Physical Programming: Beyond Mere Logic

Bran Selic
Rational Software Canada
bselic@rational.com

In Memoriam

Edsger Wybe Dijkstra (1930 – 2002)

- *"I see no meaningful difference between programming methodology and mathematical methodology"* (EWD 1209)
- *"[The interrupt] was a great invention, but also a Pandora's Box. ….essentially, for the sake of efficiency, concurrency [became] visible… and then, all hell broke loose"* (EWD 1303)

Two Opinions

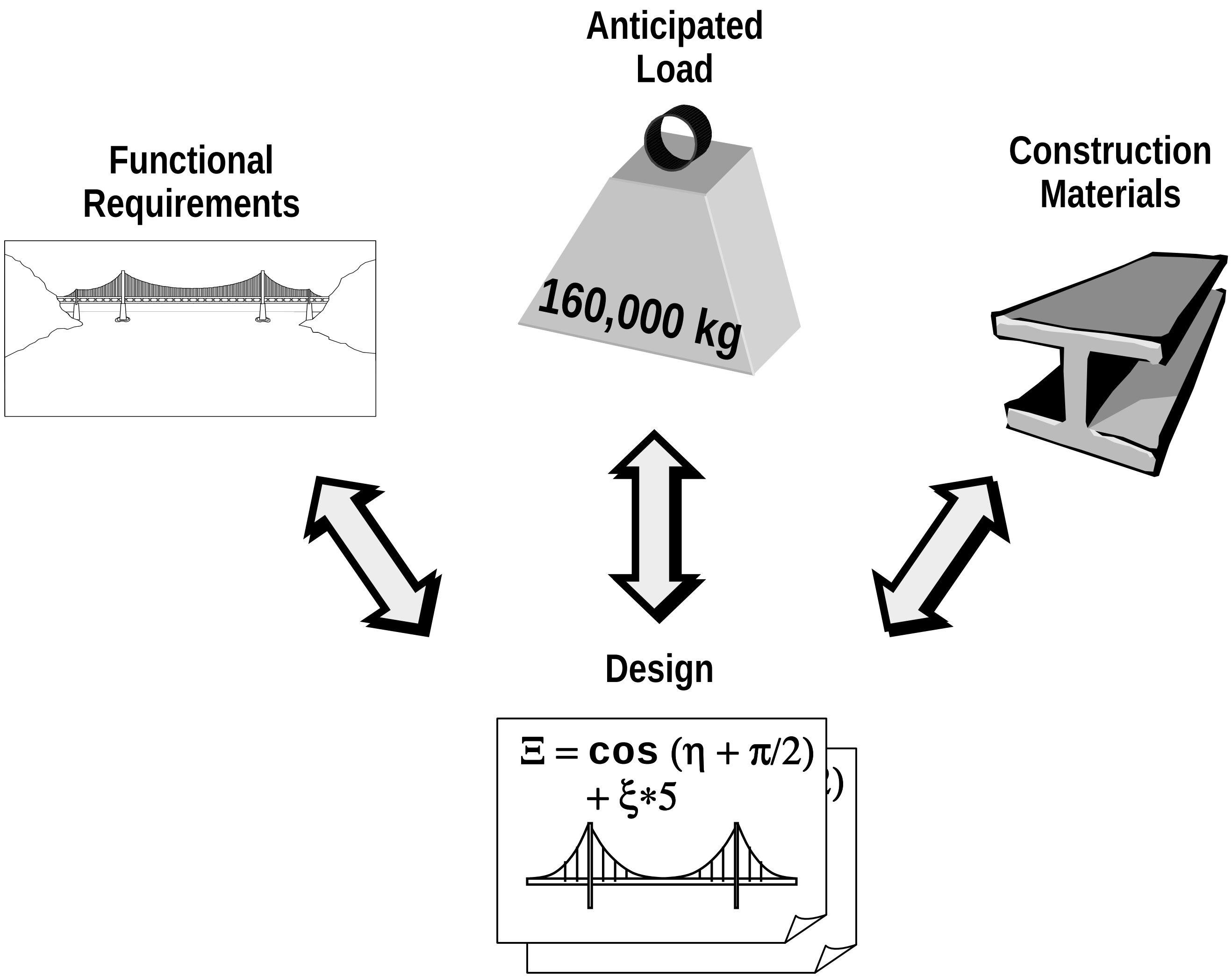
"Because [programs] are put together in the context of a set of information requirements, <u>they observe no natural limits</u> other than those imposed by those requirements. Unlike the world of engineering, there are <u>no immutable laws to violate</u>."

- Wei-Lung Wang
Comm. of the ACM (45, 5)
May 2002

"<u>All</u> machinery is derived from nature, and is founded on the teaching and instruction of the revolution of the firmament."

\- Vitruvius
On Architecture, Book X
1st Century BC

The Classical Engineering Design Problem



What is Software Made of?

Exhibit A: Transmission Delay Effects

- Possibility of out of date status information

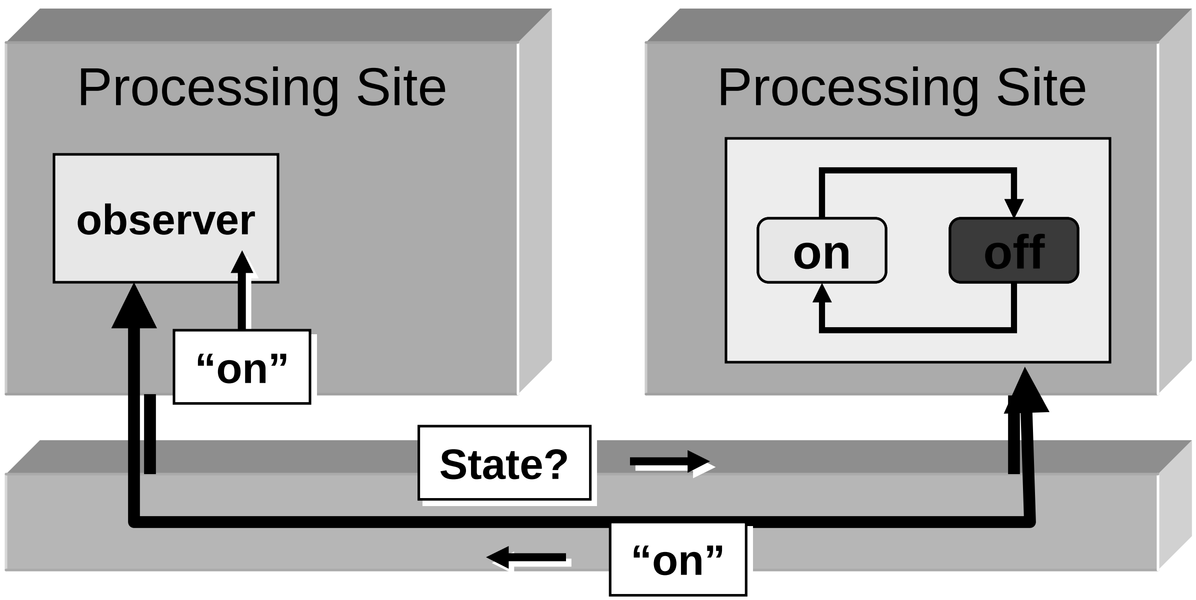
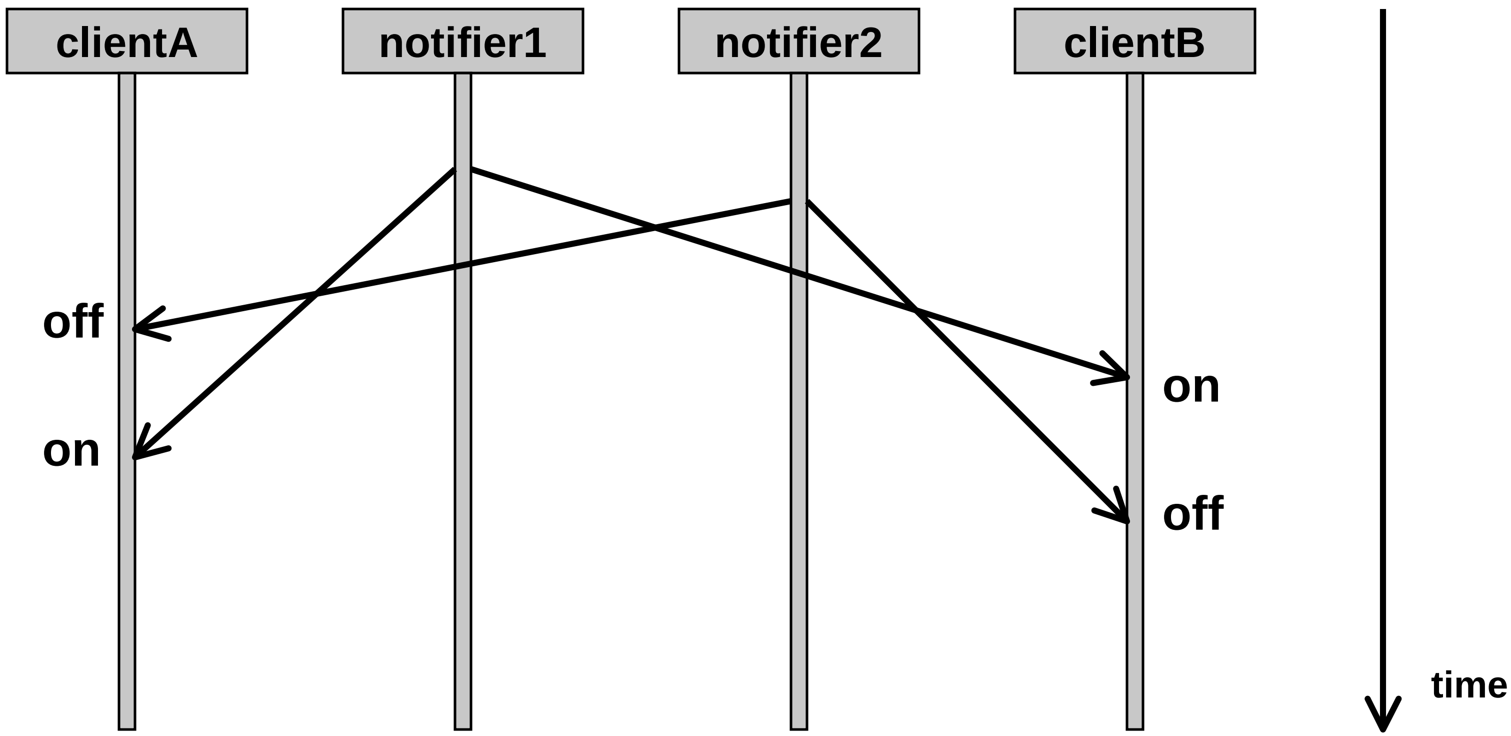


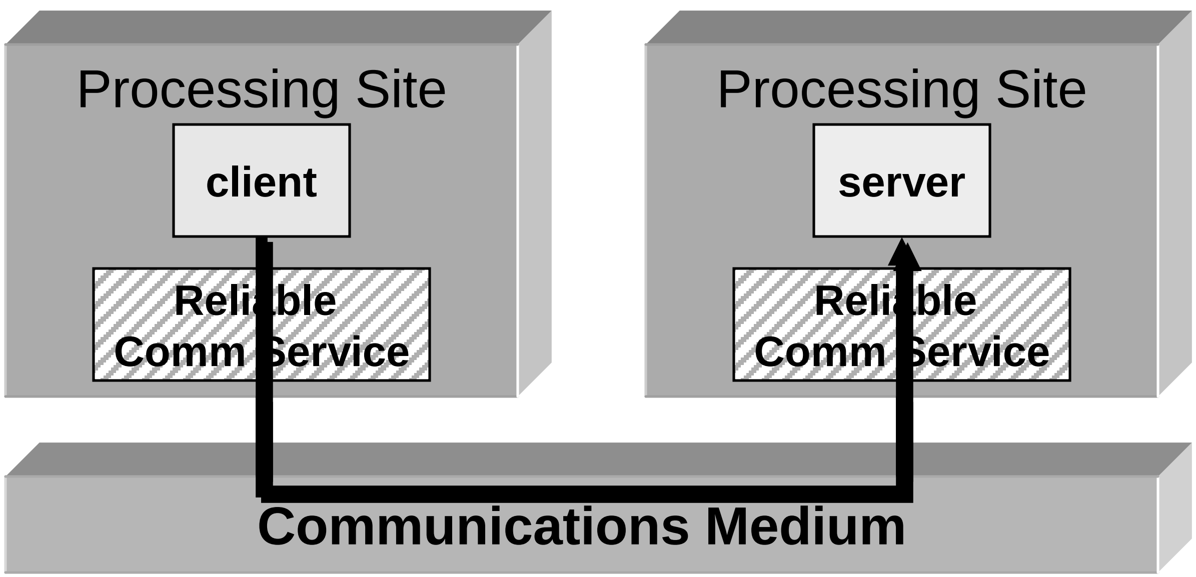
Exhibit B: Relativistic Effects

- Relativistic effects:
  - different observers see different event orderings (due to different and variable transmission delays)



Distribution Transparency Mechanisms

- Platform layers that mask out failures from the application
  - e.g., reliable RPC services, relocation transparency,…



Impossibility Result No.1

It is not possible to guarantee that agreement can be reached in finite time over an asynchronous communication medium, if the medium is lossy or one of the distributed sites can fail

- Fischer, M., N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process" *Journal of the ACM*, (32, 2) April 1985.

Impossibility Result No.2

Even when communication is fully reliable, it is not possible to guarantee common knowledge if communication delays are unbounded

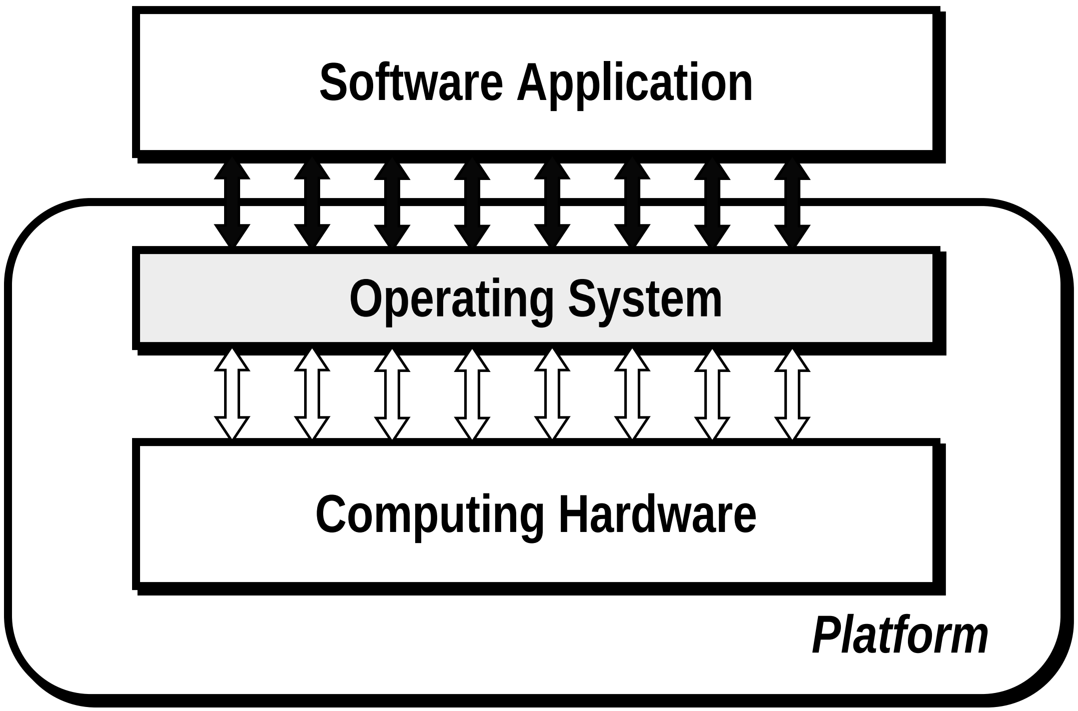
- Halpern, J.Y, and Moses, Y., “Knowledge and common knowledge in a distributed environment” *Journal of the ACM*, (37, 3) 1990.

Layering Does Not Always Help

- All forms of distribution transparency mechanisms require distributed agreement!
  - Transparency can only be approximated
  - The more transparency is desired the higher the cost (time, resources, complexity)
- *The end-to-end argument* [Saltzer et al.]:

  ⇒*the overhead of introducing transparency mechanisms may not always be justified by the benefits obtained*

What Software is Made of



- *Platform* = the complete technological base (SW and HW) required to execute an application
- The platform is the “construction material” of software, conveying its physical characteristics (speed, capacity, etc.) directly to the application

Platforms and Applications

- *What effect should a computing platform have on an application?*
- The answer: *as little as possible*

  …*but, no less!*
- *Platform-independent design (MDA?)*
  - Separation of concerns (simplifies design)

    yes…but separation of concerns is no excuse for negligence
  - Portability

    yes…but how much?
- A sound design principle that is far too often misinterpreted as "software that can run anywhere"

If Transparency is an Idealization…

- Facts to ponder:
  - In the Internet Age, most interesting applications will be distributed
  - As our dependence on computers increases, the physical characteristics of our software (response time, availability) will become much of a concern
- *Traditional Programming = Logic*
- *Physical Programming = Logic + Physics*
  - Like more traditional engineers, software designers must take into account the construction material out of which the logic is spun
  - Dealing with finite resources, finite delays, finite reliability...
- *"<u>All</u> machinery is derived from nature, and is founded on the teaching and instruction of the revolution of the firmament."*

Core Concepts for “Physical” Programming

Quality of Service

- The physical characteristics of software can be specified using the general notion of *Quality of Service (QoS):*

    a specification of how well a service can (or should) be performed

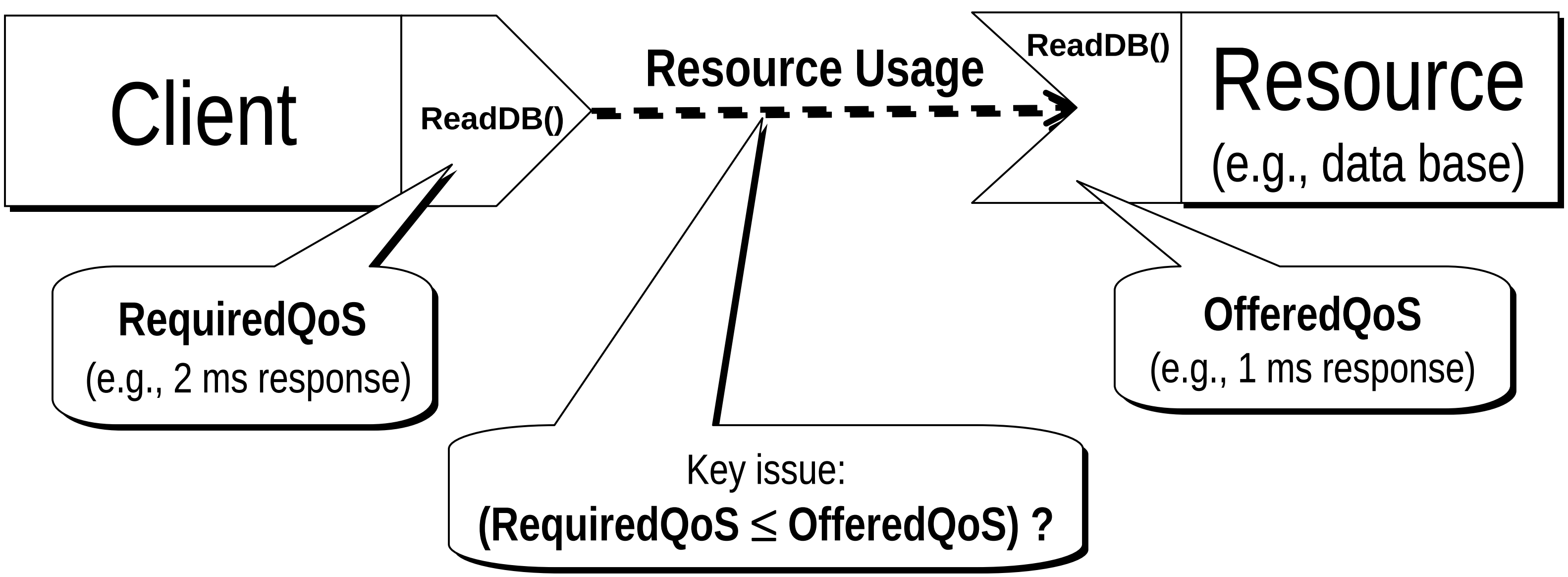
    - throughput, latency, capacity, response time, availability, security...
    - usually a quantitative measure
- QoS concerns have two sides:
    - *offered QoS:* the QoS that is available
    - *required QoS:* the QoS that is required to do a job

Resources and Resource Usage

- *Resource*:

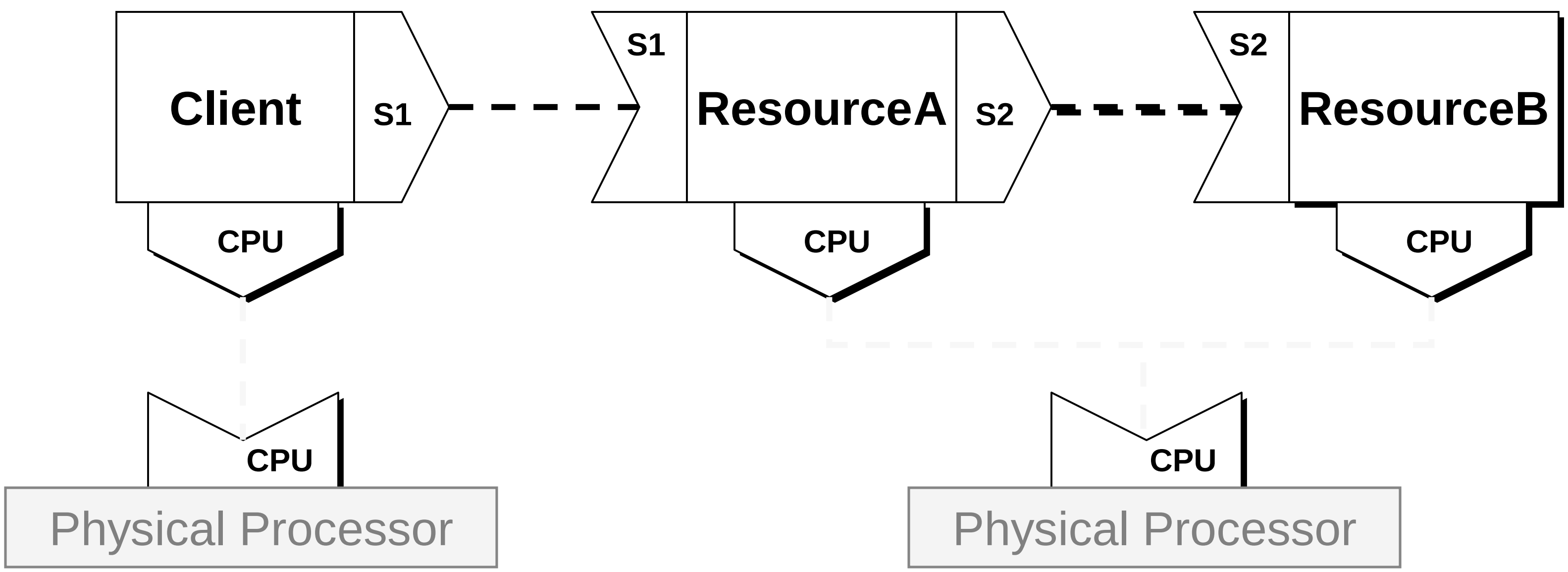
  an element whose ability or capacity is limited, directly or indirectly, by the finite capacities of the underlying physical elements

- The relationship between resources and resource users



Offered vs. Required QoS

- Like all guarantees, the offered QoS is *conditional* on the resource itself getting what it needs to do its job
- This extends in two dimensions:
  - the *peer* dimension
  - the *layering* dimension: for platform dependencies



“Physical” Types

- Types specify observable behavior
  - include QoS characteristics
- Required to answer the fundamental <u>engineering</u> question:
  - can a component (resource) support its required “load”?
- Declaration:

```
readDB (recNum : RecordId) : DBrecord
 {QoS:  responseTime = 0.75 * $CPUrate;}
  a kind of postcondition – implementation
  indepenent!
```

- Usage:

```
  curRec : DBrecord;
  recNo : RecordId;
  ...
  curRec := myDB.readDB(recNo)
   {QoS: responseTime ≤ 1};
```

Physical Type Checking

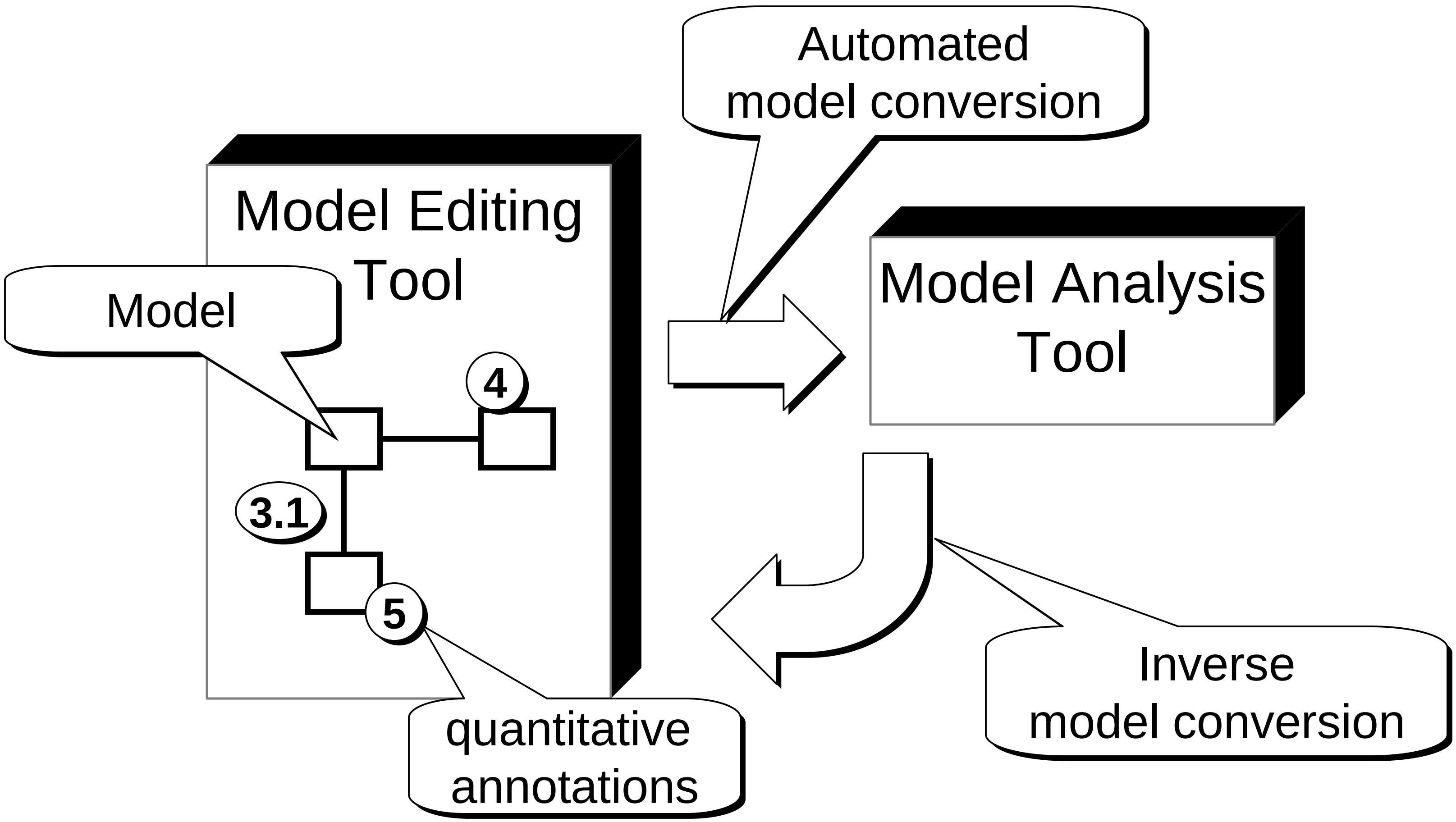
- Can physical types be statically checked by a compiler?
  - The good news: Yes (in most cases)
  - The bad news: typically requires complex analysis methods (queueing network analysis, schedulability analysis, etc.)

    …but then, model checking and theorem proving is not simple either
- Some issues:
  - In most cases QoS analysis cannot be done incrementally – the full system context is required

    …but then, the same holds for many formal verification methods
  - Each type of QoS (e.g., bandwidth, CPU performance) combines differently – no general theory for QoS analysis
- However, much of this can be automated
  - …just like model checking and theorem proving

Physical Type Checking Tools

- Method supported by the real-time UML standard



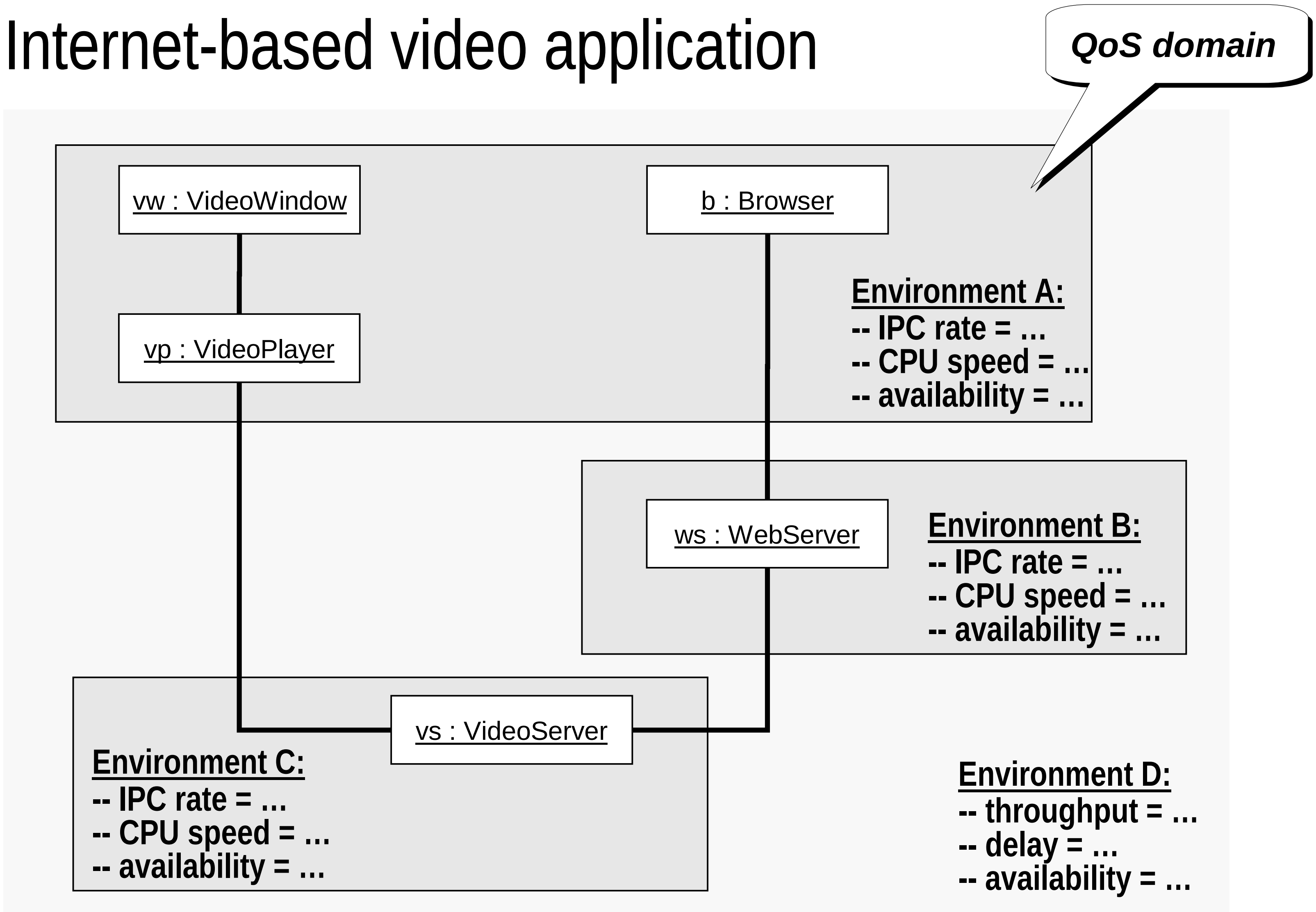
The True Path to Platform Independence

Achieving Platform Independence with QoS Concepts

- Dilemma: *How can we achieve platform independence if our application logic is a function of the physical QoS characteristics of the platform?*
- Solution: *Declare a technology-independent specification of the envelope of acceptable platform characteristics (required QoS) along with the application*
  - i.e., make platform assumptions explicit

Specifying Platform Characteristics

- An Internet-based video application



QoS Domains

- A domain in which certain QoS values apply uniformly:
  - CPU performance
  - communications characteristics (delay, throughput, capacity)
  - failure characteristics (e.g., availability, reliability)
  - etc.
- The QoS values of a domain can be compared against those offered by a concrete platform to see if that platform is adequate
  - …or, they can be used to synthesize the required domain

Summary

- The dependency of software on the physical aspects of its environment (platform) can be fundamental and must be clearly understood if we want to build correct software
  - Correctness extends beyond logical correctness to physical correctness
- We must adjust our software techniques, technologies, and methods to account for this
  - Avoid overly literal interpretations of general design principles such as separation of concerns
  - The use of models and model analysis are a step in this direction
  - Software models: can evolve directly into applications

“Physical” Programming: A Metrification of Logic

- The concepts of QoS, resource, and resource usage provide a foundation for addressing issues stemming from the physical underpinning of all software
  - the basis for formal verification
    {required QoS $\leq$ QoS of the platform}
- May also be used to automatically synthesize engineering environments that satisfy a given QoS specification of a logical model
- An initial attempt to capture this approach can be found in the real-time UML standard