservative
- Very difficult to use with Java because of JVM (and Object Orientedness)

Model based Analysis

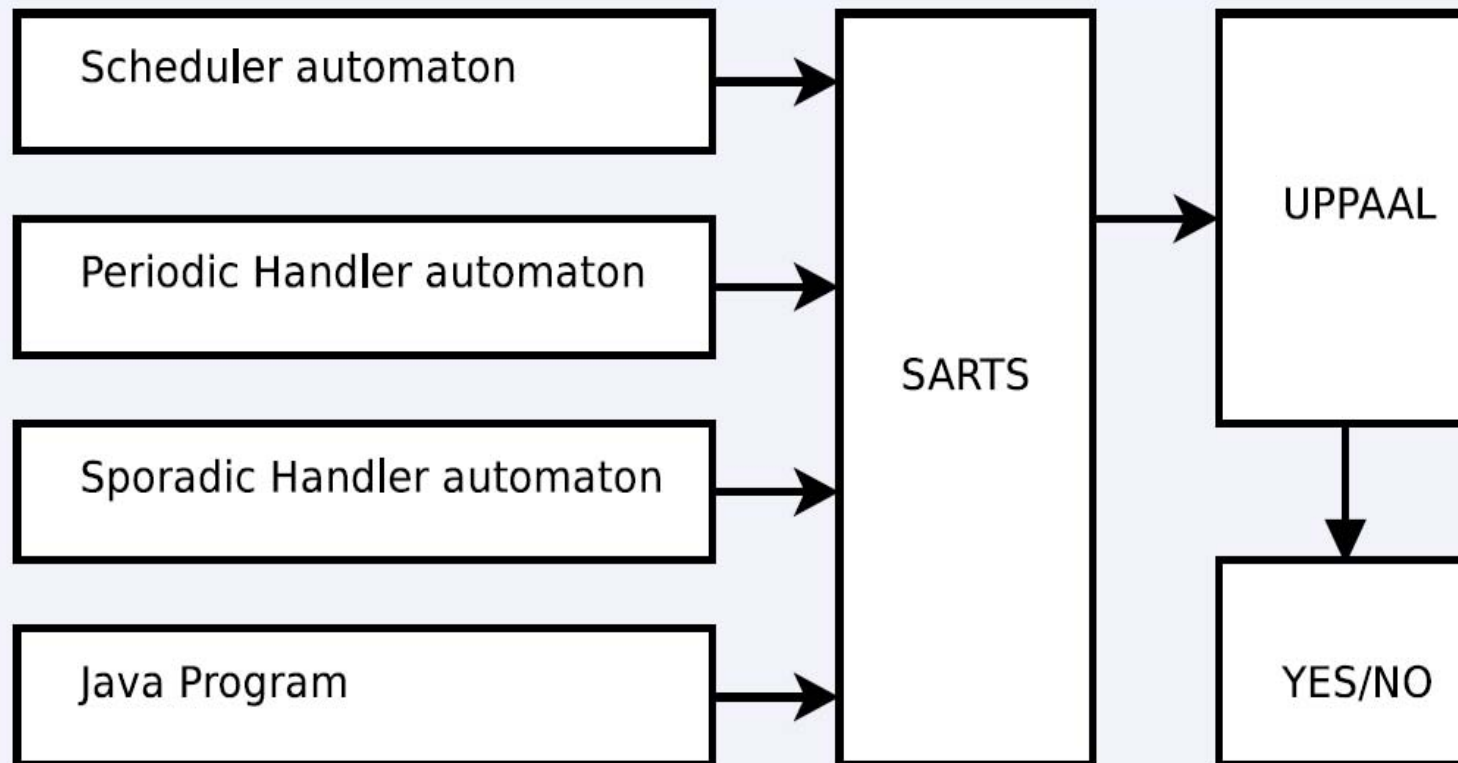
- TIMES
 - Model based schedulability tool based on UPPAAL
- WCA
 - WCET analysis for JOP
- SARTS
 - Schedulability on JOP
- TetaJ
 - WCET analysis for SW JVM on Commodity HW
- TetaSARTS
 - Schedulability analysis for SW JVM on Commodity HW and JOP

SARTS

- Schedulability analyzer for real-time Java systems
 - Assumes program in SCJ profile
 - Assumes correct Loop bounds annotations
 - Assumes code to be executed on JOP
- Generates Timed Automata
 - Control flow graph with timing information
 - Uppaal Model-checker checks for deadlock
 - Based on ideas from TIMES tool

SARTS Overview

UPPAAL model



SARTS Overview

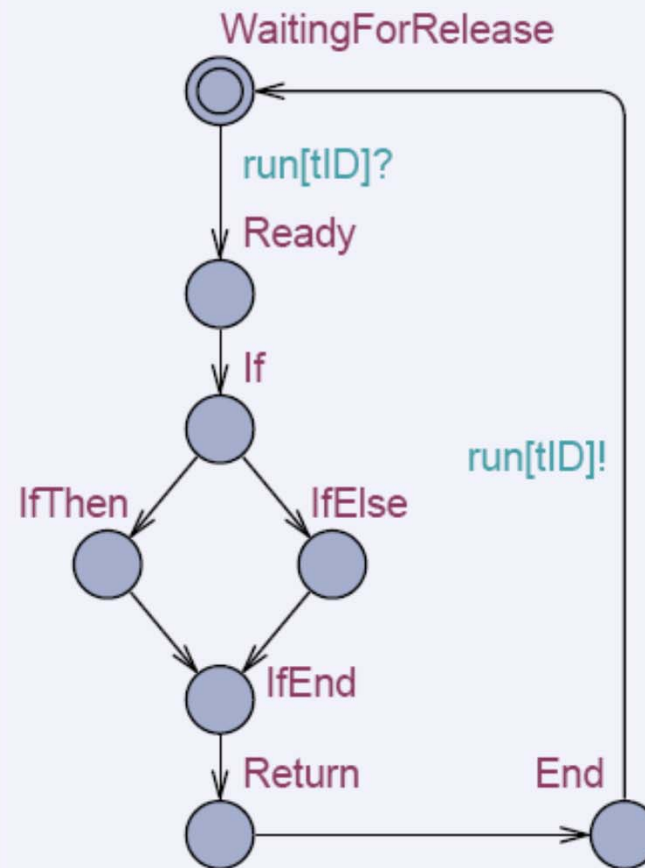
- A scheduler automaton models FPS
- A controller automaton, periodic/sporadic, is created for each handler
- Each Java method results in a parametrised automaton
 - One clock per task/thread
 - Pre-emption is modelled using stopwatches
 - Control-transfer is modelled using synchronization

Java to UPPAAL

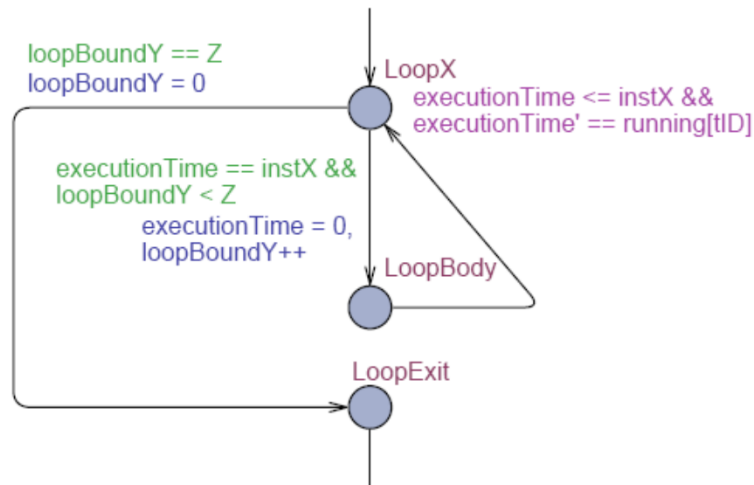
Java code

```
protected boolean run() {  
    if (condition){  
        //then statements  
    } else {  
        //else statements  
    }  
    return true;  
}
```

UPPAAL model



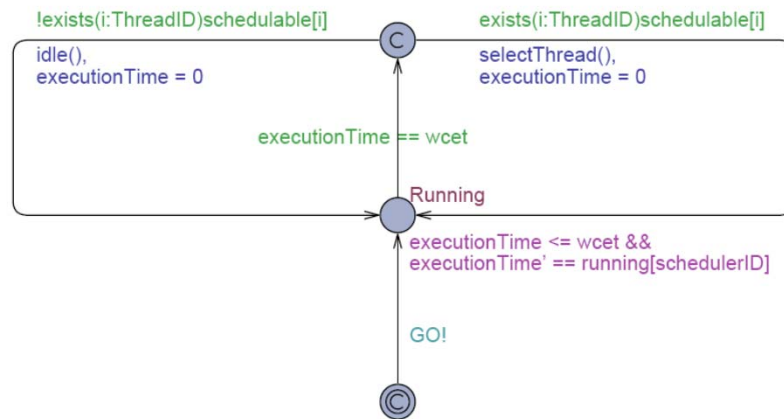
Timed Automata templates



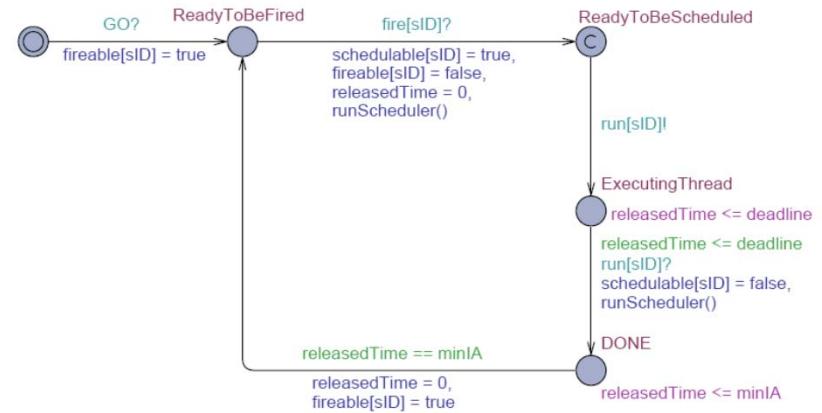
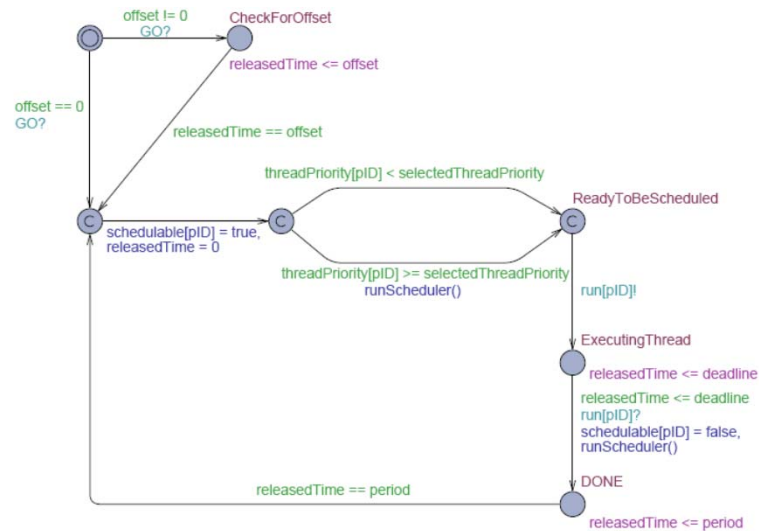
- Translation of Basic Blocks into states and transitions
- Patterns for:
 - Loops
 - Monitor statements
 - If statements
 - Method invoke
 - Sporadic task release

Simple models of RM scheduler

- Predefined models
 - Scheduler
 - Periodic Task
 - Sporadic Task



Periodic Task/Sporadic Task



SARTS sales pitch

- The schedulability question is “translated” to a deadlock question
 - no deadlock means schedulable
- Compared to traditional schedulability analysis
 - Control flow sensitive
 - Fine grained interleaving
 - Less pessimism
 - Fully automatic

SARTS can do better than utilisation test

- Example
- One periodic task
- Two sporadic tasks
 - Mutually exclusive

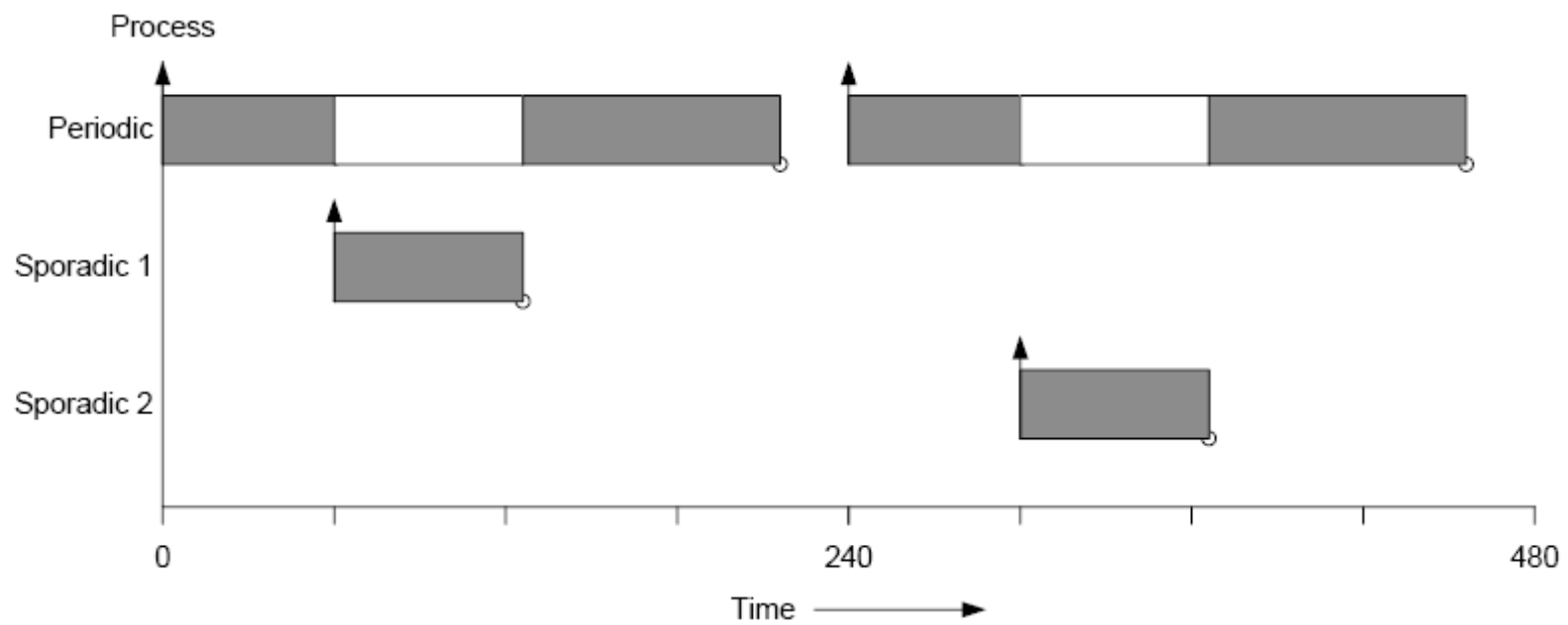
```
public class Experiment2 extends PeriodicThread {  
    public boolean run() {  
        if (b) {  
            RealtimeSystem.fire(1);  
        } else {  
            RealtimeSystem.fire(2);  
        }  
        return true;  
    }  
}
```

SARTS can do better than utilisation test

- Period: 240
- Minimum inter-arrival time: 240
- Periodic cost: 161
- Sporadic cost: 64
- Utilisation test fails:

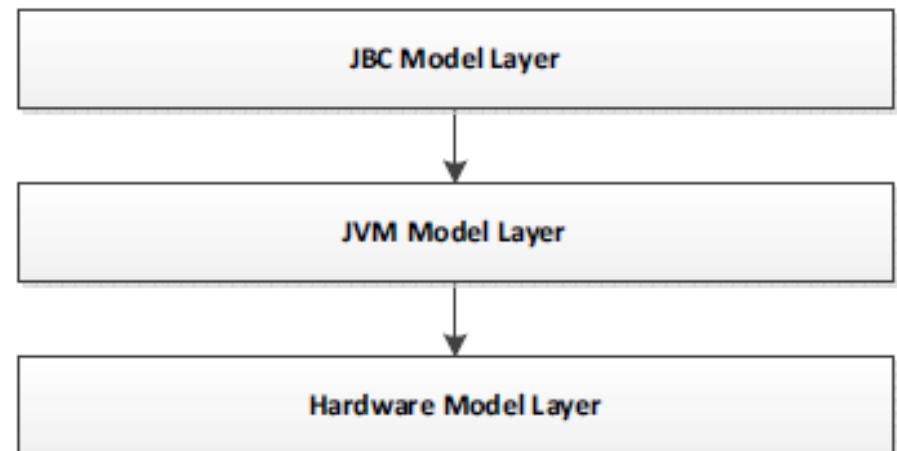
$$\left(\frac{161}{240}\right) + \left(\frac{64}{240}\right) + \left(\frac{64}{240}\right) = 1.20$$

Time Line



TetaJ

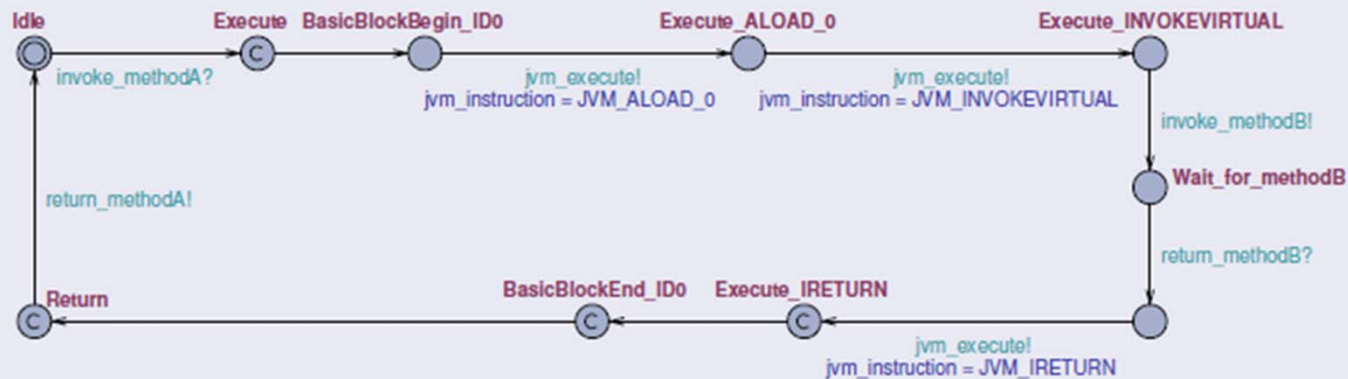
- WCET analysis tool
 - taking Java portability into account
- Analysis at method level
- Can be used interactively
- Takes VM into account
- Takes HW into account



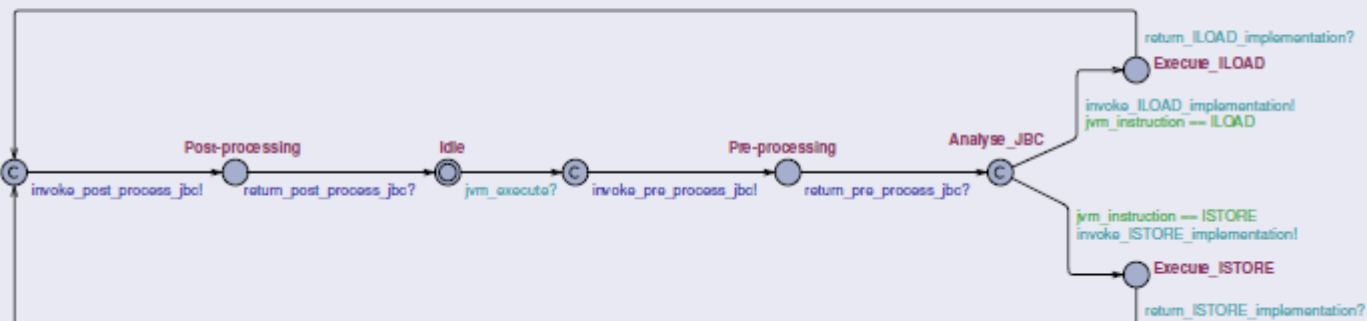
Initialisation Model



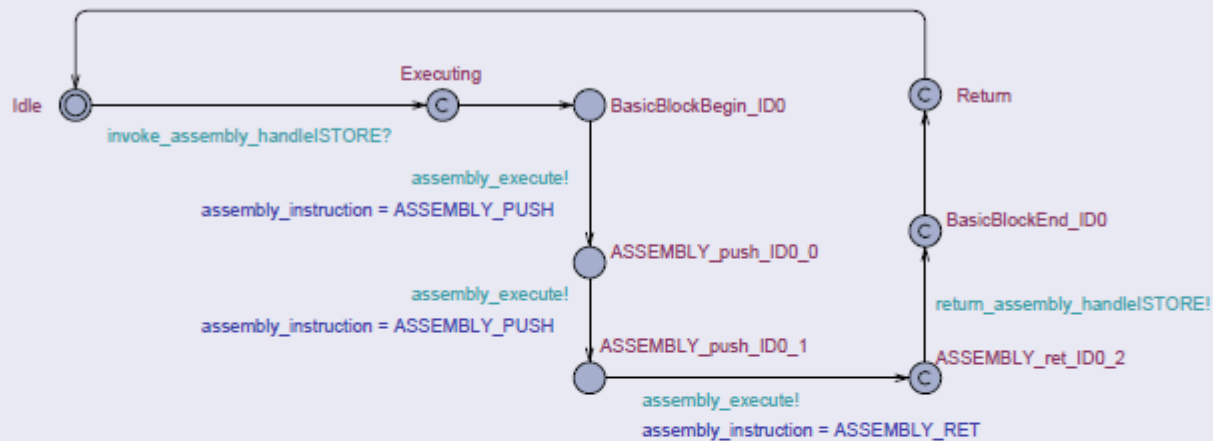
Program Model



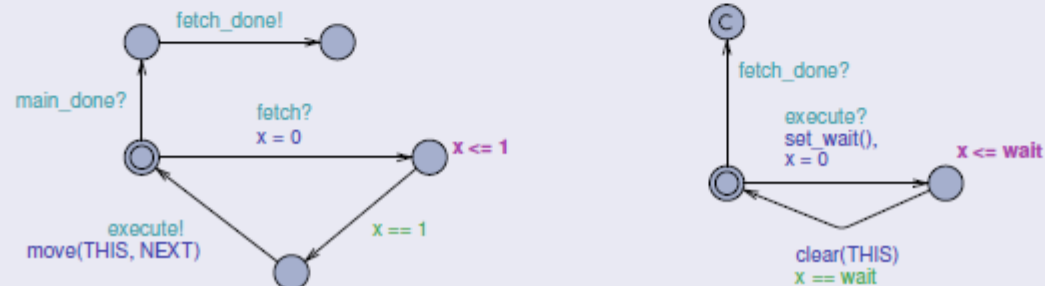
JVM Model (excerpt)



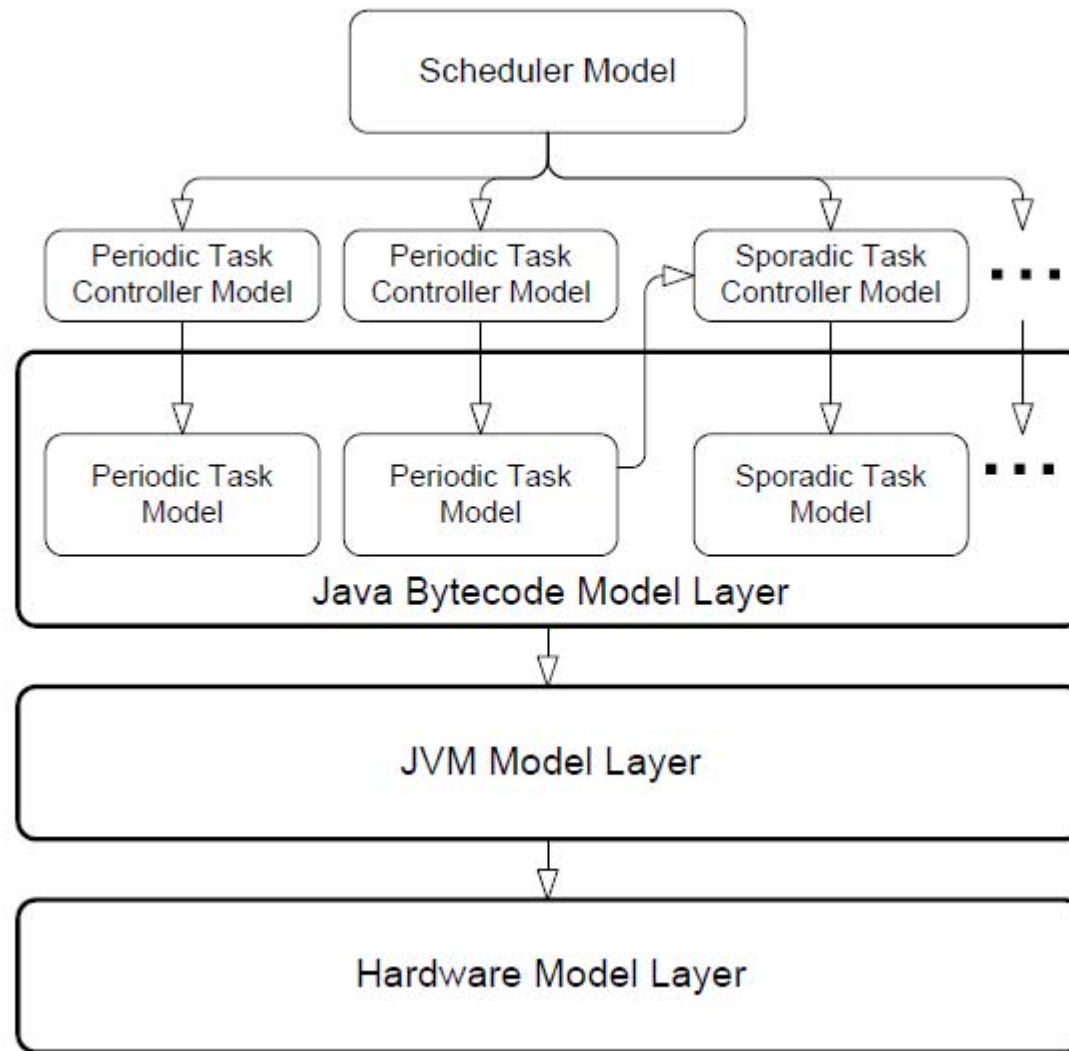
Java Bytecode Implementation



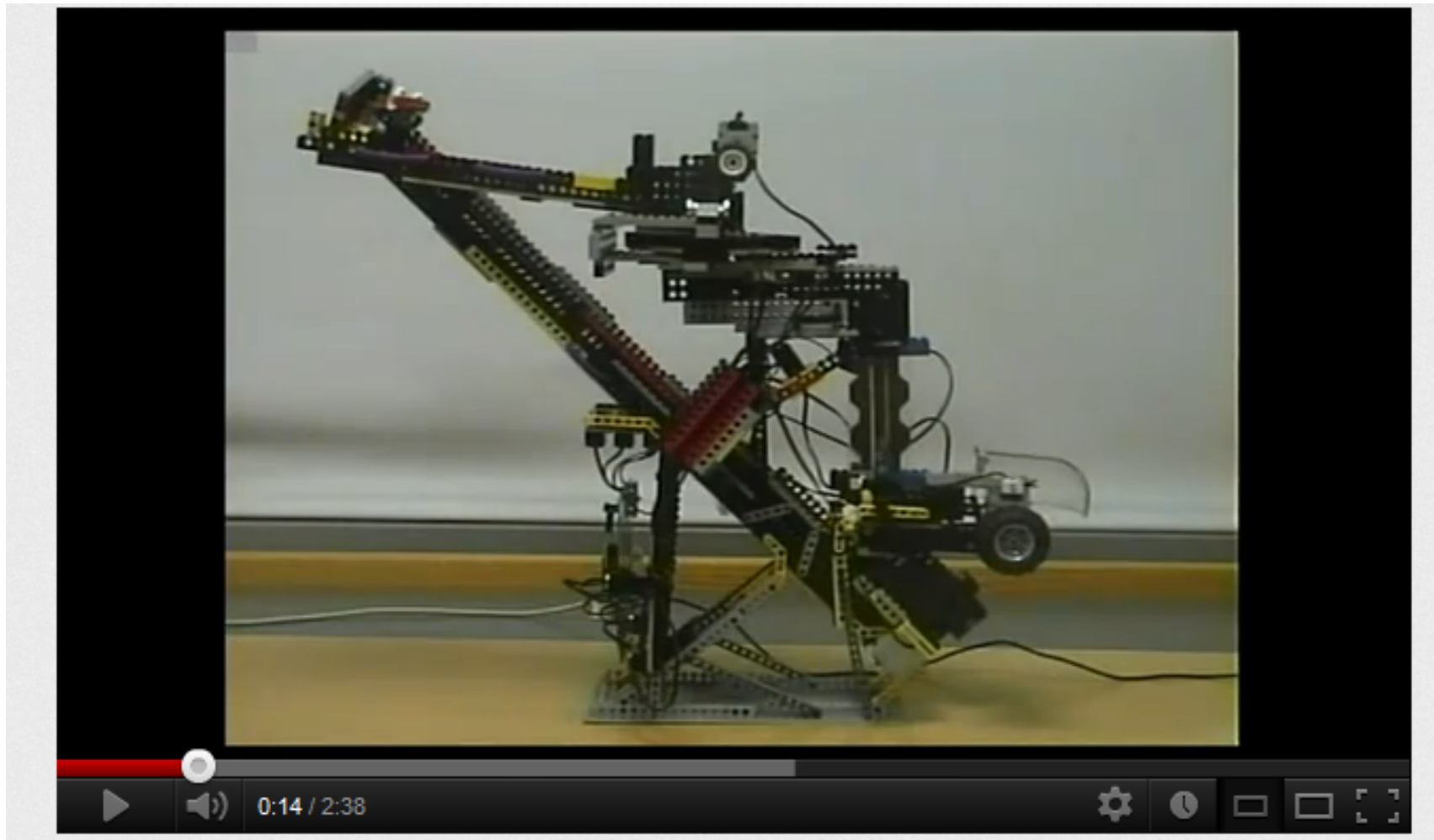
Hardware Models (From METAMOC)



TetaSARTS



Minepump example



Minepump example

Write once – run wherever possible

Execution Environment	Water Deadline	Methane Deadline	Schedulable
HVM + AVR @ 10 MHz	12 ms	12 ms	✓
HVM + AVR @ 5 MHz	12 ms	12 ms	×
HVM + AVR @ 10 MHz	6 ms	6 ms	×
JOP @ 100 MHz	6 ms	6 ms	✓
JOP @ 100 MHz	12 μ s	12 μ s	✓

Table 2. Using TetaSARTS with various execution environments.

Experiment	Exec. Env.	Optimised	Analysis Time	Mem. Usage
Minepump	HVM + AVR	✓	15h 25m 16s	17933 MB
Minepump	JOP	✓	7s	27 MB
Minepump	JOP	×	6m 18s	62 MB
SARTS Minepump	JOP	N/A	21s	42 MB
Simple System	HVM + AVR	✓	49s	168 MB
Simple System	HVM + AVR	×	22m 58s	238 MB
Simple System	JOP	✓	0.05s	7 MB
Simple System	JOP	×	0.5s	20 MB

Table 1. Results obtained using TetaSARTS and SARTS.

Energy Optimize Applications

Execution Environment	Clock Freq.	Schedulable
HVM + AVR	10 MHz	✓
HVM + AVR	5 MHz	×
JOP	2 MHz	✓
JOP	1 MHz	×

System	Clock Freq.	Proc. Util.	Proc. Idle
RTSM	100 MHz	48.5 μ s	4.0 ms
RTSM	60 MHz	80.8 μ s	4.0 ms
Minepump	100 MHz	25.9 μ s	2.0 ms
Minepump	10 MHz	259 μ s	11.8 ms

Compositional Verification

- TetraSARTS generates model for whole program
- Library routines analysed again and again
- Models based on control flow can be complicated
- Idea: Annotate interfaces with abstract description of behaviour
 - Time and Resource Specification Language (TRSL)
 - Could have been any of a range of spec. lang.
 - UML/Marte, ACSR, TADL

```

class Task2 extends PeriodicEventHandler{
    Buffer buf;                // shared buffer
    //@ TRSL = [5]
    private int calculate(){..}
    //@ TRSL = [2]
    private void prepare(..){..}
    //@ TRSL = [1]
    private void register(..){..}
    //@ TRSL = [1 ; 7? ; using(r)[2] ; 1 ]
    public void handleEvent(){
        if(!ready){           // wcet: 1
            value = calculate(); // wcet: 5
            prepare(value);      // wcet: 2
        }
        input = buf.remove();   // wcet: 2
        register(input);        // wcet: 1
    }
}

```

Note – could have used [1..8 ; using(r)[2] ; 1] since
[1 ; 7? ; using(r)[2] ; 1] $\leq$ [1 ..8 ; using(r)[2] ; 1]

TetaSARTS+

- Schedulability analysis now in three steps
 - Verify that implementation is simulated by specification
 - Check $L(\text{Implementation}) \leq L(\text{specification})$
 - Possible since TRSL TAs are simple instances of the Event-Clock Automata
 - Generate TAs from Specs
 - Use TetaSARTS

Further Analysis and tools

- Scope compliance analysis for SCJ
- SCJ compliance analyzer
- Eclipse plug-in
- Lot's of work on (analyzable) Real-time GC

Future Work

- Experiment with deductive verification
 - Functional requirements
 - JML and Key
 - Especially loop bounds
- Symbolic model checking
 - JavaPathFinder
- Termination Analysis
 - Recursion bounds
- Analyse non-SCJ programs
 - Java, Groovy, Scala
- Multi-core HVM

Learn more

- Model-based schedulability analysis of safety critical hard real-time java programs
 - T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen
 - JTRES 2008
- Schedulability Analysis Abstractions for Safety Critical Java
 - Thomas Bøgholm, Bent Thomsen, Kim G. Larsen, Alan Mycroft
 - ISORC 2012
- Wcet analysis of java bytecode featuring common execution environments
 - C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen
 - JTRES 2011
- TetaSARTS: A Tool for Modular Timing Analysis of Safety Critical Java Systems
 - Kasper Luckow, Thomas Bøgholm, Bent Thomsen, and Kim Larsen
 - To appear JTRES 2013

Join InfinIT network on High Level Languages in Embedded Systems

- http://www.infinitt.dk/dk/interessegrupper/hoejniveau_sprog_til_indlejrede_systemer/hoejniveau_sprog_til_indlejrede_systemer.htm

Try it out?

- TetaSARTS
 - <http://people.cs.aau.dk/~luckow/tetasarts/>
- Hardware Near Virtual Machine
 - <http://icelab.dk/>
- oSCJ (open Safety-Critical Java Implementation)
 - <http://sss.cs.purdue.edu/projects/oscj/>
- Java Optimized Processor
 - <http://www.jopdesign.com/>
- JamaicaVM
 - <http://www.aicas.com/jamaica.html>

Joint work with:

- Allan Mycroft
 - Cambridge University
- Hans Søndergaard, Stephan Korsholm
 - Via University College
- Thomas Bøgholm, Kasper Søren Luckow, Anders P. Ravn, Kim G. Larsen, Rene R. Hansen and Lone Leth Thomsen
 - CISS/Department of Computer Science, Aalborg University