MULTI-STAGE DEPLOYMENT OF ROBOT CONTROL SOFTWARE

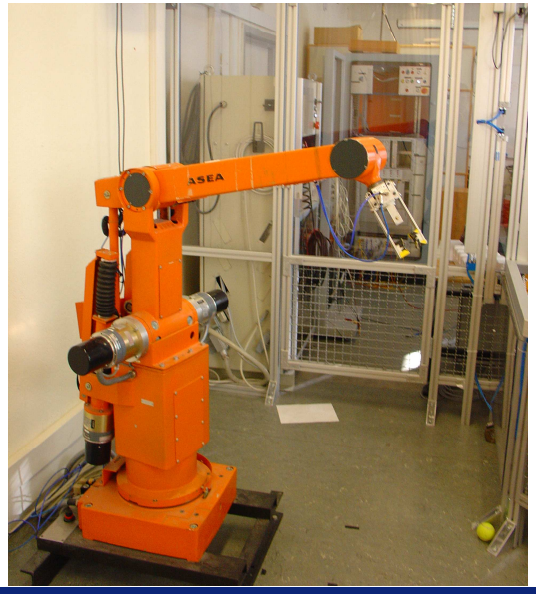
A Java-based approach by

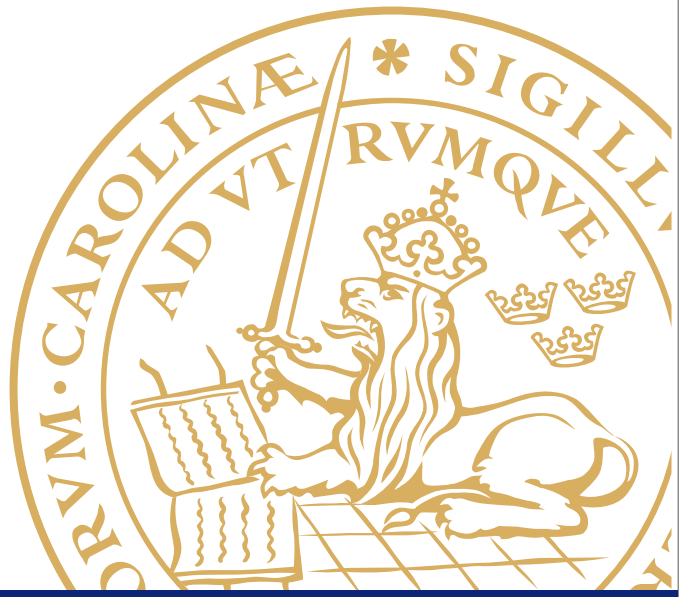
Sven Gestegård Robertz & Anders Nilsson

& ***Klas Nilsson*** & Mathias Haage

Dept. of Computer Science, Lund University, Sweden

{sven|andersn|klas|mathias}@cs.lth.se





Robotic developments hampered by too complex/costly software engineering.

1. How can we trust that a deployed software component/function doesn't harm other (previously tested) parts of the control system?

- Use of a **safe** programming **language such as Java/C#** (with a strictly maintained sandbox model) for manually written code.

2. How can we stepwise deploy embedded control software such that the control and timing properties are verified in multiple small stages?

- **Multistage deployment** strategy, ranging from portable desktop-suitable simulation to cross compilation into target-specific hard real-time software functions.

Except for device drivers and automatically generated code we take a Java-based approach..... Why and how??.....

Deployment stages

1a) Running both the application and ***simulated environment*** in a ***standard JVM*** (J2SE from Sun) on a workstation.

1b) Running both the application and the simulated environment in a standard JVM on a workstation, using the ***free*** Java ***library*** classpath from GNU.

2) ***Natively compile*** application using LJRT, ***run on desktop*** in simulated environment.

3) Running natively compiled LJRT application with ***POSIX threads on target system***.

4a) Running natively compiled LJRT application with ***native RTOS threads in user space*** on target.

4b) Running natively compiled LJRT application with native RTOS threads, in **kernel space, on target system**.

Primary RTOS: www.xenomai.org

Robot control depends on real-time software; use Java for portability and modularity!?

Typically experienced questions:

- **Why** would you do such a thing?
  The most industrially acceptable, freely available and portable way of imposing the modularity needed for robot software development: The Java **language**
- **How** would you do such a thing?
  As in the Lund Java-based Real-Time (LJRT) platform.

Java-based: *Full Java language with library subset.* ***Runs on any J2SE VM but not Java legally****. Cross-compilable to embedded targets by our compiler and run-time system.*

Problems in software development

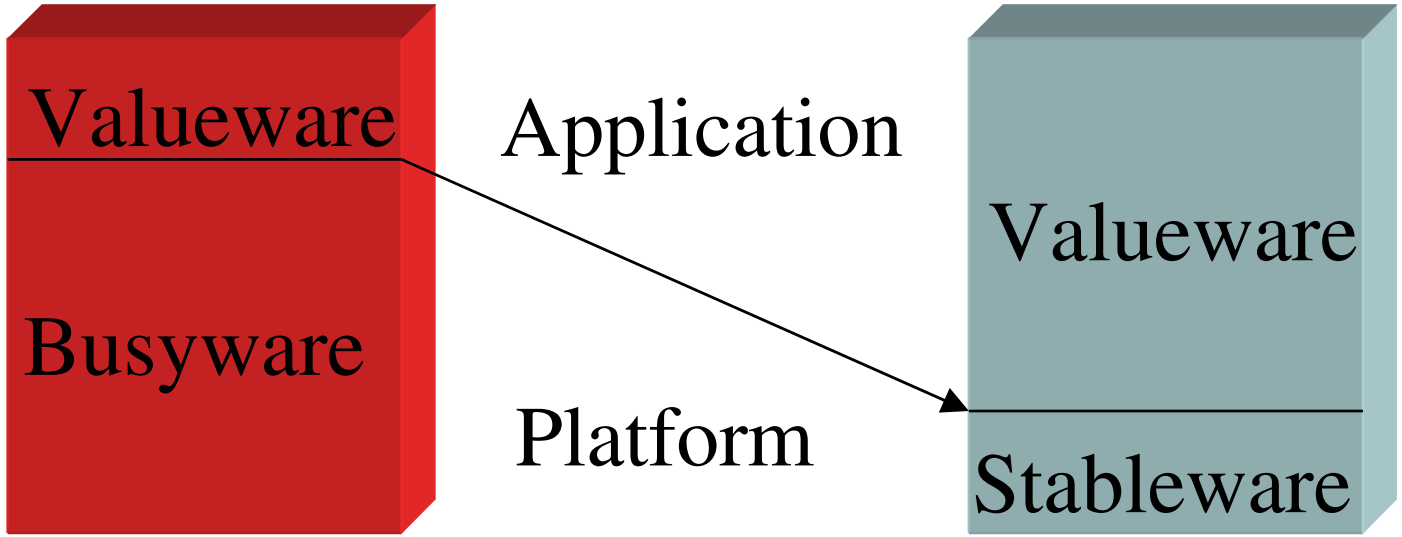
- Managing System Complexity
  Complex systems, weak structuring mecahnisms make it worse
- Managing System Development
  Late project, late errors makes it worse

What is the role of languages in this ?

  - Errors detected eariler, in process, in tools
  - Errors avoided

Problem

- Engineering efforts spent on BusyWare; less profit from the ValueWare.
- How to accomplish useful StableWare?



Scalability

Scalability

- Learned from history: **only scalable solutions survive**.
- Large systems
- Tiny systems

Considering:

- Interoperation
- Wireless comm.
- Open systems
- Virtual reality
- etc.

Scalability(safety)→Java/C#

- Composing components and plugins for RT: Safety?
- Ensuring enabled error handling: **Safe** language required! (*All possible executions are expressed by the program*.)
- Performance and predictability: Static type checking (whenever possible).
- Automatic dynamic memory management.

Unsafe language mechanisms

- Manual memory management (malloc-free)
  - When to do free? - ”when last pointer removed”!
  - Too early - dangling pointers
  - Too late - memory leaks
- Cast as in C
- Pointer arithmetic
- Arrays with no boundchecks
  - Programmer error leads to chaos

- Problems often show up late, sometimes only after long execution times
- Program-wide consistency problem
- How to trust your robot?

Programming languages, the rudimentary Java view

Human

Natural:

If level x greater than zero
then …

In program:

if (x>0) { … /*C, Java*/

if x>0 then ... -- Ada

Machine

- *Mac (PowerPC)*

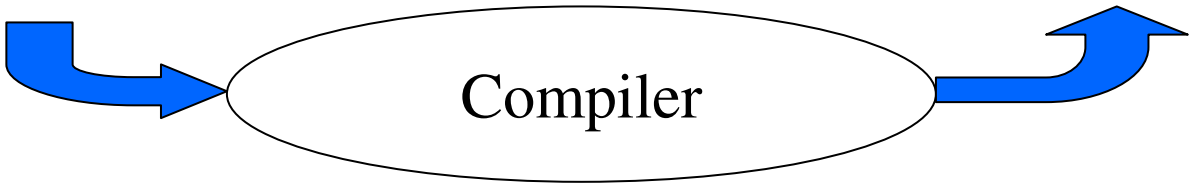
```
lwz R0,x(R1)        80 01 00 08
cmpwi CR1,R0,0      2c 80 00 00
bc 4,5,else         40 85 00 0c
```

- *PC (Pentium)*

```
cmpl 0,x(%ebp)      83 7d 08 00
jle else            7e 07
```

- *JVM (Java bytecode):*

```
iload x             15 xa
ifle else           9e e1 e2
```



Misconceptions

- Just another programming language.
- Java is slow.
- Not for real time due to GC.
- Large memory footprint.
- Language $\leftrightarrow$ OS $\leftrightarrow$ JVM.
- Cannot utilize dedicated hardware.

Technology to promote methodology

Support programmer by both techniques and methods.

- Dr Dobb's, June 2005 (John Dibling):
  "When I was a rookie C++ programmer, I thought that the key to writing code that was defect free was to know more C++, more techniques, more tricks. I don't think that anymore, and I'm much happier."
- "But you should write your C/C++ program such that....", but you (or someone adding a feature later) does not.
- "Well, with proper software engineering and testing you can develop correct code." OK for a device driver or a JVM, but rapidly writing robot control functions.....

Errors, correctness and tool support

Software error types:

Syntactical errors: Caught by the compiler and should thus not give problems when running the application.

Logical errors: The application has unintentionally, by mistake or erroneous thinking, been given different semantics than what was intended. Mostly platform independent.

Concurrency errors: Failing to protect shared resources, and/or taking resource locks in the wrong order, may, depending on the thread interleaving pattern, lead to run-time errors (such as deadlocks or corrupted data). May, but should not, depend on actual run-time environment and thread model.

Timing errors: Real-time threads missing deadlines or starvation of lower priority threads. Depends on run-time environment and thread model.

Considerations

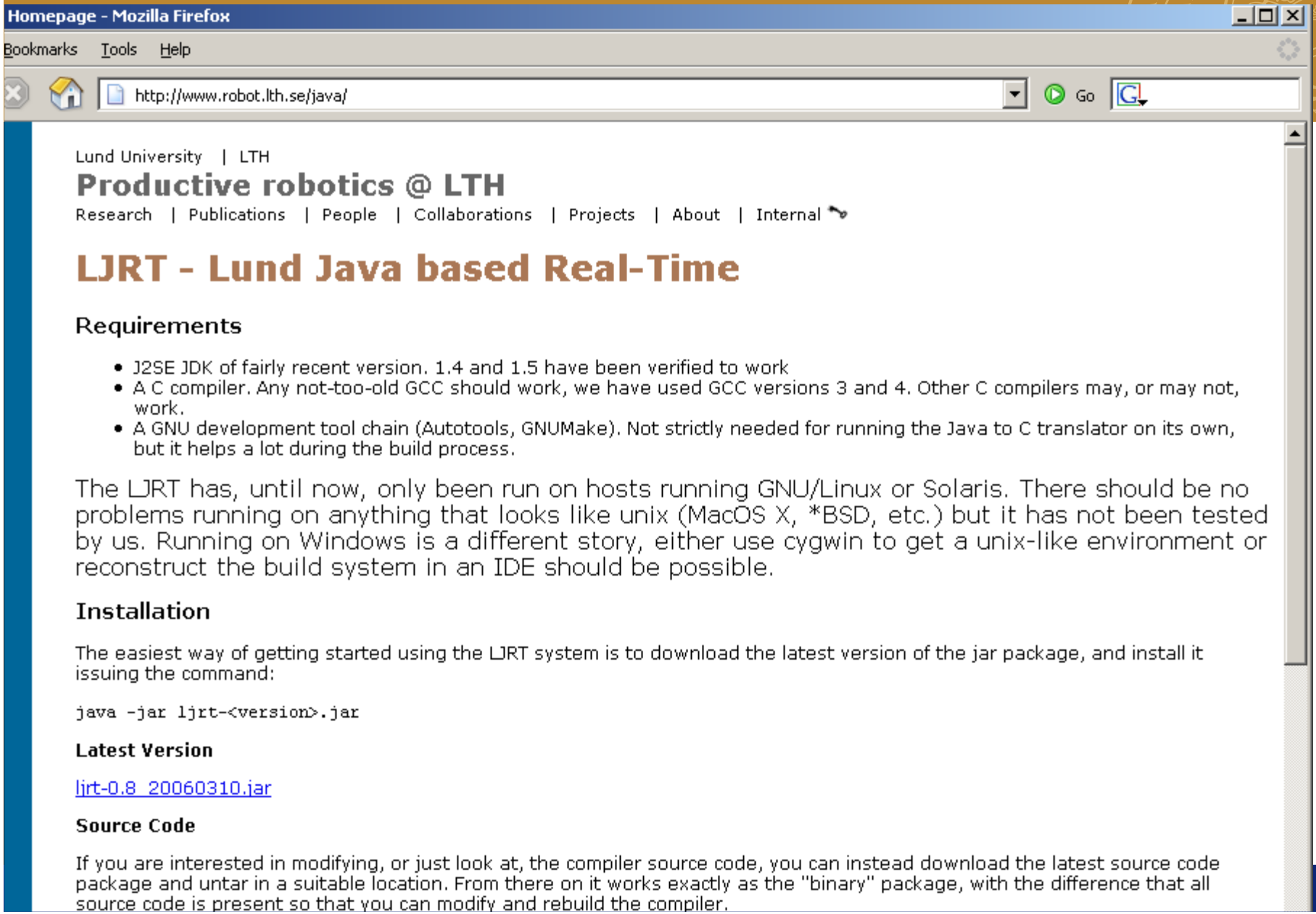
- Where is the fault (who to blame) when the robot software fails? Clear separation between platform and application!
- Testing, monitoring debugging.
- Formal methods.....
- Code generation from engineering tools.
- Resources and composability?

Answeres from Safe languages

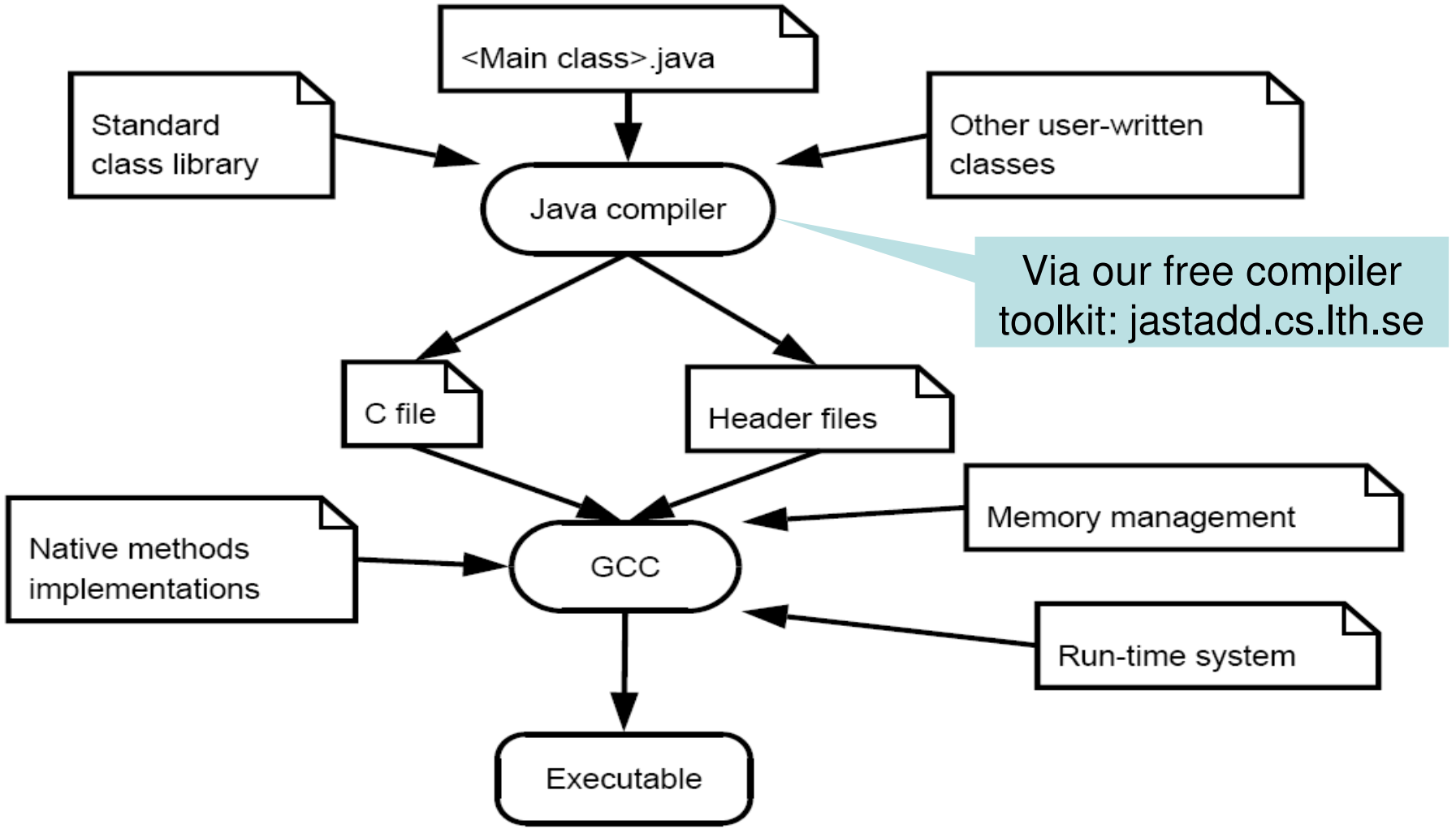
- Many errors catched by compiler
  - Remaining ones by runtime checks
  - Costs runtime efficiency
- Automatic memory management
  - When last pointer to object removed, object will be removed by Garbage Collector
  - Used to disrupt execution - not anymore
- Programming error leads to error message
  - not uncontrolled execution. (= illegal memory access, blue screen, …)
  - Uncontrolled execution indicates error in the platform - not in the program

Thus, a better separation between appliacation and platform.

Free downloadable software



Java compilation via C



Conclusions

- Four-stage deployment strategy
- Free real-time Java-based platform
- Compiling Java to C to native binaries for full efficiency
- Safe language: The sandbox model for application software, and the separation of the application from the platform, needed in future robotics.
- Download from www.robot.lth.se/java
  alpha available, beta soon to appear, stay tuned!

The ”principle of superposition” needed for modular development

Important need: Composable components and computations!

The principle of Super-position

Separation into sub-problems which solutions together comprise the solution to the overall problem.

- Used in physics, mathematics, solid-state mechanics, electro-technology, etc.
- Within mechanics and electronics there are composable components.
- **However, missing composability for software and hence for robotics...**

⇒ Too extensive engineering needed for development of mechatronic systems.

Technical resources: embedded systems

Resources in embedded systems

- Timing (CPU, HW,..)
- Memory (#bytes,..)
- Communication
- Device/unit-physical (energy, ..)
- Engineering effort

Bounded and interconnected, use optimally for best possible product properties and profit

Cost:
- Production
- Market opportunity

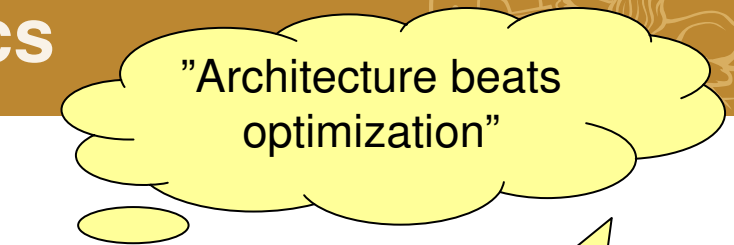
Interface:
- Interoperability
- Openness
- Usability
- Client satisfaction

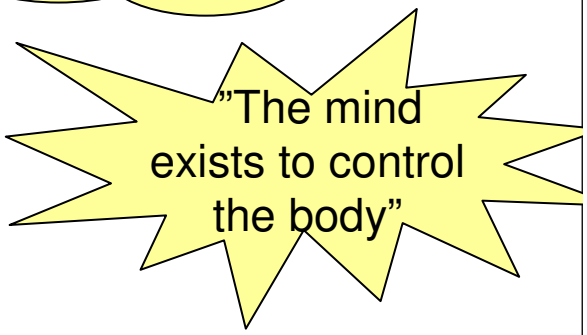
Adaptation:
- Portability
- Modifiability
- Evolvability
- Expandability
- Flexibility
- Configurability
- Reusability
- Scalability

Performance:
- Performance (Speed)
- Timeliness (Deadlines)
- Determinism
- Security
- Robustness
- Reliability
- Availability
- Safety

Design:
- Feasibility
- Maintainability
- Understandability
- Correctness
- Simplicity
- Integrability
- Testability&Debugging

The software crisis in robotics





Embedded information processing:

- Expensive monolitic systems today.
- Scalable software technology needed.

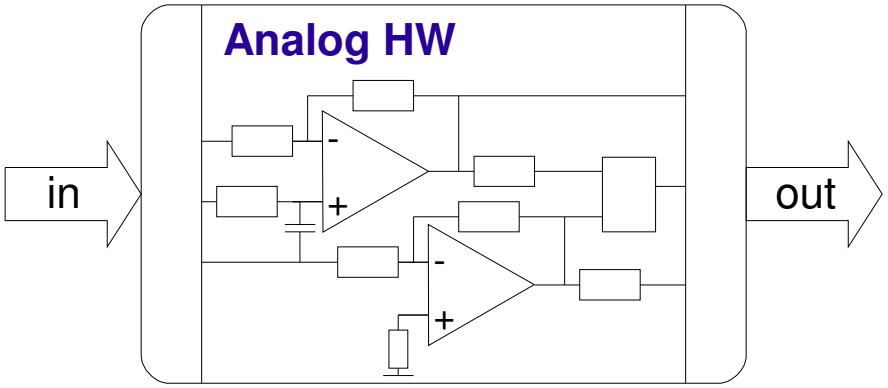
<u>Current and future research:</u>

1. Support for modular development and use of robotic software components, to enable modular robots that can assist humans in a flexible manner.
2. Enhanced technologies for implementation of control systems.

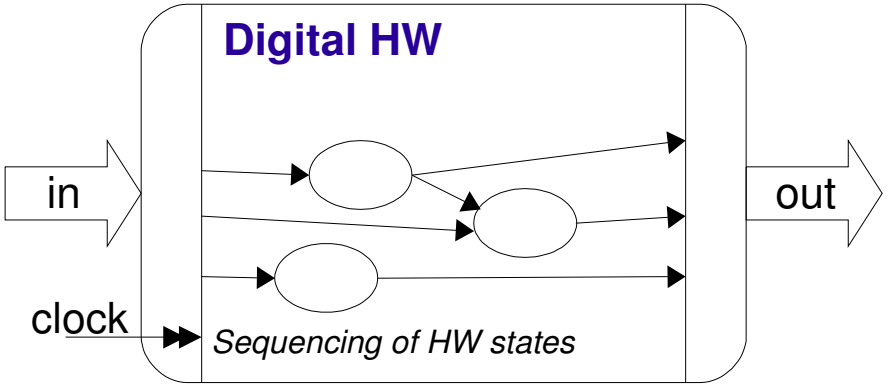
Resources for embedded systems, developments from past to present (for every improvement new drawbacks have resulted):

Embedded information processing: Going from Analog to Digital (still hardware)

+ Direct effect
+ No quantization
+ Truly parallel
– Cost
– Repeatability (drift, etc.)

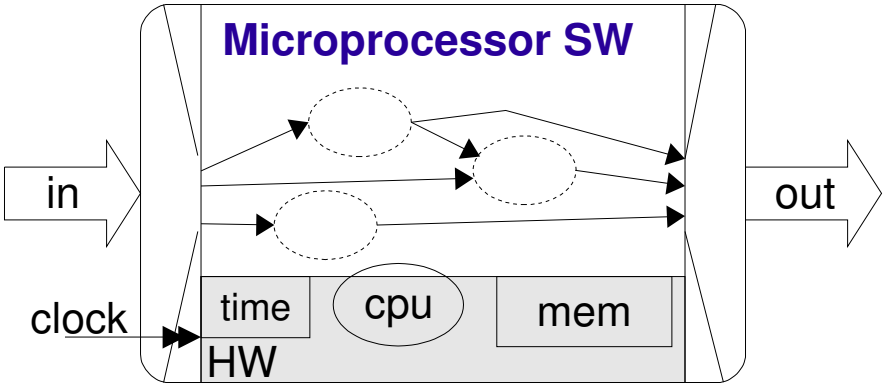


+ **Cost**
+ **Repeatability**
+ Truly parallel
– Quantization
– Latencies

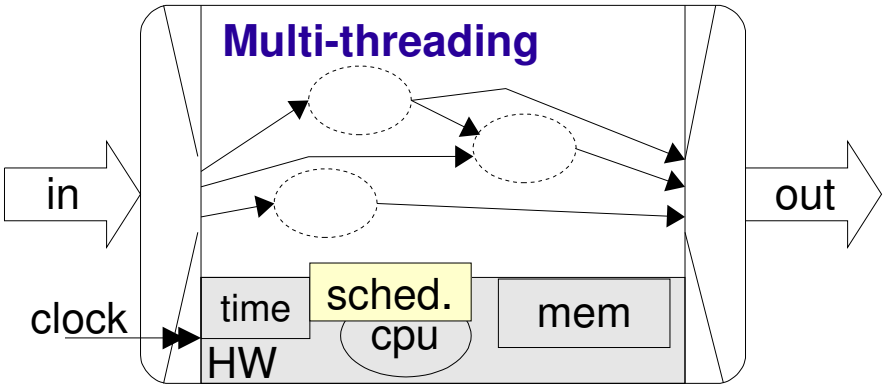


Then programmable units (software), Multitasking RTOS (Real-Time Operating System)

+ **Cost reduction**
+ **Programability**
– Modularity (resources)
– Delays
– Timing variations (jitter)

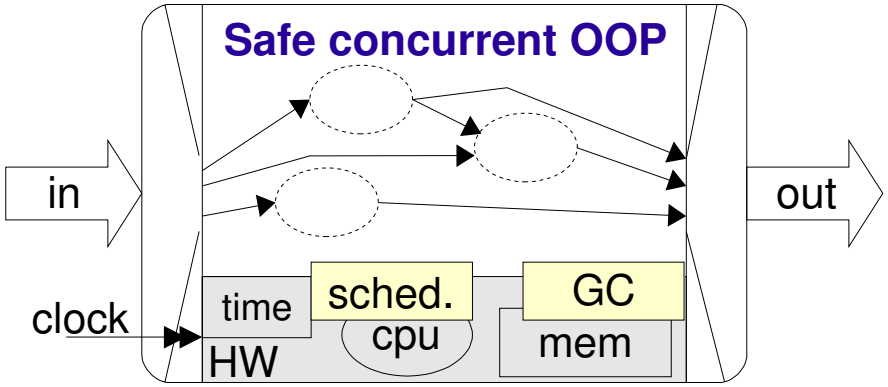


+ **Modularity of execution**
+ Flexibility
– Predictability
– Resource management
– Modularity

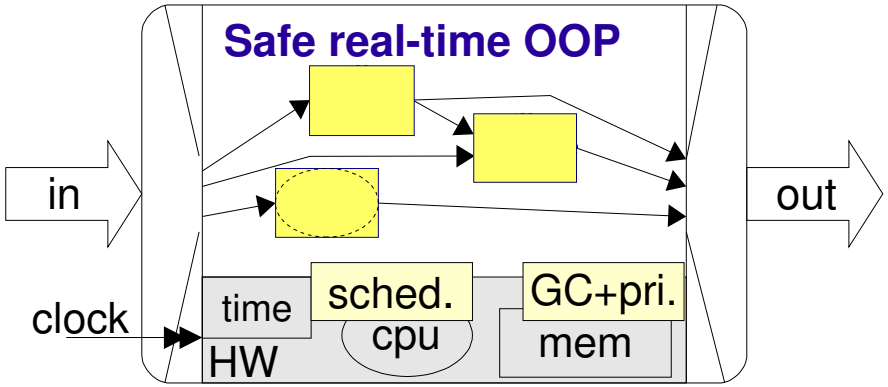


Software components using safe language, first concurrency and then for real-time

+ **Modular reactivity**
+ **Safety**
– Modularity (IO, memory)
– Timing variations

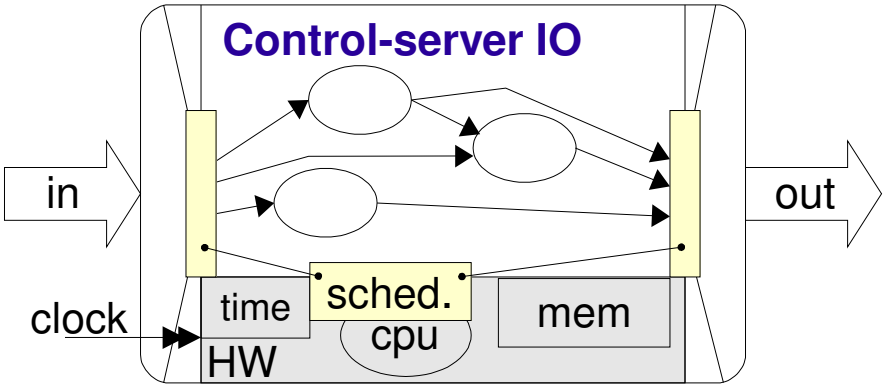


+ **Modular real-time**
+ **Robustness**
+ **Portable compilation**
– Resource optimization
– Timing variations for IO

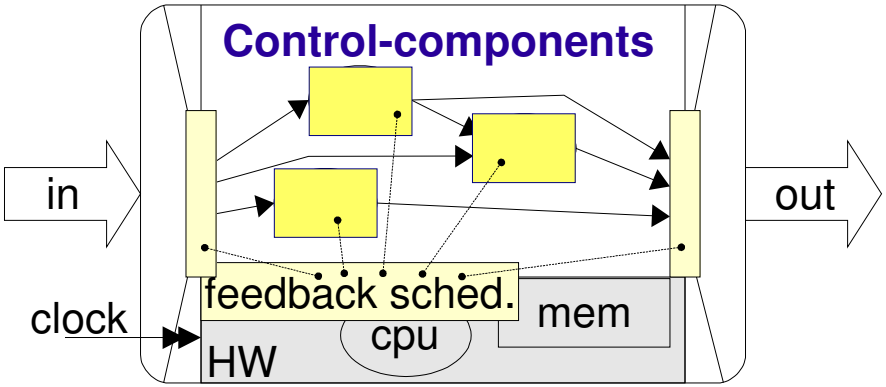


Control components, scheduled IO, then feedback scheduling of resources

+ **Virtual CPUs**
+ **Composable IO**
– Global memory
– Resource optimization
– Safety



+ **Performance tuning**
+ **Control components**
– Global memory
– Resource optimization
– Safety



Towards the ”principle of superposition” for embedded software

Ongoing integration and further development:

1) Object-oriented and portable safe real-time SW
2) Control components as composable SW

⇒ Resource-aware components & control systrems!