Beyond Mere Logic: A Vision of Computer Languages for the 21st Century

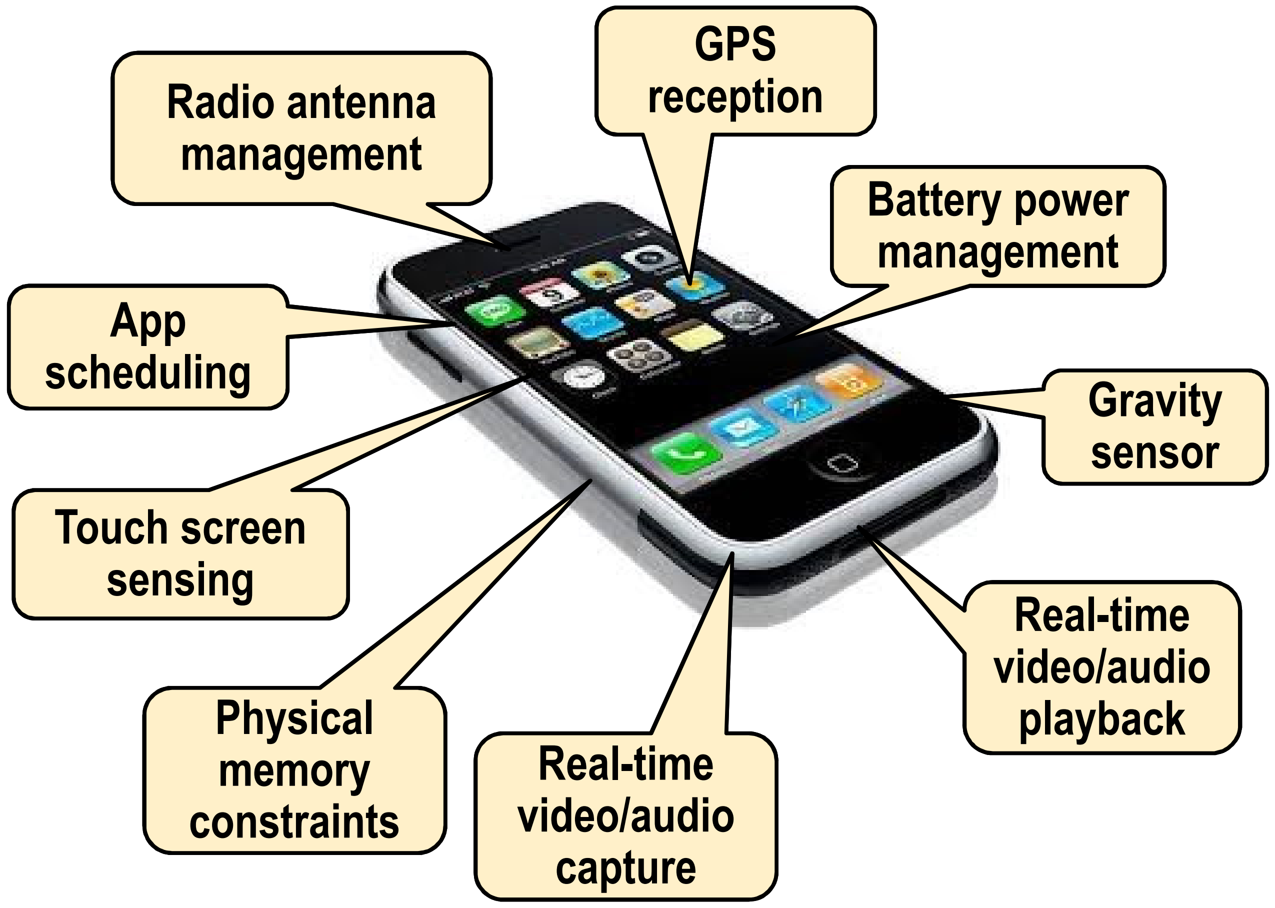
- A discourse on software physics -

Bran Selić

Malina Software Corp. CANADA
Simula Research Laboratory, NORWAY
Zeligsoft Limited (2009), CANADA
University of Toronto, CANADA
University of Sydney, AUSTRALIA

selic@acm.org

From Real Time to Real World



Real-time software has traditionally been perceived as a niche discipline, but...

From Real Time to Real World (cont.)

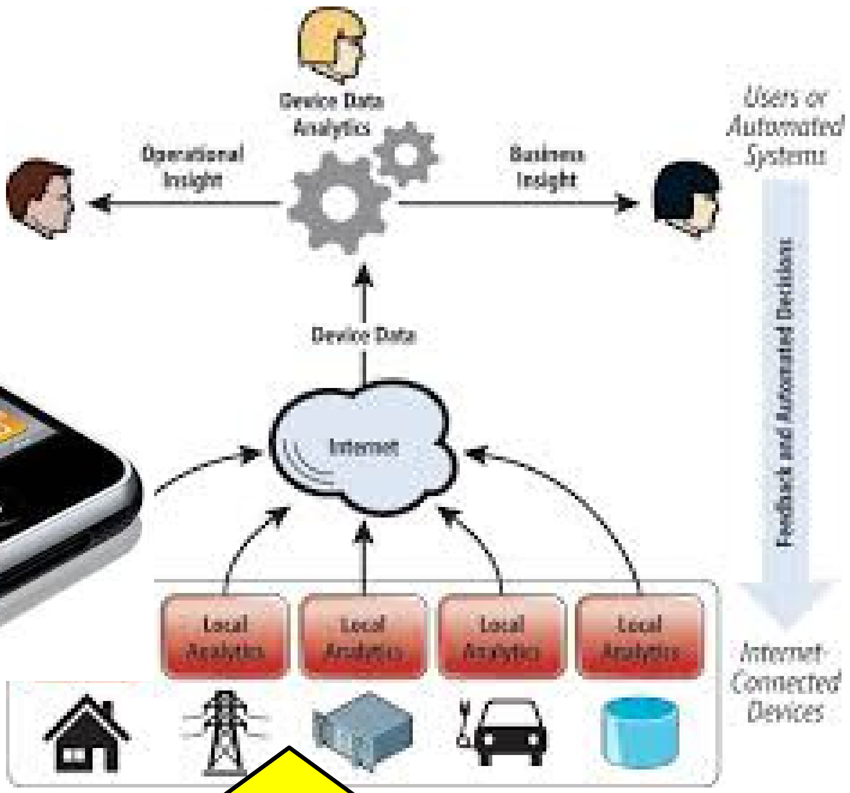
Smart Phones

Smart Grids

Smart Houses

Smart Almost Everything (SAE)

Internet of Things



⇒ *An increasing number of software applications interact directly with the* ***physical world***

Application Types in This Category

- Control and monitoring systems, communications systems, industrial control systems, automotive systems, etc.
- Financial systems (banking, point of sale terminals, etc.)
- Computer-aided design tools (AutoCAD, CATIA, etc.)
- Simulation software (physics, weather, machinery, etc.)
- Computer games software
- etc.

All of these application types either interact directly with the physical world and/or incorporate a representation of it

Q: Are our software technologies up to the task?

The Case of the MARS Climate Orbiter

The Mars Climate Orbiter

~$650M!

"The 'root cause' of the loss of the spacecraft was ***<u>the failed translation of English units into metric units</u>*** in a segment of ground-based, navigation-related mission software…"

-- NASA report, 1999

Q: Why was this not detected by the compiler as a type mismatch?

No mainstream programming language has a <u>first-class</u> concept of a "physical" value or time

e.g., `force:Force = 225;`
`delay(100);`

Sidebar: User Types vs. (First-class) Language Concepts

- Q: Can't we just define a special "physical" type?

```
enum LengthUnit {mm, cm, m, km};

type Length {
   real value,
   LengthUnit unit};
```

- No: a compiler would still not catch unit mismatches or know how to compare two or more values of such a type

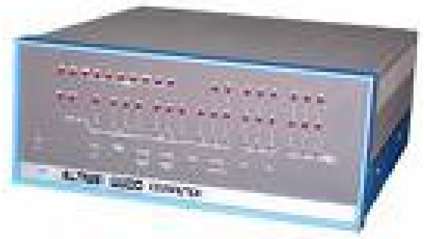
In contrast, a first-class language construct has semantics defined by the language that are known and supported by all conforming tools (compilers, validators, interpreters, debuggers, etc.)

The Case of the Vista™ OS

Q:Which of these Computing platforms can support Vista™?

Clearly, not much thought was given to the capabilities of the underlying hardware platform

Windows Vista


(a) MITS Altair 8800
(8080 CPU) 4KB

(b) Sinclair ZX81
(Z80 CPU) 8KB

(c) Lenovo ThinkPad X61
(Intel® Core™2 Duo CPU)
1GB

A:None of them

State of the Practice

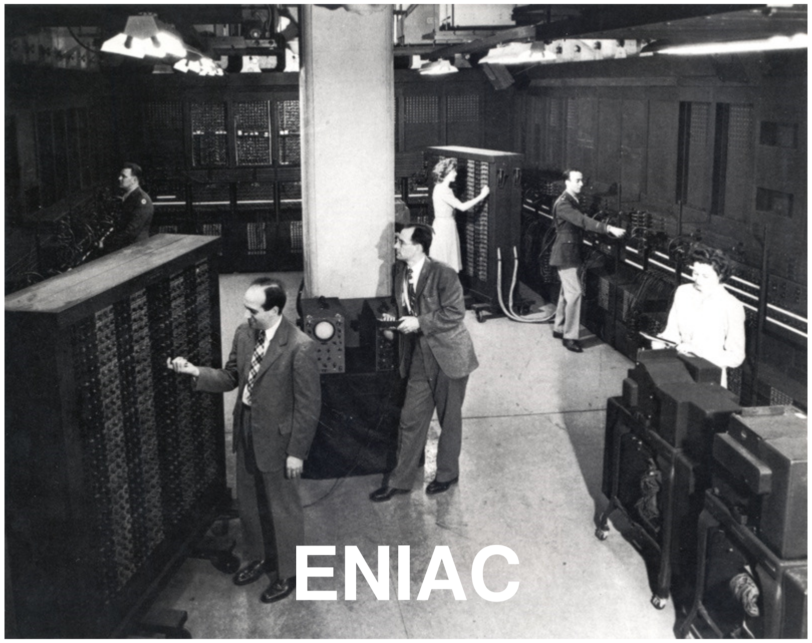
Our current software technologies and design methods are not very well suited for tackling interactive applications

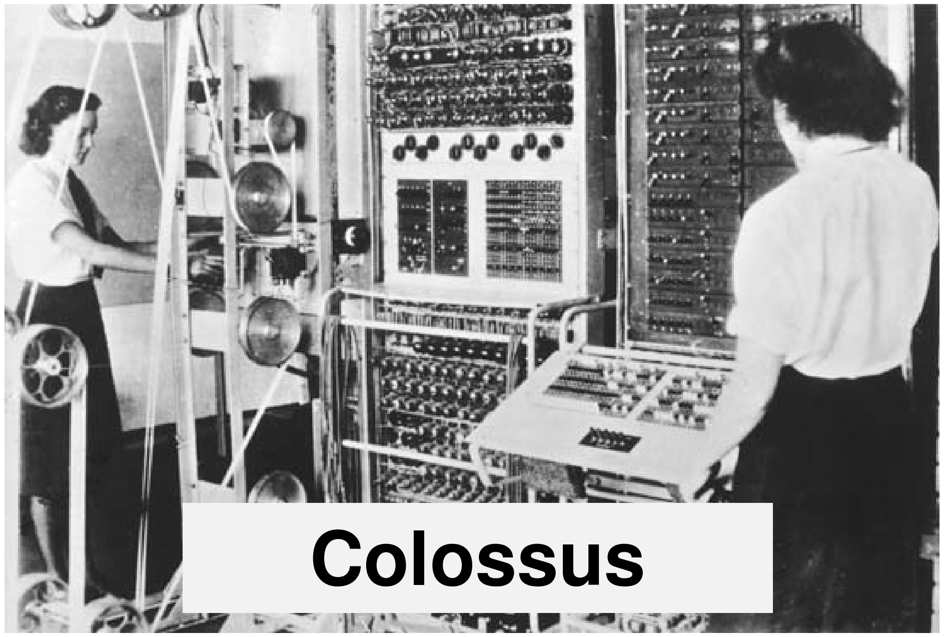
(A problem of <u>accidental</u> complexity)

Why not?

To understand why things are the way they are, we need to know how they came to be…

A Brief Look Back



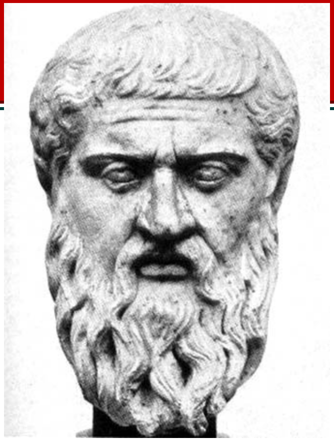


- **Original computer applications were devised to mechanize computation of complex algorithms**
  - Ballistics tables, code breaking, etc.
  - …which is why they are called "computers"

⇒ ***<u>Strong focus on numerical methods, mathematical logic, and symbol manipulation</u>***

A clear algorithmic bias

The Response: Software Platonism



- "I see _no meaningful difference between programming methodology and mathematical methodology._"

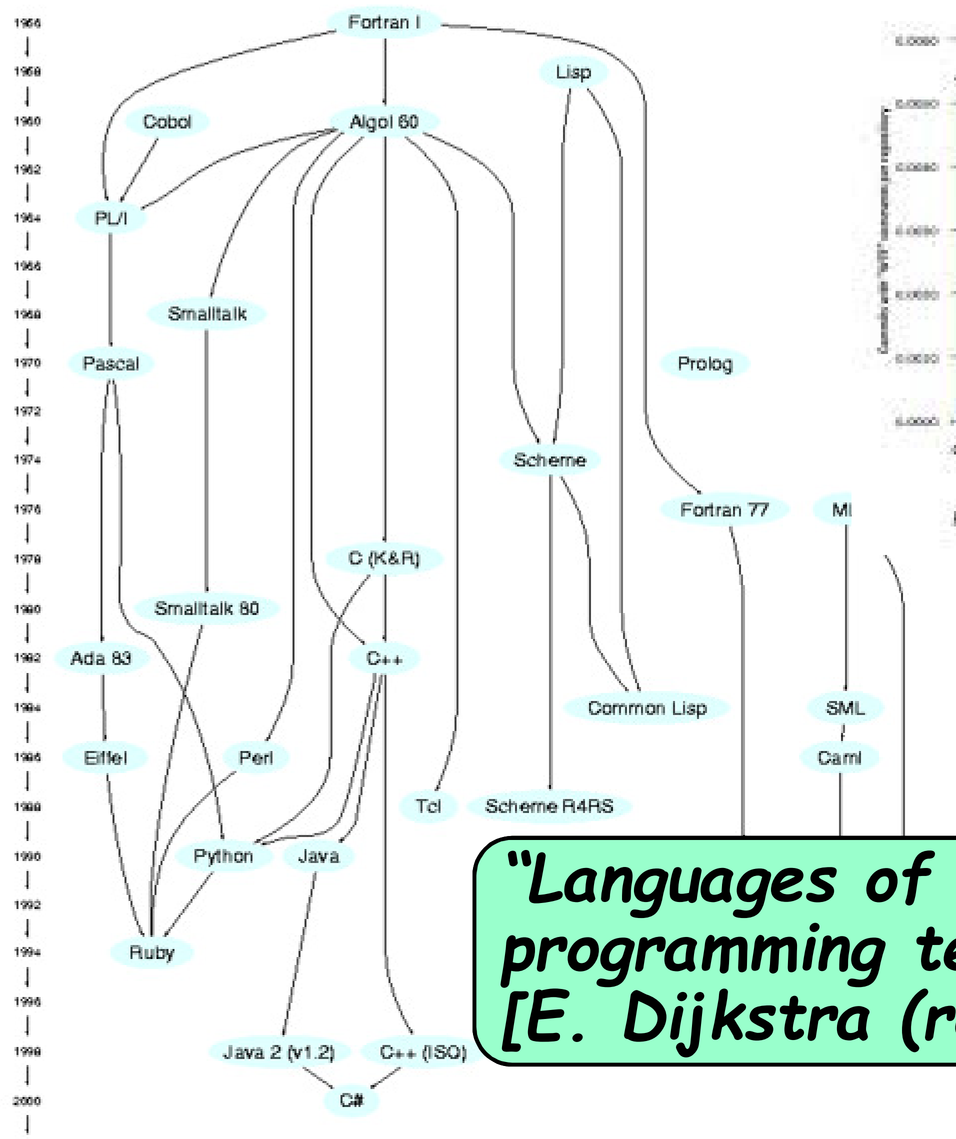
  -- Edsgar W. Dijkstra (EWD 1209)

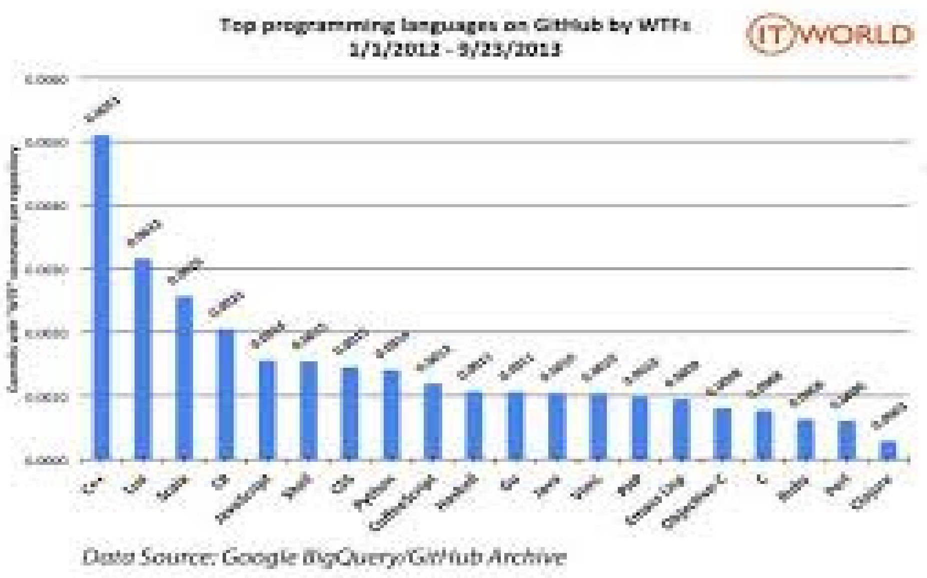
- "Because [programs] are put together in the context of a set of _information requirements_, *they observe no natural limits* other than those imposed by those requirements. Unlike the world of engineering, _there are no immutable laws to violate_."

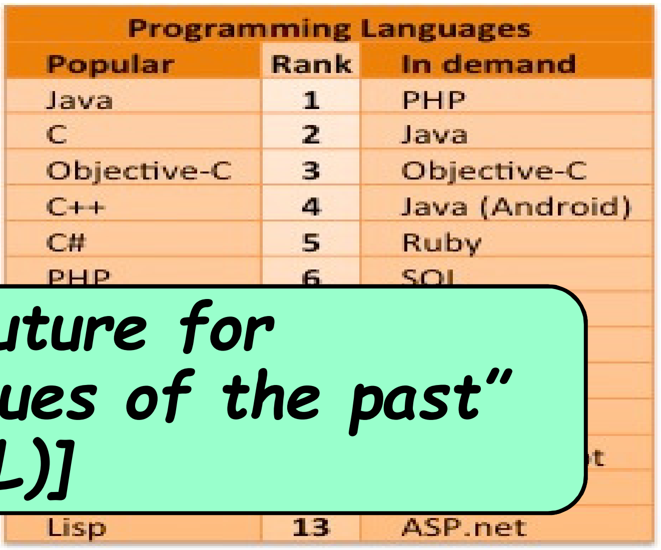
  -- Wei-Lung Wang, *Comm. of the ACM (45, 5), 2002*

This was and _still is_ a highly influential view

Current Mainstream Programming Languages







Programming Languages		
Popular	Rank	In demand
Java	1	PHP
C	2	Java
Objective-C	3	Objective-C
C++	4	Java (Android)
C#	5	Ruby
PHP	6	SQL
Lisp	13	ASP.net

"Languages of the future for programming techniques of the past" [E. Dijkstra (re: APL)]

http://www.pasteur.fr/formation/infobio/python/ch01s03.html

Source: Tiobe & Jobs

The Platonist Approach to Software Design

- Focus on system functionality ("business logic") first and foremost
  - No point in worrying about other concerns (e.g., performance, availability) if that is incorrect
- Donald Knuth:
  "Premature optimization is the root of all evil"
- "Platform independence"

Unstated assumption:
Other concerns are separable from functionality and, hence, can be retrofitted without disrupting the business logic (?)

Those “Other” Concerns

- The “ilities” of software
  - Reliability, scalability, availability, testability, performance/throughput, security, maintainability, stability, controllability, observability, extensibility, interoperability, usability, etc.

Most of these are affected either directly or indirectly by the physical aspects of the system (e.g., platform, communication networks)

Did someone just say "NON-FUNCTIONAL"!?

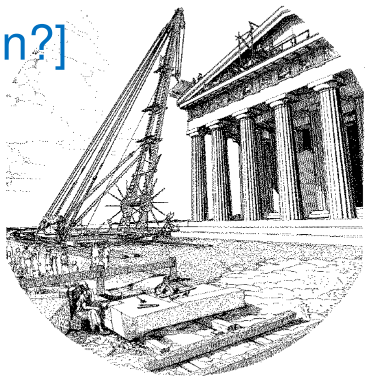
So, What's Wrong with Saying "Non-functional"?

1. Negative identification (does not tell us what they are)
2. Suggests second-order concerns (auxiliary, miscellaneous, etc.)
3. Bundles in an arbitrary way a collection of very diverse but often critical characteristics
   - Although each of them is achieved by different idiosyncratic means
4. Most critical: separates them from associated functionality
   - Many have a fundamental impact on how the functionality is realized
   - NB: They are mostly non-modular and pervasive ⇒ quality cannot be retrofitted easily (e.g., no such thing as a reliability or scalability module *or aspect*)

- Is "cross-cutting" a better term?
  - Not much: only deals with points 1 and 2 above
  - False impression that the problem can be solved with aspect-oriented solutions

The Wisdom of the Ancients*

* "The ancients stole all our good new ideas" [M. Twain/ R.W. Emerson?]



- "All 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

- Software + Computer (Hardware) = Special-purpose machine

  --Nancy Leveson, Safeware, 1995

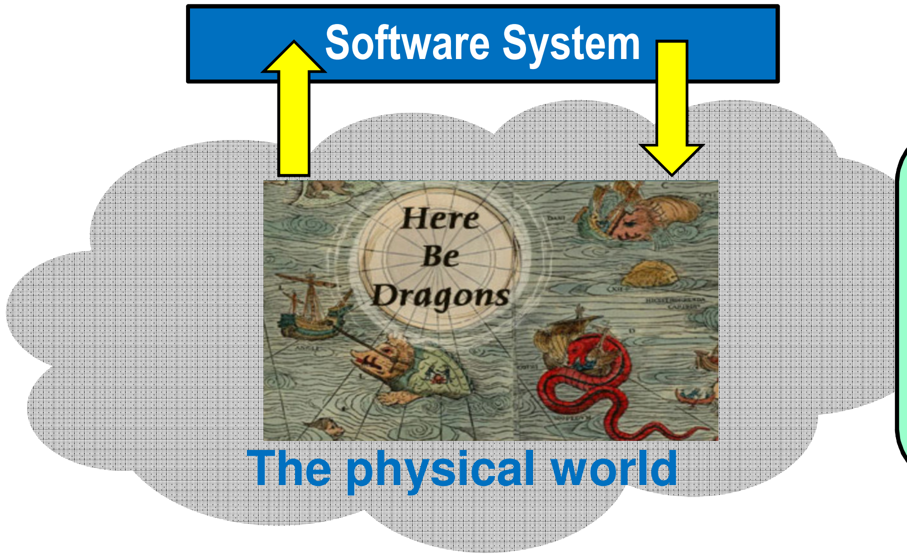
Q: What impact do the *physical* characteristics of this have on...

…this

…and this

Software Physics – and how to cope with it

What Makes Things Difficult for Software

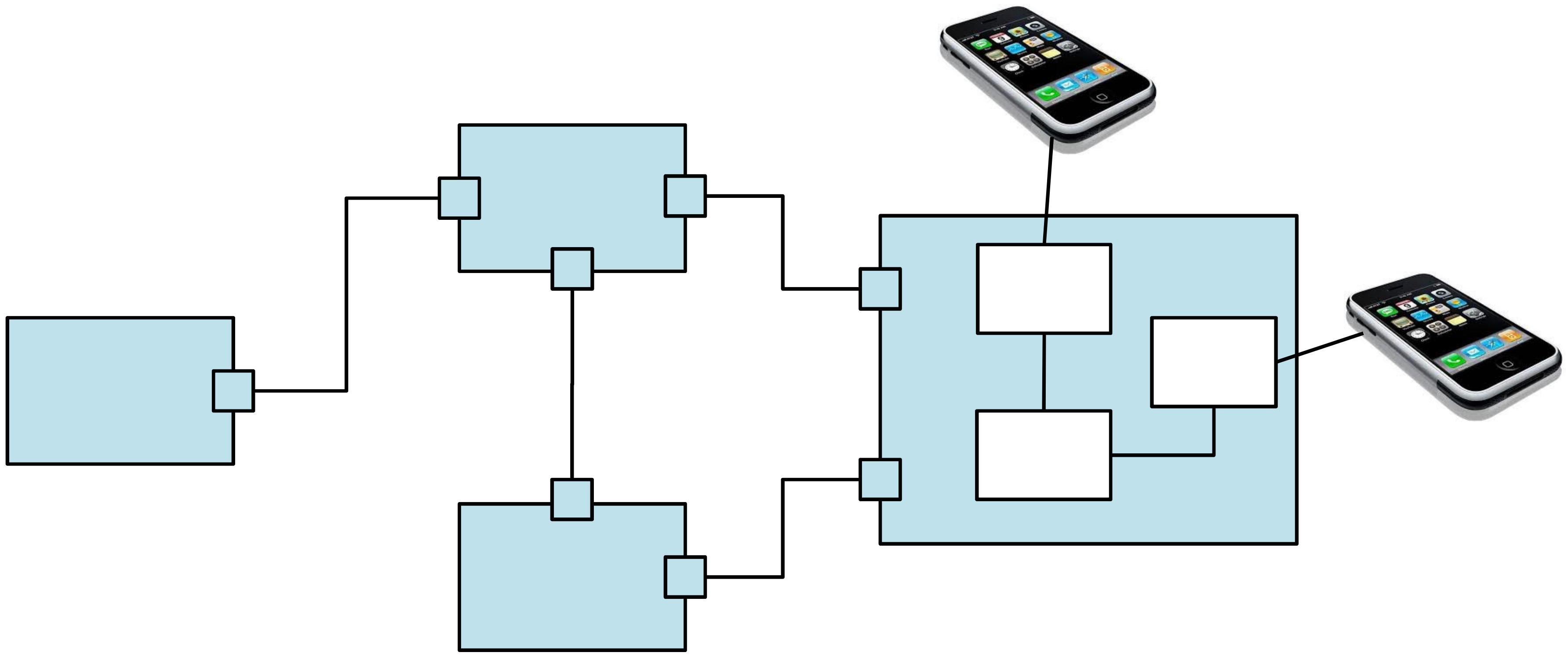


The physical world is complex and some of this complexity is necessarily transferred to the software

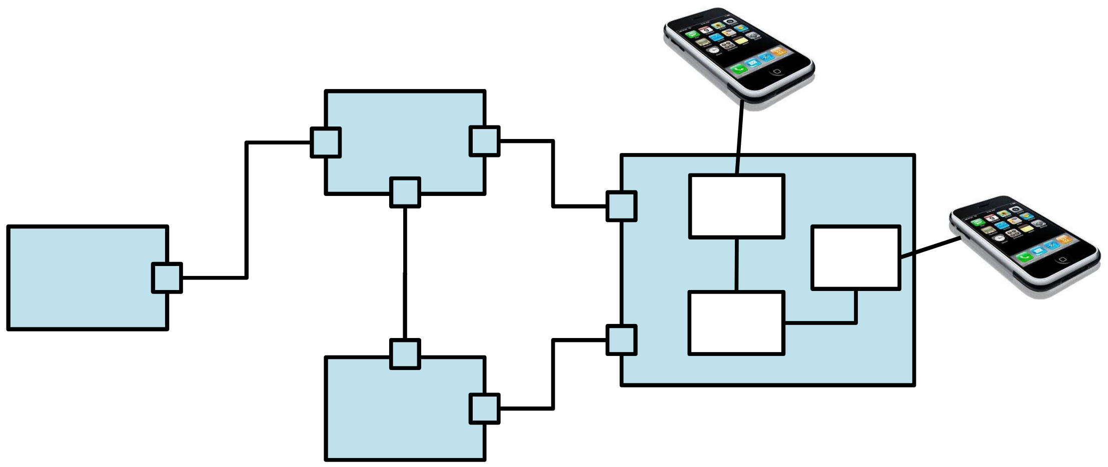
- The *essential complexities* of the physical world:
  - Physical distribution
  - Modal behaviour
  - Non-determinism (asynchrony)
  - Concurrency
  - Qualitative diversity
  - Quantity can affect quality

The Effects of Physical Distribution (1)

- **Structural impact:**
  - Need to specify complex *topological* structures
  - Need for local software "agents" that represent and interact with that world to the rest of the software

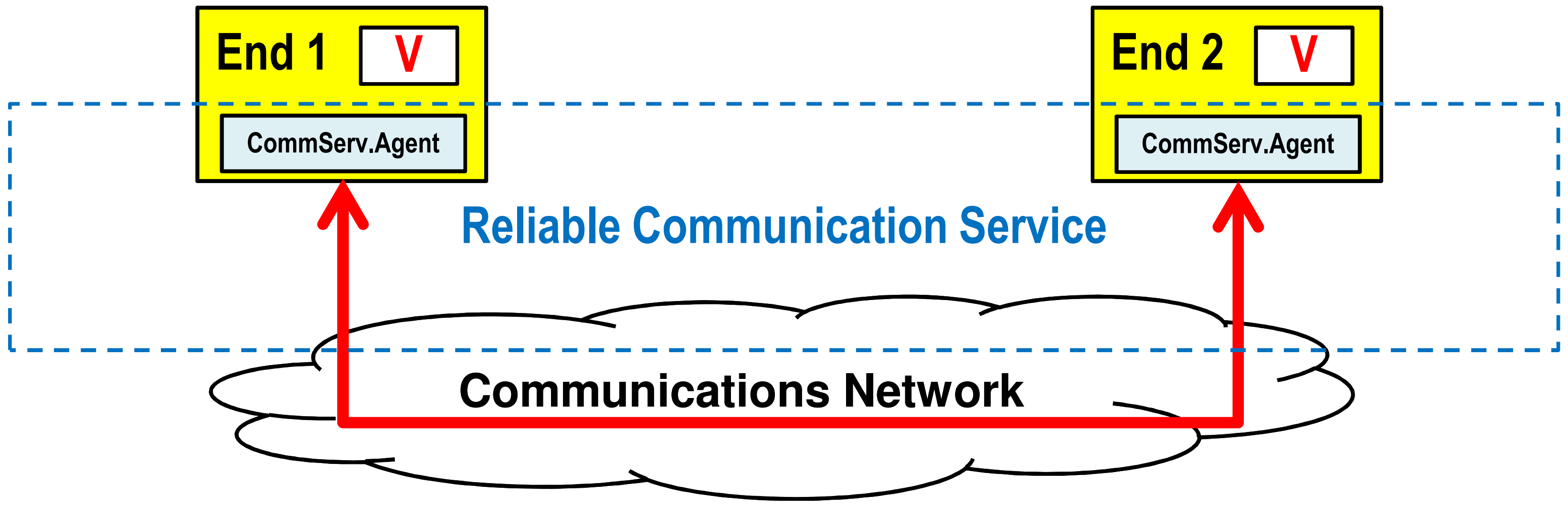


Coping with Structural Impacts of Distribution



- Introduction of the OO paradigm has proved fundamental here
  - A structural approach: programs represented by networks of collaborating machines
  - Introduction of logical entities (e.g., a "call" object)
- Enhanced by the introduction of architectural description languages (ADLs)
  - E.g., UML structured classifiers, collaborations, AADL

Physics vs. Logic: The Great Impossibility Result



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.

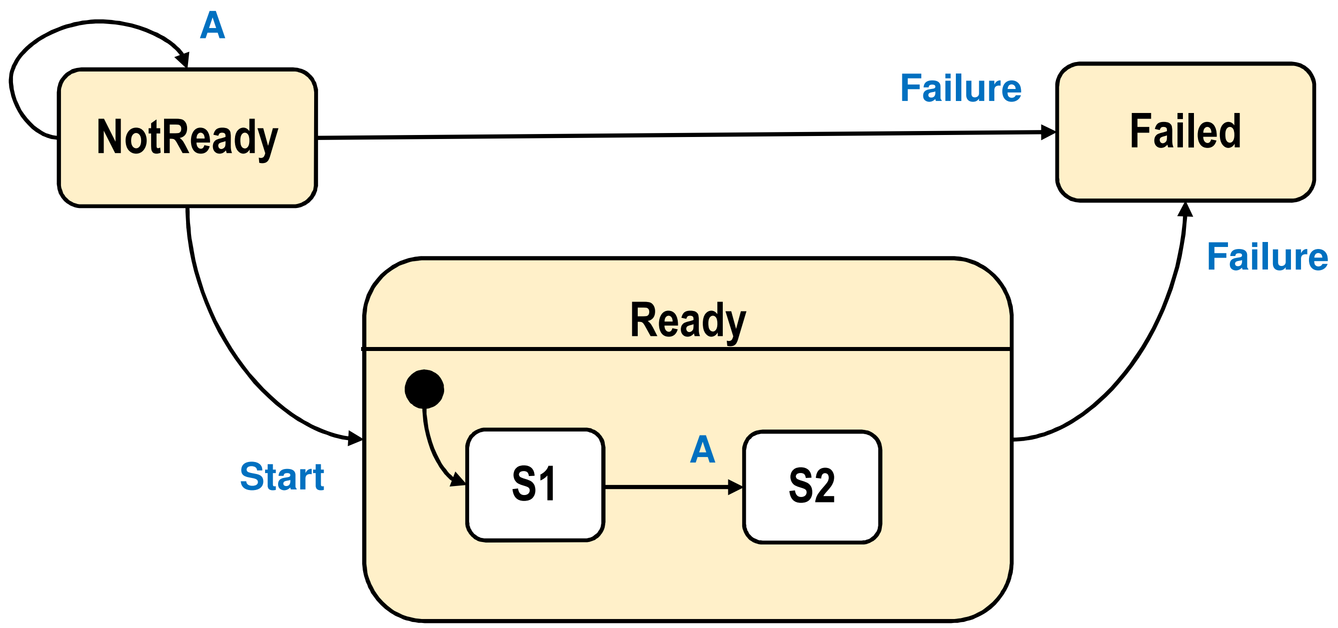
[Fischer, M., N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process" *Journal of the ACM*, (32, 2) April 1985]

The Effects of Physical Distribution (2)

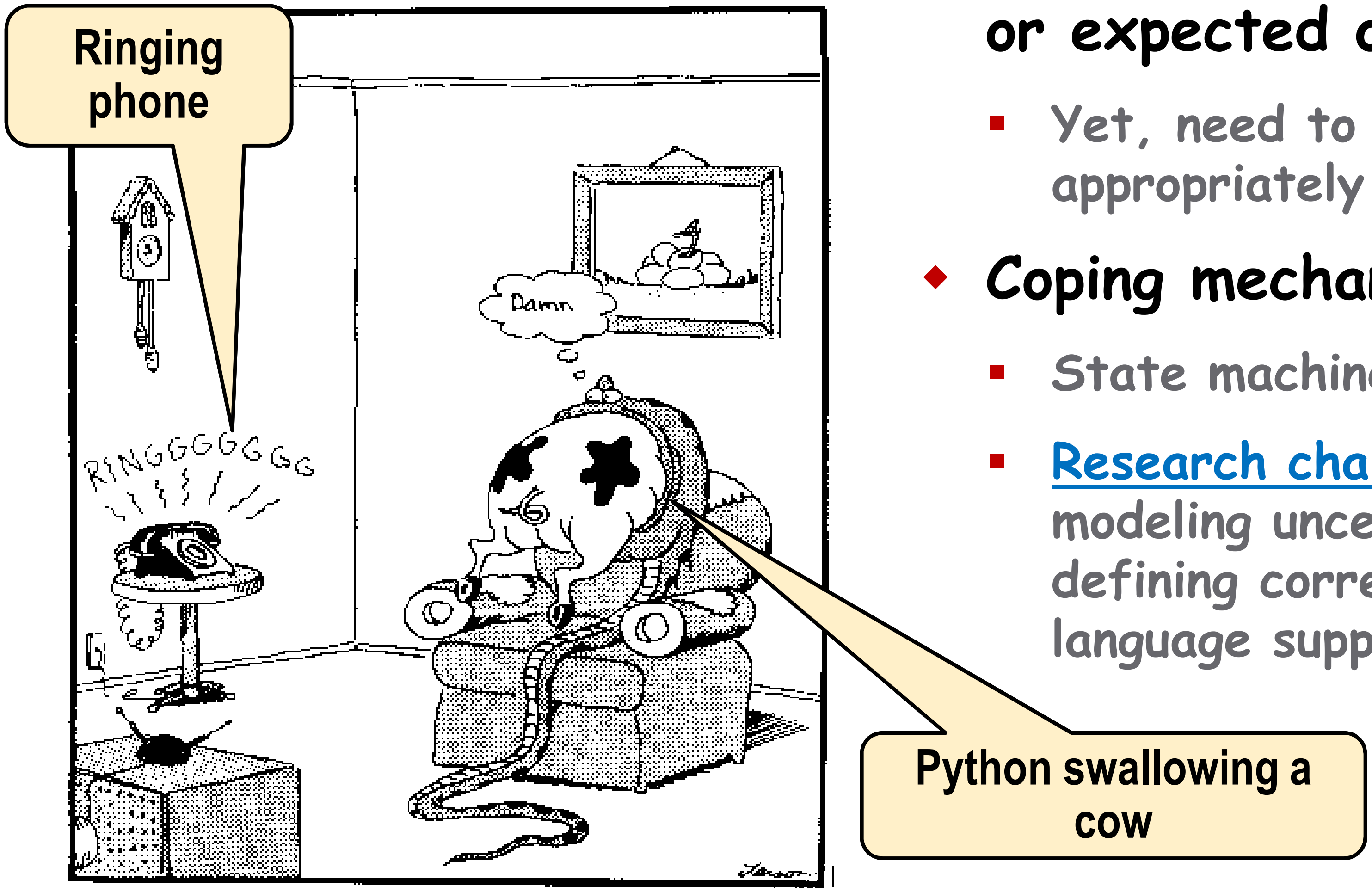
- **Behavioral impact:**
  - Communication delays (outdated status data) and failures (e.g., loss, duplication, reordering of messages)
  - Partial system (i.e., node) failures
- **Coping mechanisms:**
  - Fault-tolerance strategies (e.g., protective redundancies, fault diagnosis, fault recovery) have been defined
  - Need an ontological framework of failures and corresponding remedies
  - *First-class language support needed for these types of mechanisms*
    - Research challenge: can and how should a computer (modeling) language support these?

Modal Behaviour

- Response to an event depends on what happened before (history)
- Coping mechanism: state machines
  - In particular hierarchical state machines for specifying modal behaviors (e.g., UML state machines)



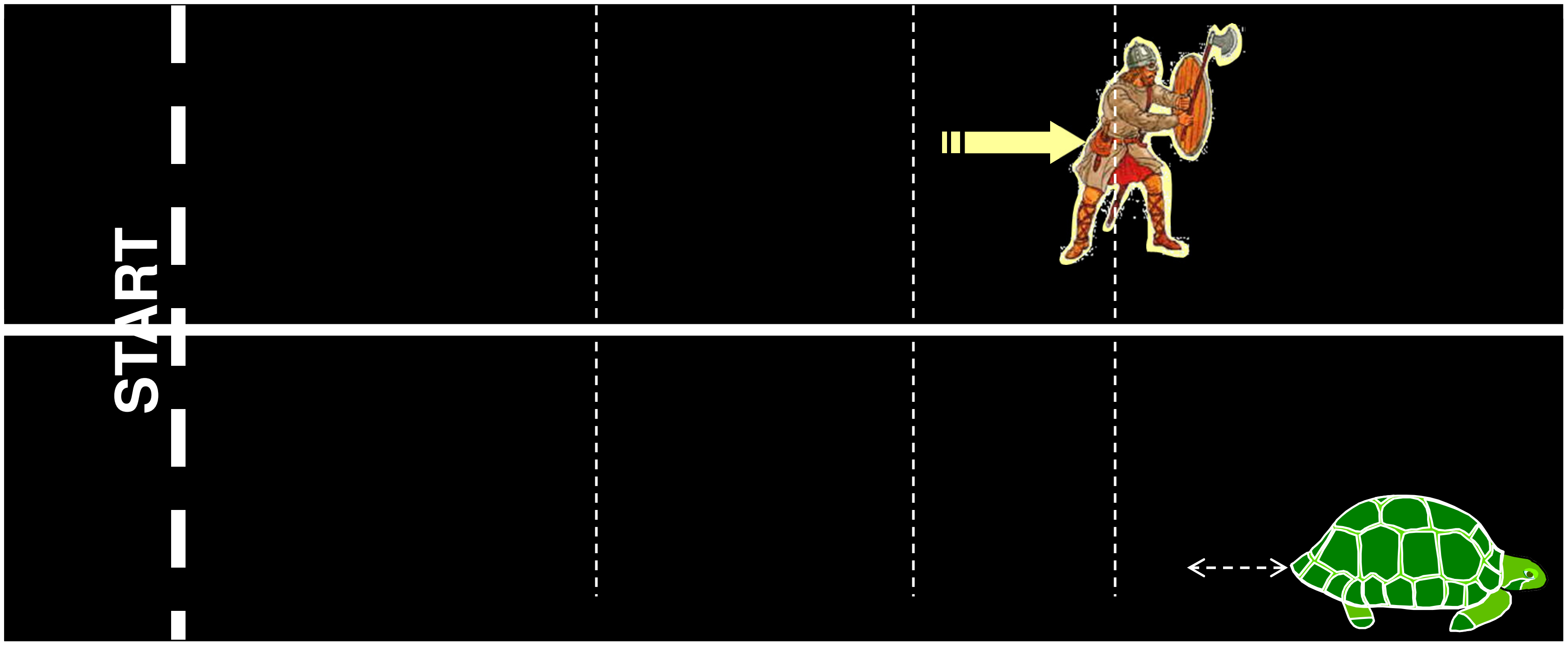
Non-Determinism (Asynchrony)



- Events can and do occur out of desired or expected order
  - Yet, need to be handled appropriately
- Coping mechanisms:
  - State machines
  - Research challenge: modeling uncertainty and defining corresponding language support

Concurrency

- Difficult to reason about concurrency



Coping with Concurrency

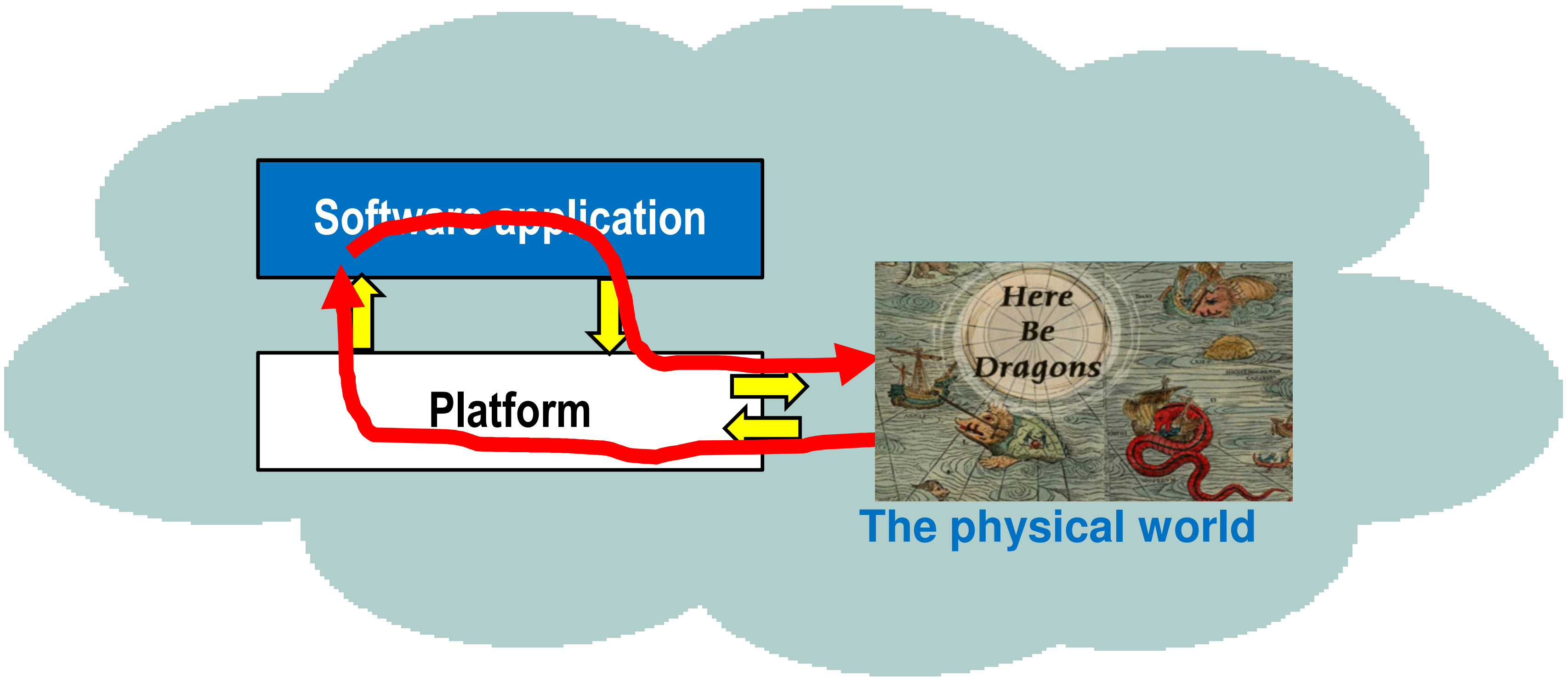
- Direct language support for existing concurrency management and synchronization mechanisms
  - Active objects (e.g., UML): programs as networks of concurrent entities
  - Synchronization mechanisms (run-to-completion, priority scheduling mechanisms, mutual exclusion mechanisms, etc.)
- The MARTE profile as an example

Beyond Logic: MARTE

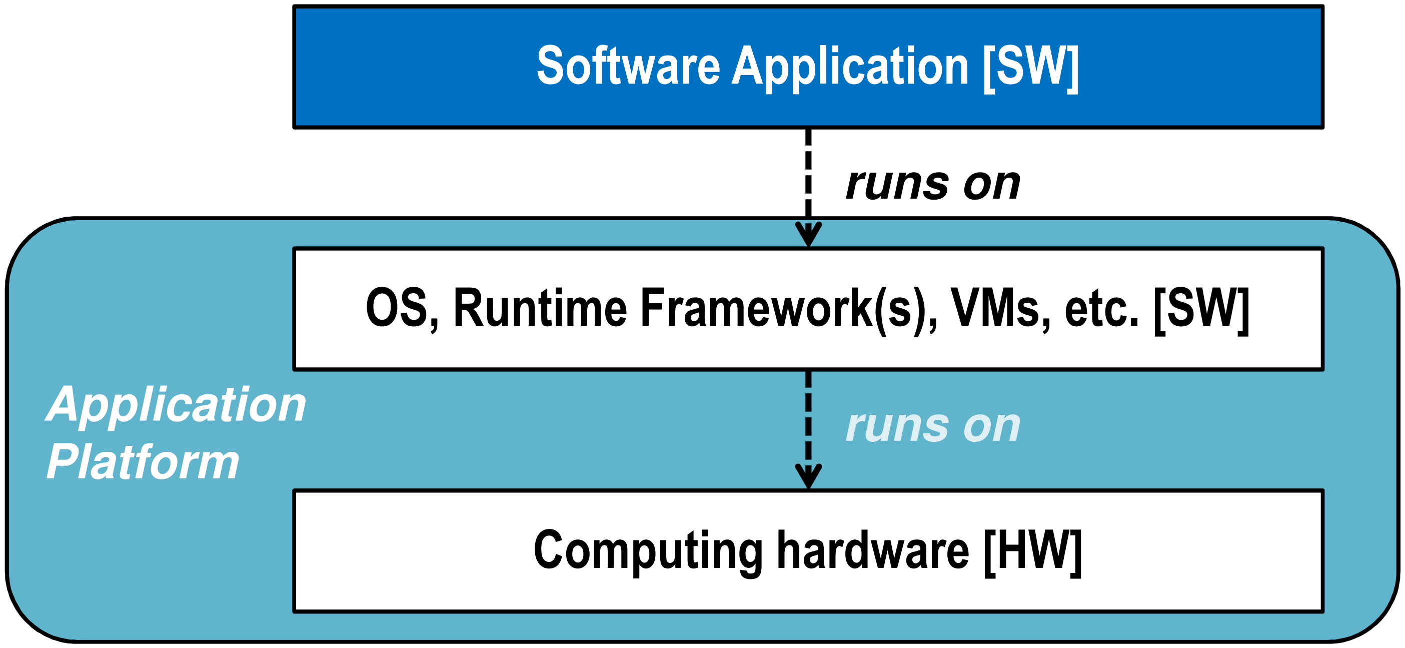
coping with quality and quantity in software

Where Software Meets Physics

- Everything that the software senses and performs is mediated by the platform and *is influenced by its physical properties*



Platforms: The Raw Material of Software



Software Application [SW]
runs on
OS, Runtime Framework(s), VMs, etc. [SW]
Application Platform
runs on
Computing hardware [HW]

- ***[Software] Platform**: The full complement of software and hardware required for a given application program to execute correctly*

Mainstream programming and modeling languages lack support for representing platforms and their characteristics!

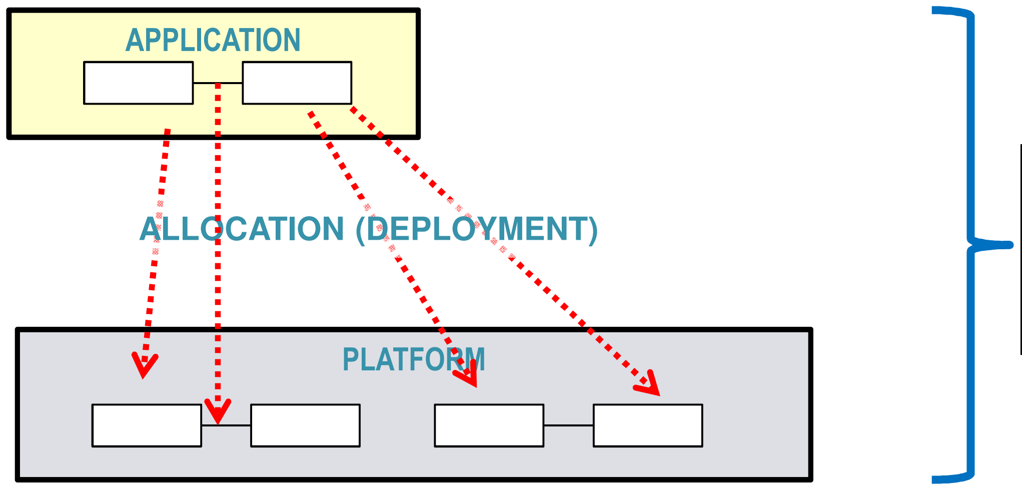
What About Platform Independence?

- An important and useful notion
  - Helps abstract away irrelevant technological detail
  - Necessary for software portability
- *Platform independence does not mean platform ignorance*
  - There are ways of achieving platform independence that account for the influence of platform characteristics

Any claims of "platform independence" should be accompanied by clear statements of the range of platforms that the application is independent of

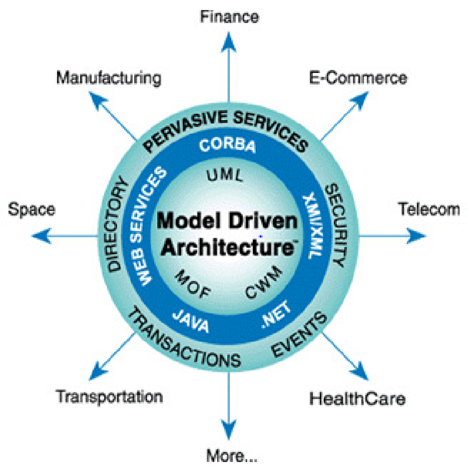
What We Need to Know About Platforms

1. Its relevant quality of service characteristics (size, capacity, performance, bandwidth, etc.)
2. Its computing and communications structure
3. The deployment of application software components across the platform



What is MARTE?

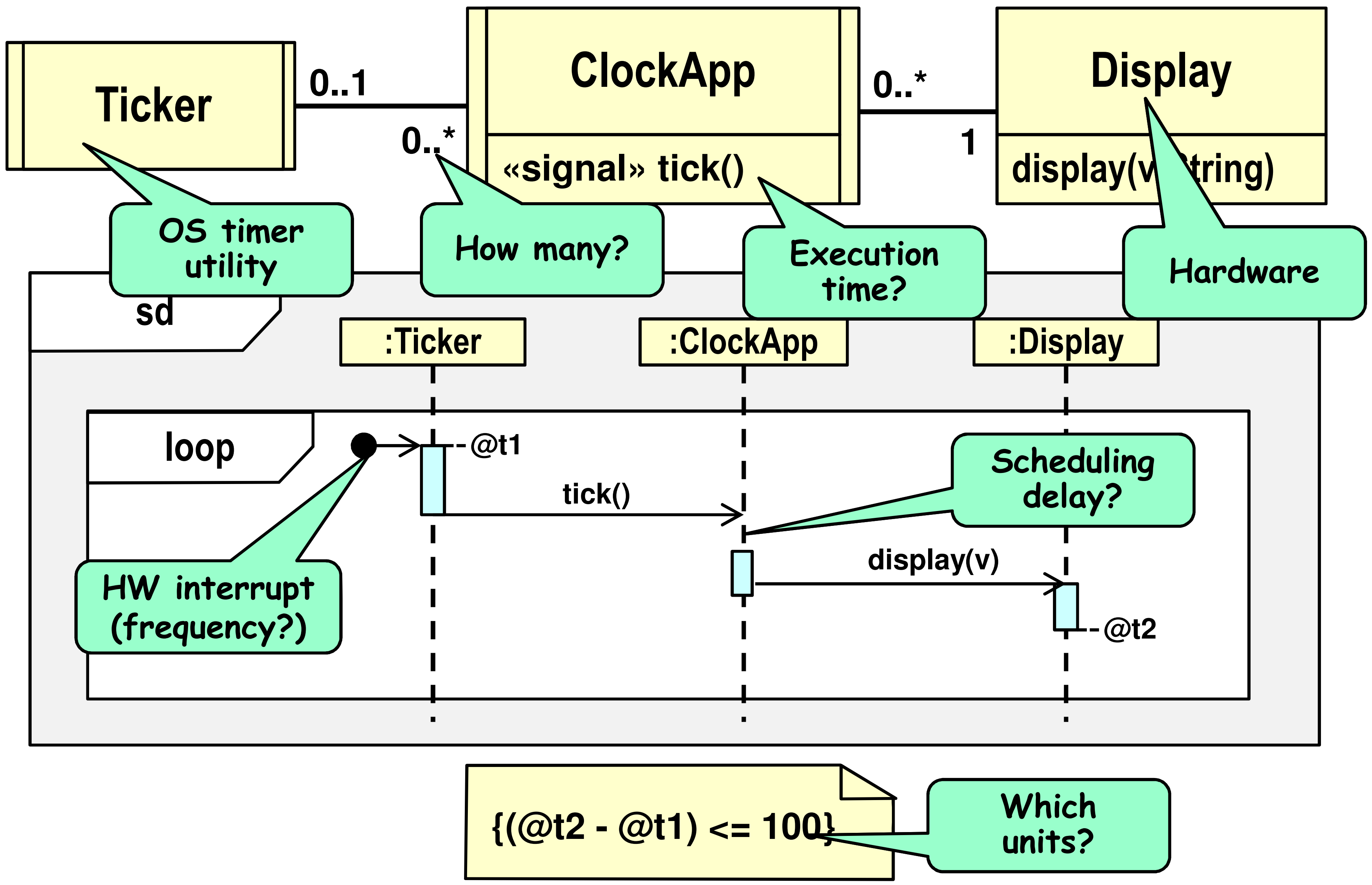
- A *domain-specific modeling language* (DSML) for the design and analysis of modern cyber-physical systems
  - Modeling and Analysis of Real-Time and Embedded systems
  - Supplements UML (i.e., does not replace it)
  - Realized as a UML profile



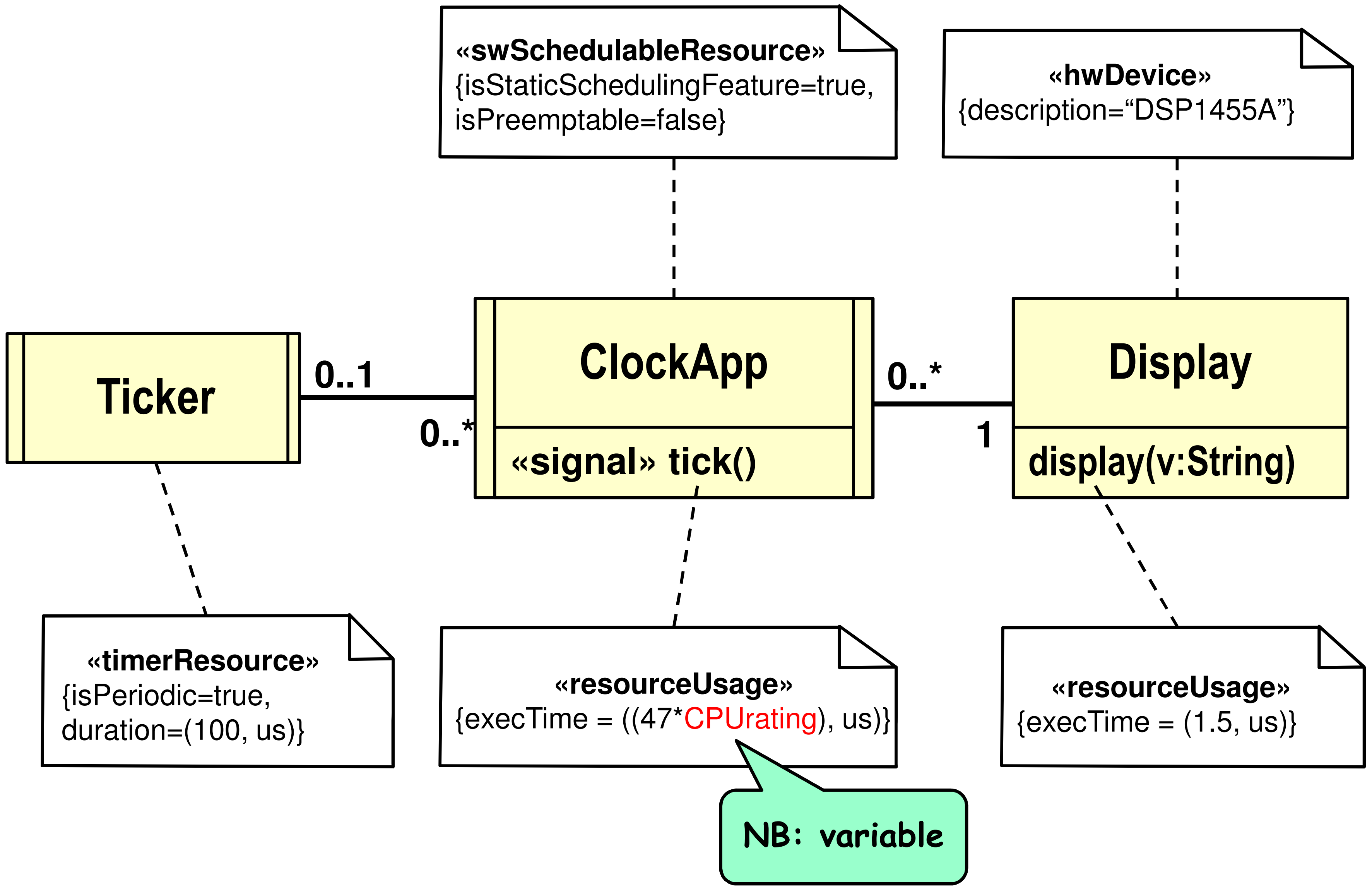
What MARTE Adds to UML

1. SUPPORT FOR *CONCISE AND SEMANTICALLY MEANINGFUL MODELING OF CPS SYSTEMS*:
   - A domain-specific modeling language for modeling real-time, embedded, and cyber-physical systems
   - Support for precise specifications of quality of service (QoS) characteristics (e.g., delays, memory capacities, CPU speeds, energy consumption)
   - Can be used directly in conjunction with SysML for greater CPS support
2. SUPPORT FOR *FORMAL ENGINEERING ANALYSES OF MODELS OF RTE/CPS*:
   - A generic framework for certain types of (automatable) quantitative analyses of UML models
   - Suited to computer-based automation

Example: "Bare" UML Model



Annotating a UML Model with MARTE

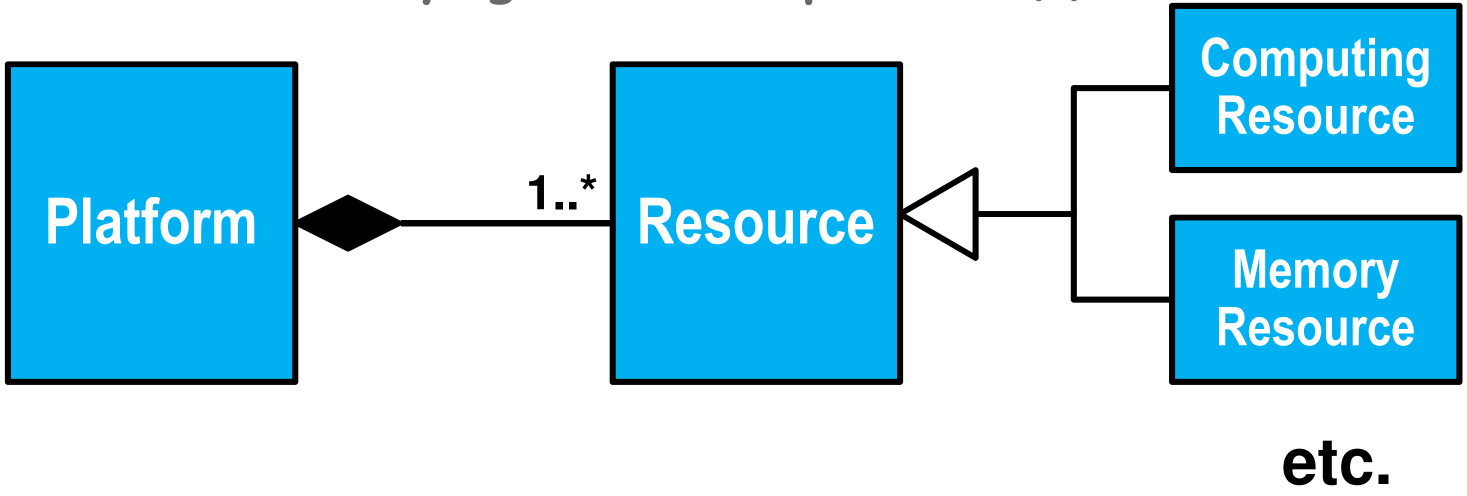


Core Concept: Resource

- ***Resource***: [Oxford Dictionary definition]

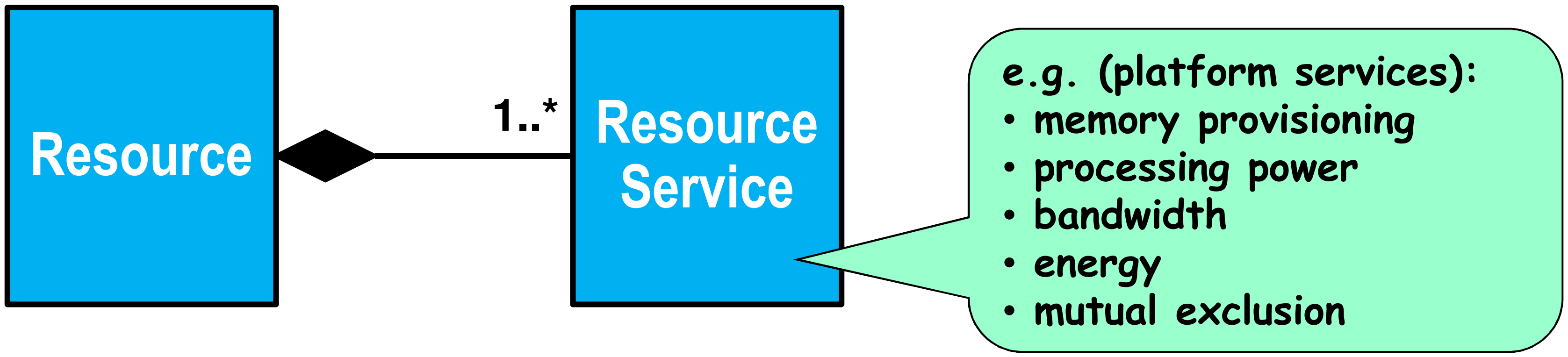
  "A source of supply of money, materials, staff and other assets that can be drawn upon…in order to function effectively"

- **In MARTE, a platform is viewed as a *collection of different types of resources,* which can be drawn upon by applications**
  - The *finite nature of resources* reflects the physical nature of the underlying hardware platform(s)



Core Concept: Resource Services

- In MARTE resources are viewed as <u>*service providers*</u>
  - Consequently, applications are viewed as <u>*service clients*</u>



- Resource services are characterized by their
  - Functionality
  - Quality of service (QoS)

Core Concept: Quality of Service (QoS)

- ***Quality of Service (QoS):***
  - A measure of the effectiveness of service provisioning
- **Two complementary perspectives on QoS**
  - Required QoS: the demand side (what applications require)
  - Offered QoS: the supply side (what platforms provide)

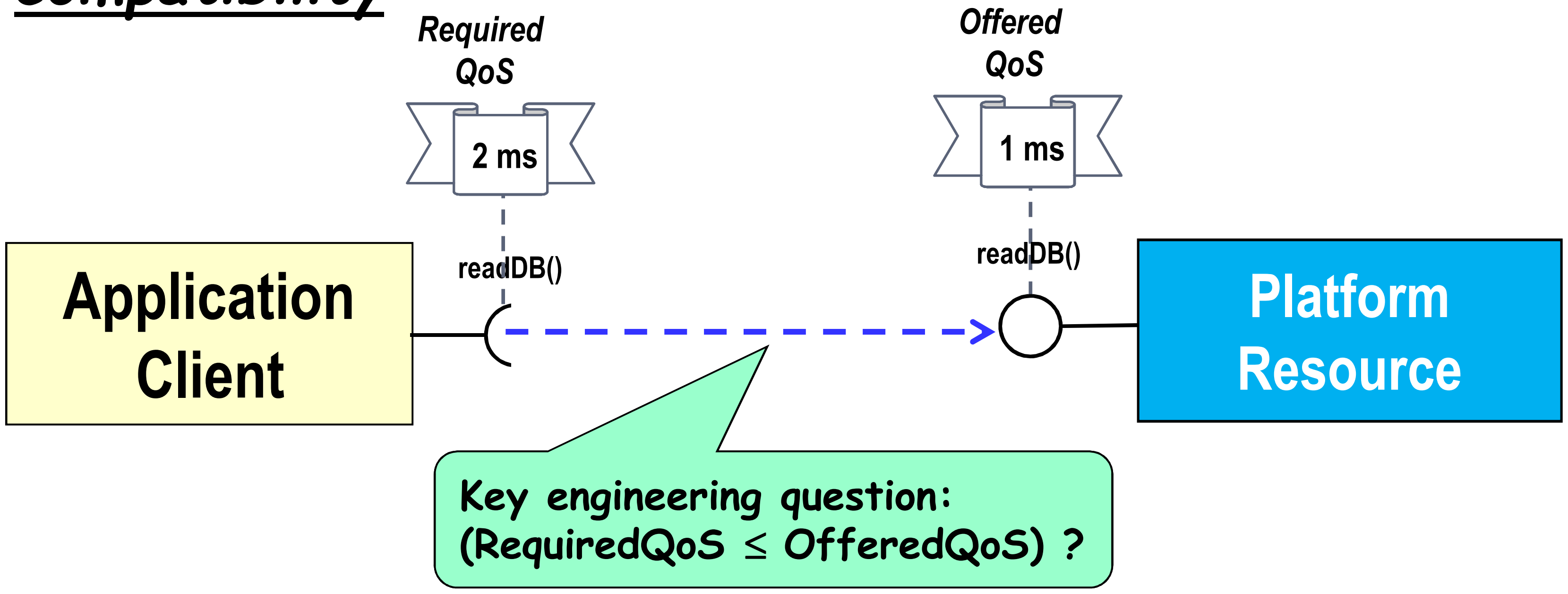
Many engineering analyses consist of calculating whether (QoS) supply can meet (QoS) demand

"Virtually every calculation an engineer performs…is a failure calculation…to provide the limits than cannot be exceeded"

-- Henry Petroski

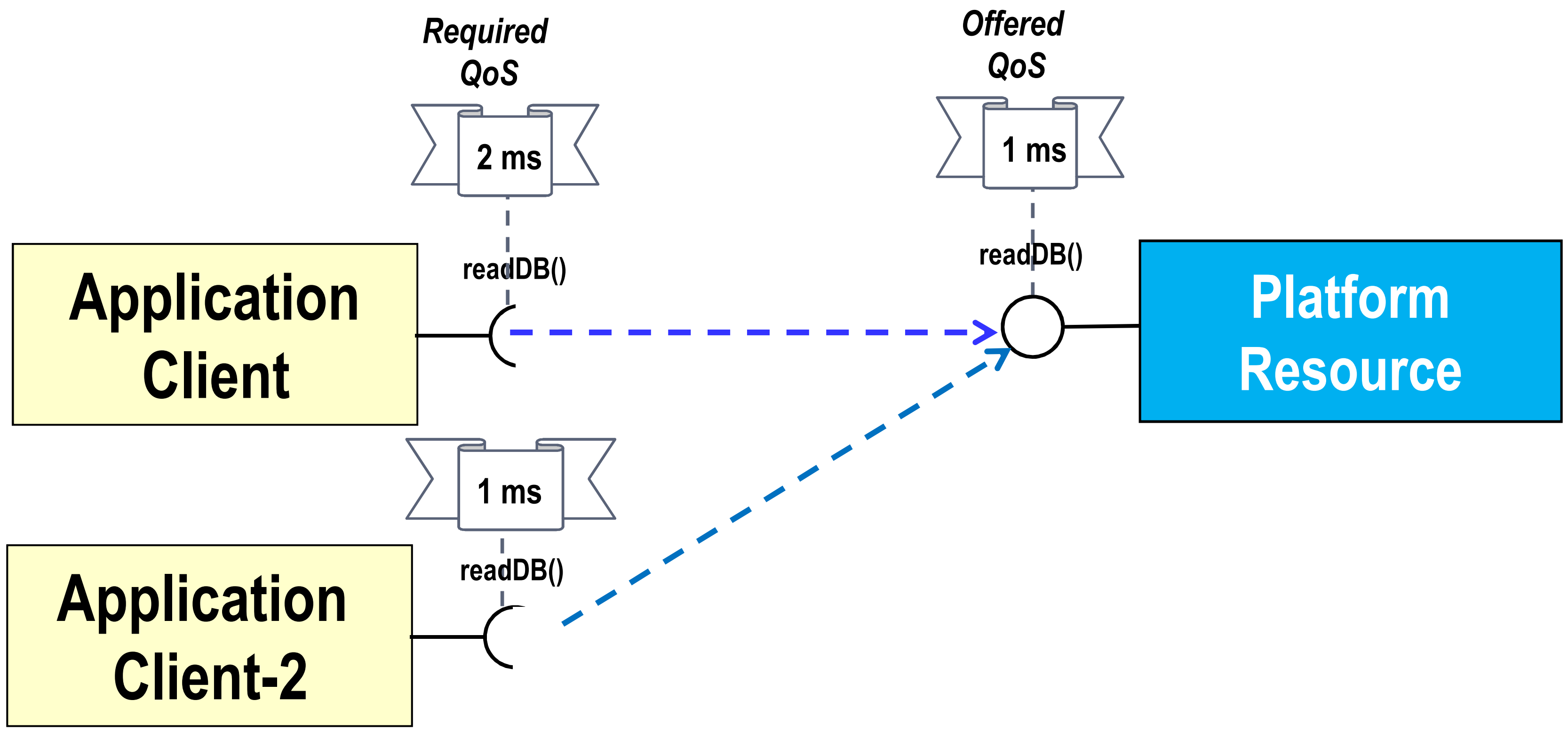
QoS Compatibility

- We have powerful mechanisms for verifying functional compatibility (e.g., type theory) but relatively *little support for verifying QoS compatibility*



Why It is Difficult to Predict Software Properties

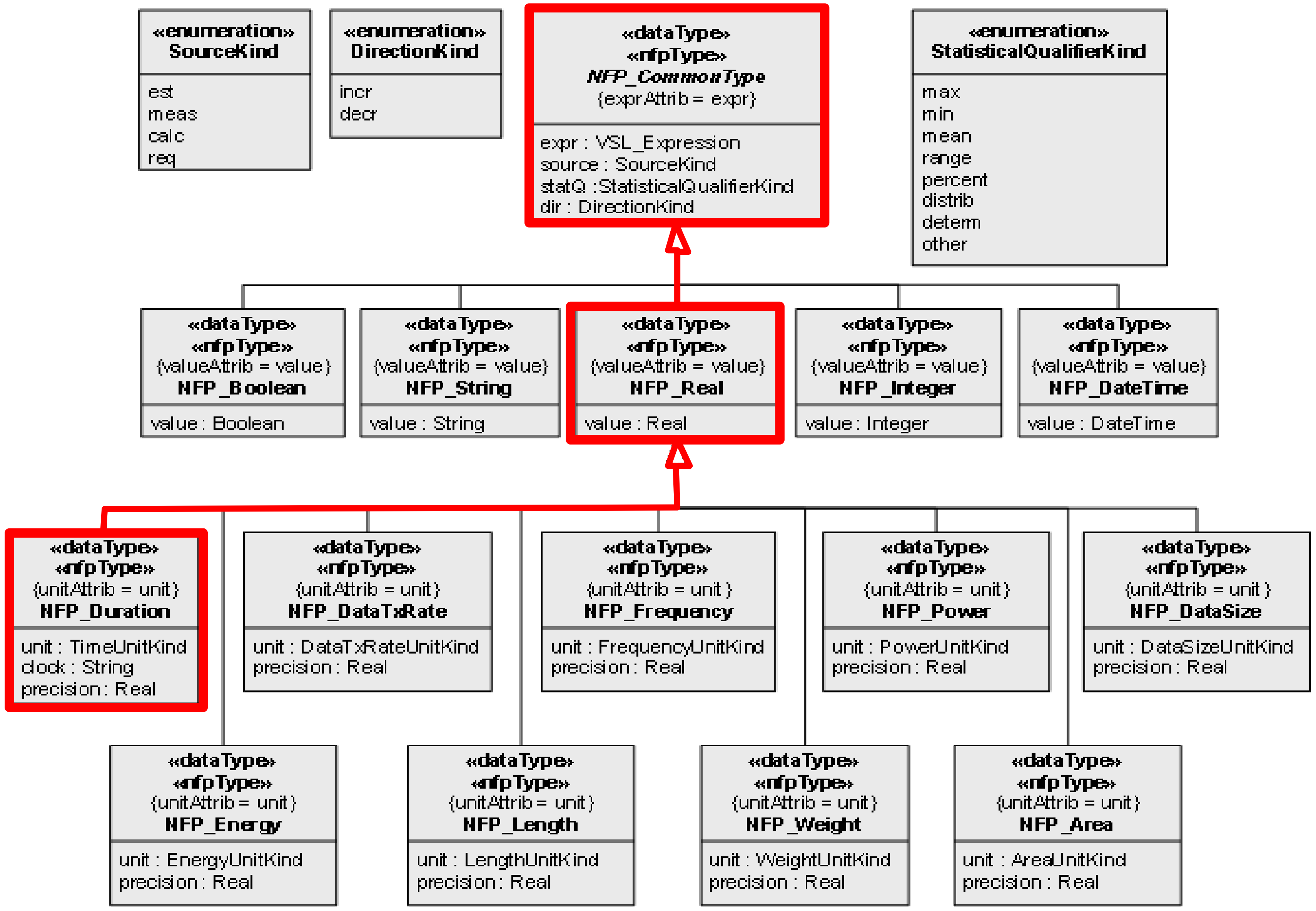
- Because platform resources are often shared
  - ..often by independently designed applications
  - Contention for resources



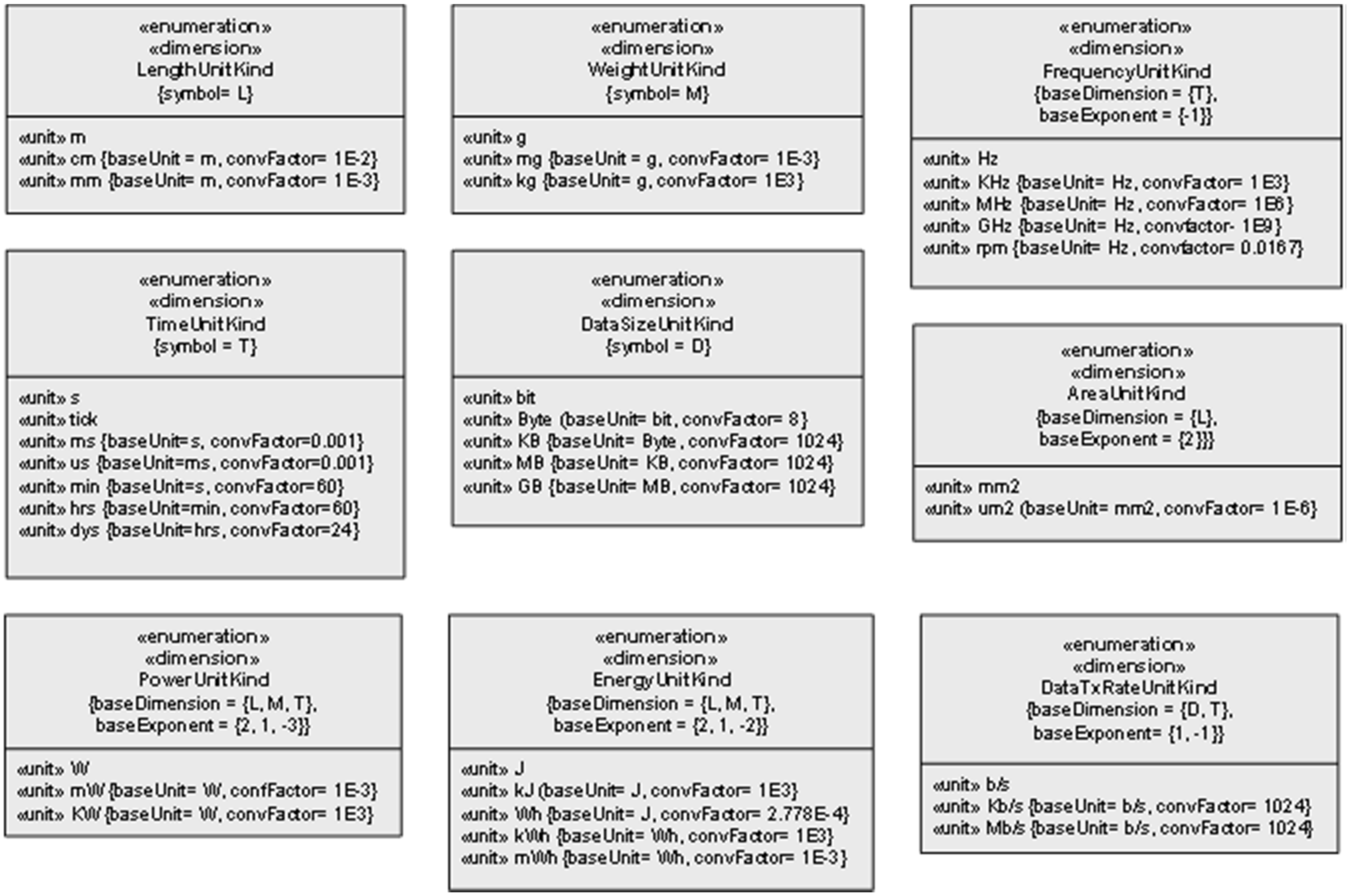
Quantitative QoS Values

- Expressed as an *amount of some physical measure*
- Need a means for specifying physical quantities
  - *Value*: quantity
  - *Dimension*: kind of quantity (e.g., time, length, speed)
  - *Unit*: measurement unit (e.g., second, meter, km/h)
- However, additional optional qualifiers can also be attached to these values:
  - *source*: estimated/calculated/required/measured
  - *precision*
  - *direction*: increasing/decreasing (for QoS comparison)
  - *statQ*: maximum/minimum/mean/percentile/distribution

MARTE Library: Predefined Types


«enumeration» SourceKind
est
meas
calc
req
«enumeration» DirectionKind
incr
decr
«dataType» «nfpType» NFP_CommonType {exprAttrib = expr}
expr : VSL_Expression
source : SourceKind
statQ :StatisticalQualifierKind
dir : DirectionKind
«enumeration» StatisticalQualifierKind
max
min
mean
range
percent
distrib
determ
other
«dataType» «nfpType» {valueAttrib = value} NFP_Boolean
value : Boolean
«dataType» «nfpType» {valueAttrib = value} NFP_String
value : String
«dataType» «nfpType» {valueAttrib = value} NFP_Real
value : Real
«dataType» «nfpType» {valueAttrib = value} NFP_Integer
value : Integer
«dataType» «nfpType» {valueAttrib = value} NFP_DateTime
value : DateTime
«dataType» «nfpType» {unitAttrib = unit} NFP_Duration
unit : TimeUnitKind
clock : String
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_DataTxRate
unit : DataTxRateUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_Frequency
unit : FrequencyUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_Power
unit : PowerUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_DataSize
unit : DataSizeUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_Energy
unit : EnergyUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_Length
unit : LengthUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_Weight
unit : WeightUnitKind
precision : Real
«dataType» «nfpType» {unitAttrib = unit} NFP_Area
unit : AreaUnitKind
precision : Real

MARTE Library: Measurement Units



«enumeration»
«dimension»
LengthUnitKind
{symbol= L}

«unit» m
«unit» cm {baseUnit = m, convFactor= 1E-2}
«unit» mm {baseUnit= m, convFactor= 1E-3}

«enumeration»
«dimension»
WeightUnitKind
{symbol= M}

«unit» g
«unit» mg {baseUnit = g, convFactor= 1E-3}
«unit» kg {baseUnit= g, convFactor= 1E3}

«enumeration»
«dimension»
FrequencyUnitKind
{baseDimension = {T},
baseExponent = {-1}}

«unit» Hz
«unit» KHz {baseUnit= Hz, convFactor= 1E3}
«unit» MHz {baseUnit= Hz, convFactor= 1E6}
«unit» GHz {baseUnit= Hz, convfactor- 1E9}
«unit» rpm {baseUnit= Hz, convfactor= 0.0167}

«enumeration»
«dimension»
TimeUnitKind
{symbol = T}

«unit» s
«unit» tick
«unit» ms {baseUnit=s, convFactor=0.001}
«unit» us {baseUnit=ms, convFactor=0.001}
«unit» min {baseUnit=s, convFactor=60}
«unit» hrs {baseUnit=min, convFactor=60}
«unit» dys {baseUnit=hrs, convFactor=24}

«enumeration»
«dimension»
DataSizeUnitKind
{symbol = D}

«unit» bit
«unit» Byte (baseUnit= bit, convFactor= 8}
«unit» KB {baseUnit= Byte, convFactor= 1024}
«unit» MB {baseUnit= KB, convFactor= 1024}
«unit» GB {baseUnit= MB, convFactor= 1024}

«enumeration»
«dimension»
AreaUnitKind
{baseDimension = {L},
baseExponent = {2}}

«unit» mm2
«unit» um2 (baseUnit= mm2, convFactor= 1E-6}

«enumeration»
«dimension»
PowerUnitKind
{baseDimension = {L, M, T},
baseExponent = {2, 1, -3}}

«unit» W
«unit» mW {baseUnit= W, confFactor= 1E-3}
«unit» KW {baseUnit= W, convFactor= 1E3}

«enumeration»
«dimension»
EnergyUnitKind
{baseDimension = {L, M, T},
baseExponent = {2, 1, -2}}

«unit» J
«unit» kJ (baseUnit= J, convFactor= 1E3}
«unit» Wh {baseUnit= J, convFactor= 2.778E-4}
«unit» kWh {baseUnit= Wh, convFactor= 1E3}
«unit» mWh {baseUnit= Wh, convFactor= 1E-3}

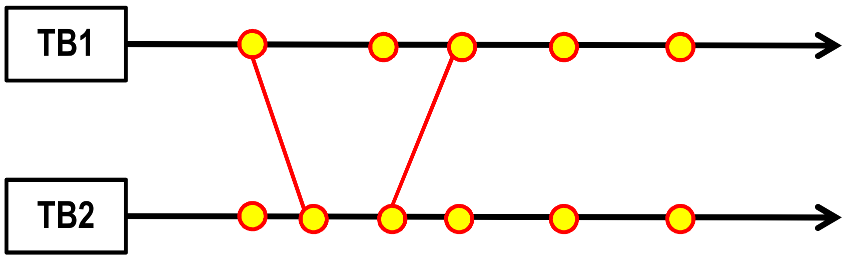
«enumeration»
«dimension»
DataTxRateUnitKind
{baseDimension = {D, T},
baseExponent= {1, -1}}

«unit» b/s
«unit» Kb/s {baseUnit= b/s, convFactor= 1024}
«unit» Mb/s {baseUnit= b/s, convFactor= 1024}

Explicit Approach: Topics Covered

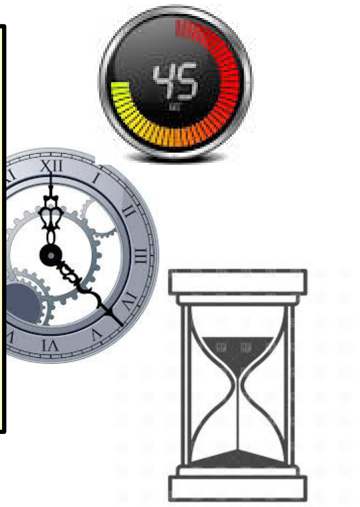
Structure of Time

- time bases
- multiple time bases
- instants
- time relationships



Access to Time

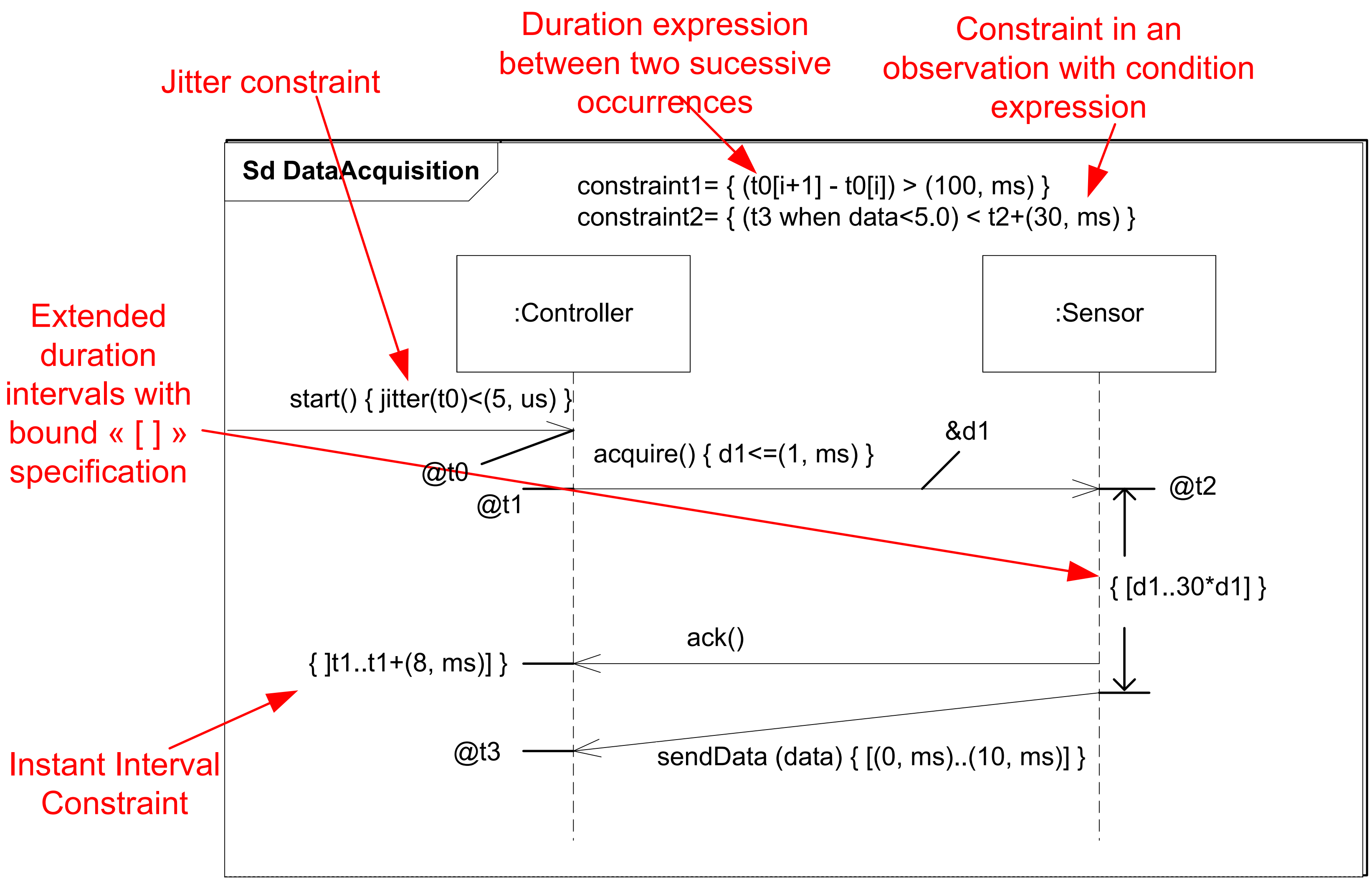
- clocks
- logical clocks
- chronometric clocks
- current time



Using Time

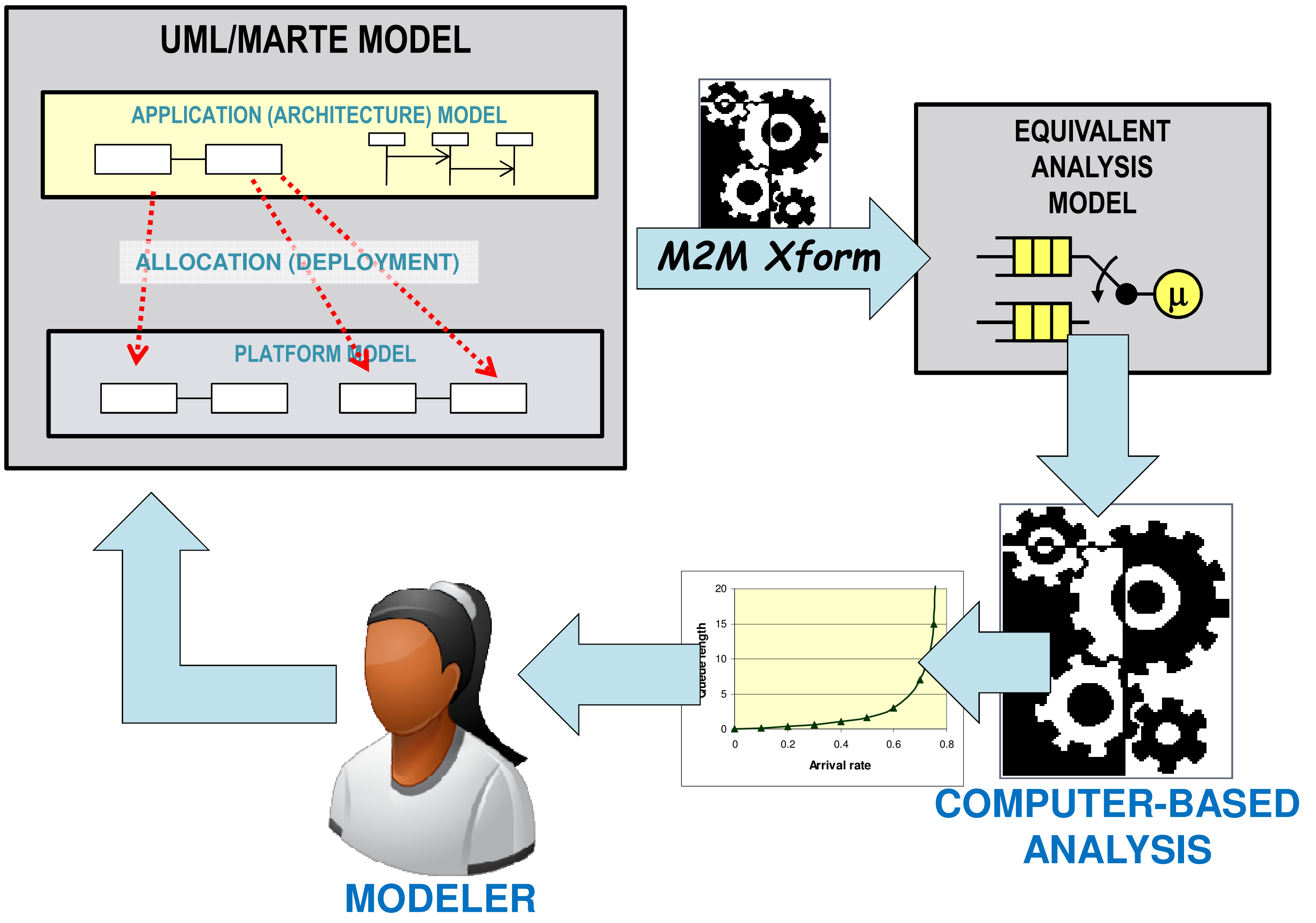
- timed elements
- timed events
- timed actions
- timed constraints

Example: Time Annotations



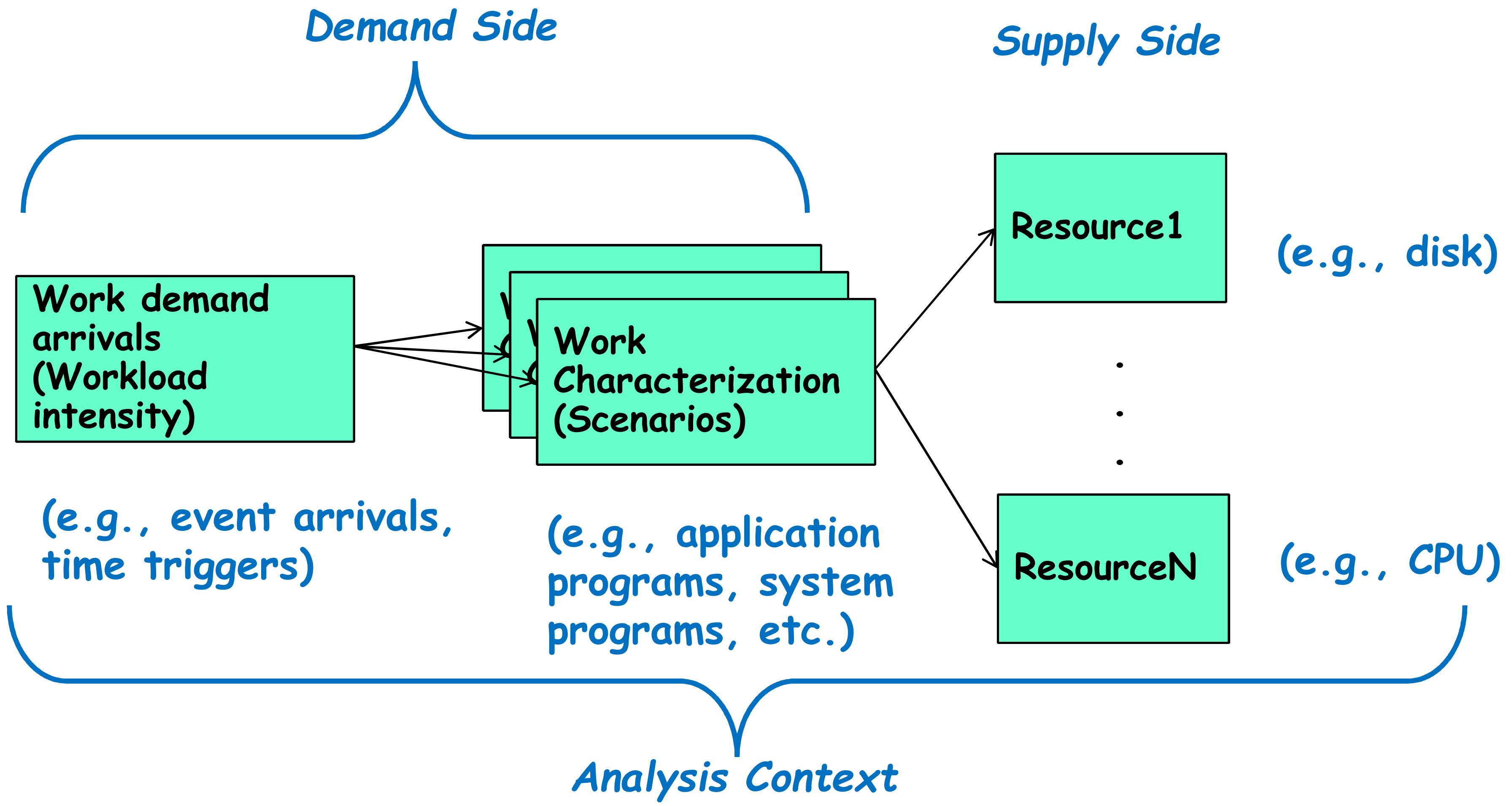
Jitter constraint
Duration expression between two sucessive occurrences
Constraint in an observation with condition expression
Sd DataAcquisition
constraint1= { (t0[i+1] - t0[i]) > (100, ms) }
constraint2= { (t3 when data<5.0) < t2+(30, ms) }
:Controller
:Sensor
Extended duration intervals with bound « [ ] » specification
start() { jitter(t0)<(5, us) }
@t0
acquire() { d1<=(1, ms) }
&d1
@t1
@t2
{ [d1..30*d1] }
ack()
{]t1..t1+(8, ms)] }
@t3
sendData (data) { [(0, ms)..(10, ms)] }
Instant Interval Constraint

MARTE Support for Computer-Aided Analysis



Generic Quantitative Analysis Model (GQAM)

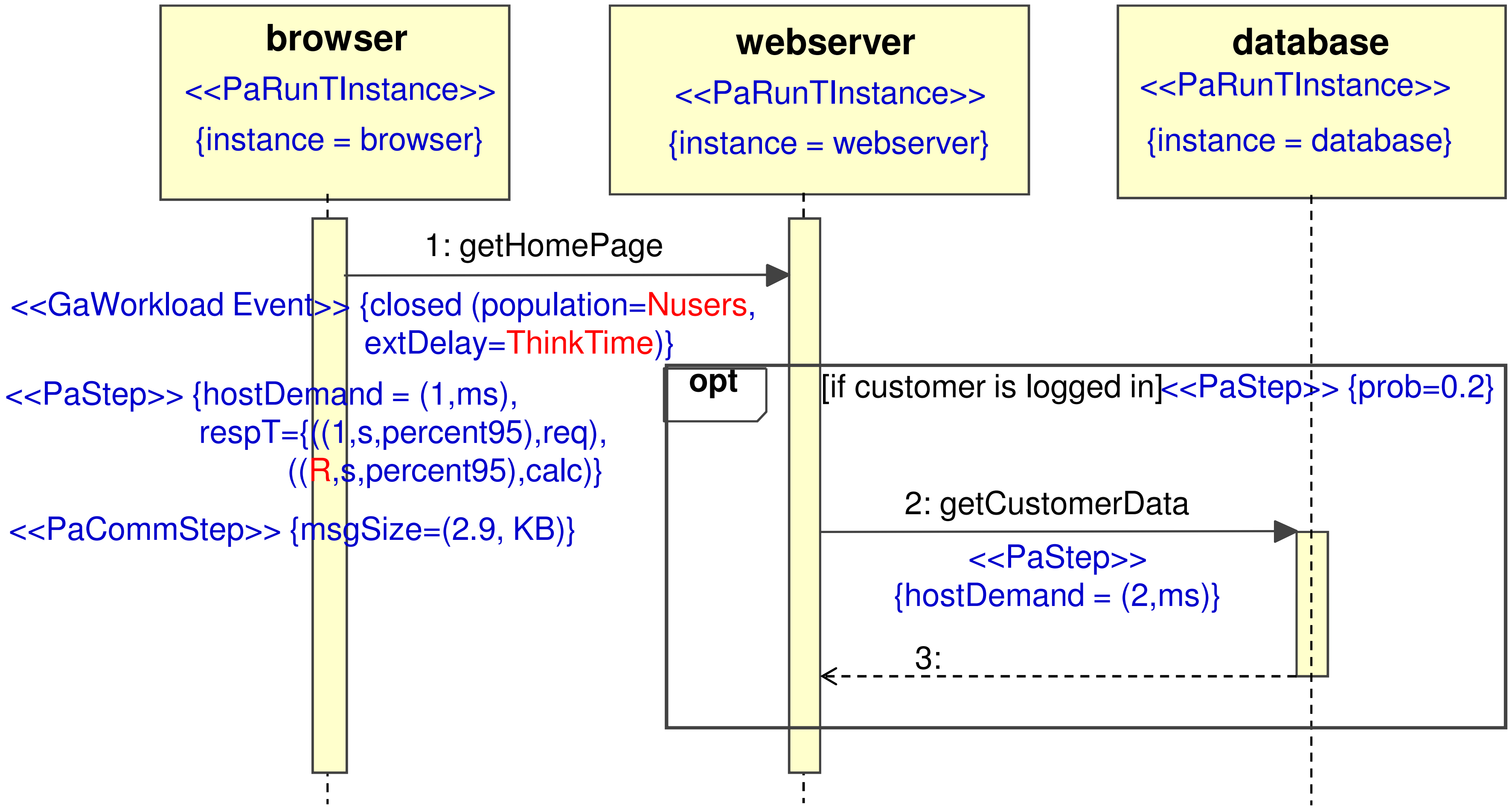
- Captures the pattern common to many different kinds of quantitative analyses (using concepts from GRM)
  - Specialized for each specific analysis kind


Demand Side
Supply Side
Work demand arrivals (Workload intensity)
Work Characterization (Scenarios)
Resource1
ResourceN
(e.g., disk)
(e.g., CPU)
(e.g., event arrivals, time triggers)
(e.g., application programs, system programs, etc.)
Analysis Context

Performance Analysis Example - Context

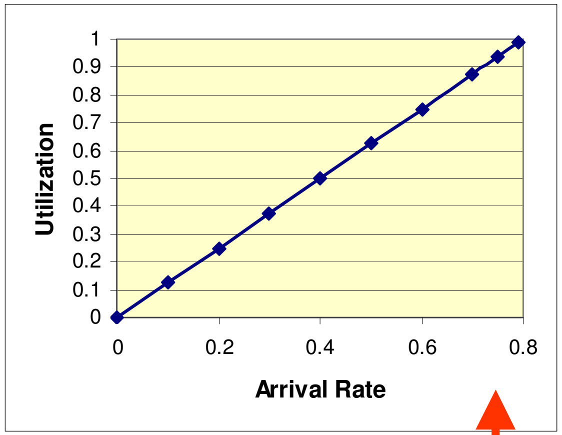
- **An interaction (seq. diagram representation)**

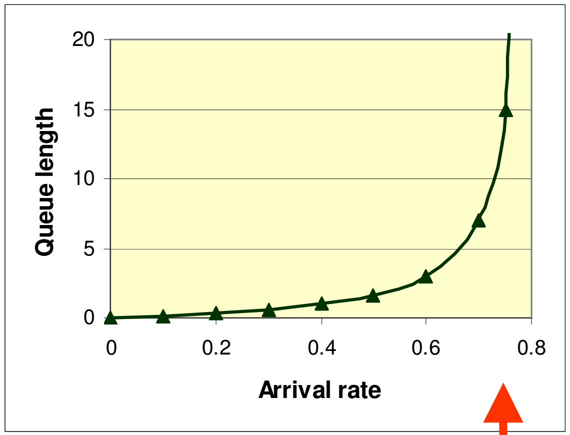
<<GaPerformanceContext>> {contextParams= in$Nusers, in$ThinkTime, in$Images, in$R}


browser
<<PaRunTInstance>>
{instance = browser}
webserver
<<PaRunTInstance>>
{instance = webserver}
database
<<PaRunTInstance>>
{instance = database}
1: getHomePage
<<GaWorkload Event>> {closed (population=Nusers, extDelay=ThinkTime)}
opt
[if customer is logged in]<<PaStep>> {prob=0.2}
<<PaStep>> {hostDemand = (1,ms), respT={((1,s,percent95),req), ((R,s,percent95),calc)}
<<PaCommStep>> {msgSize=(2.9, KB)}
2: getCustomerData
<<PaStep>> {hostDemand = (2,ms)}
3:

Slide courtesy of D. Petriu, M. Woodside (Carleton U.)

Typical Performance Analysis Results





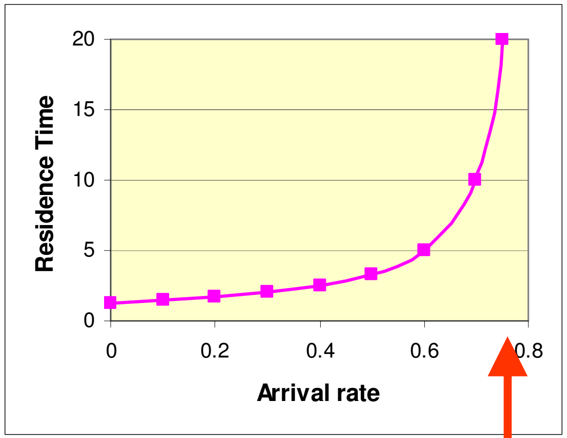
Utilization

saturation

Residence Time

Queue length

saturation



saturation

Slide courtesy of D. Petriu, M. Woodside (Carleton U.)

Summary

- Software is increasingly more integrated into everyday operations, which involves an ongoing interaction with the physical world
- Our mainstream programming languages are not well suited for this environment
- Needed: Higher-order languages that are more directly connected to this environment
  - ⇒ Model-based technologies and practices
  - ⇒ Higher levels of abstraction and automation
- Still a research topic, but we already have a number of important components of the solution

- THANK YOU-

QUESTIONS,
COMMENTS,
ARGUMENTS...

Supplementary Slides

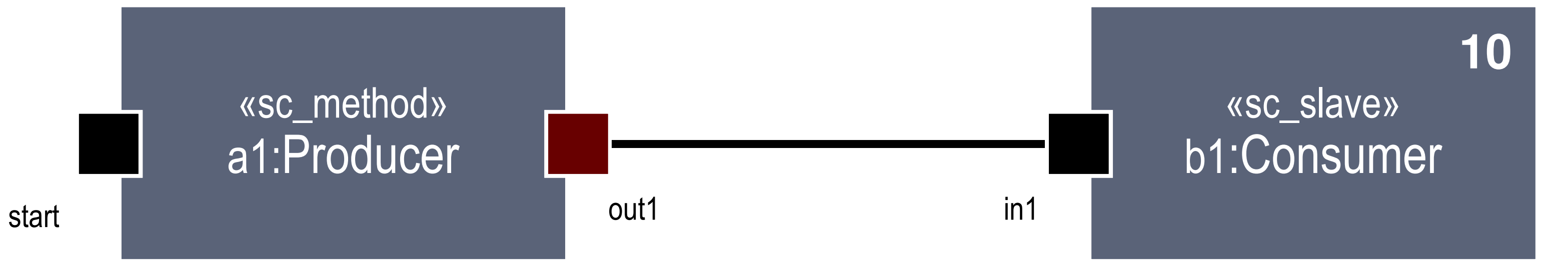
Accidental Complexity or Why It's Called "Code"*

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

Can you see what this program is doing?

Code: a system used for brevity or secrecy [Dictionary.com]

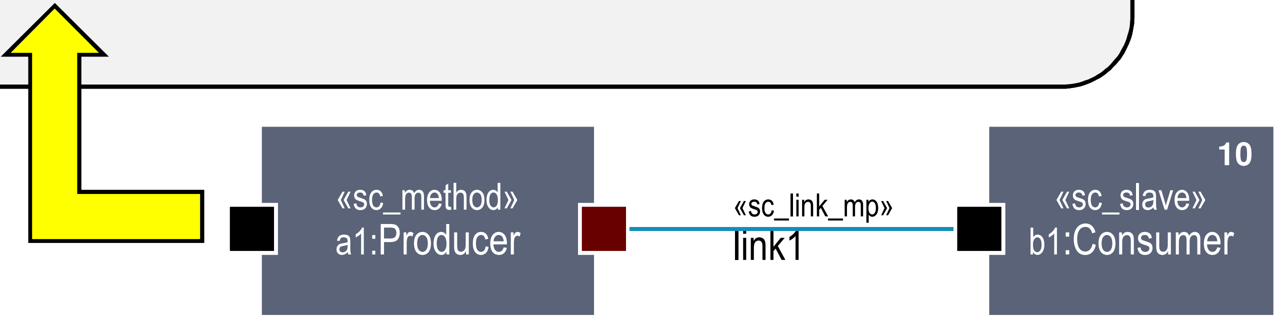
The Corresponding UML Model



Can you see it now?

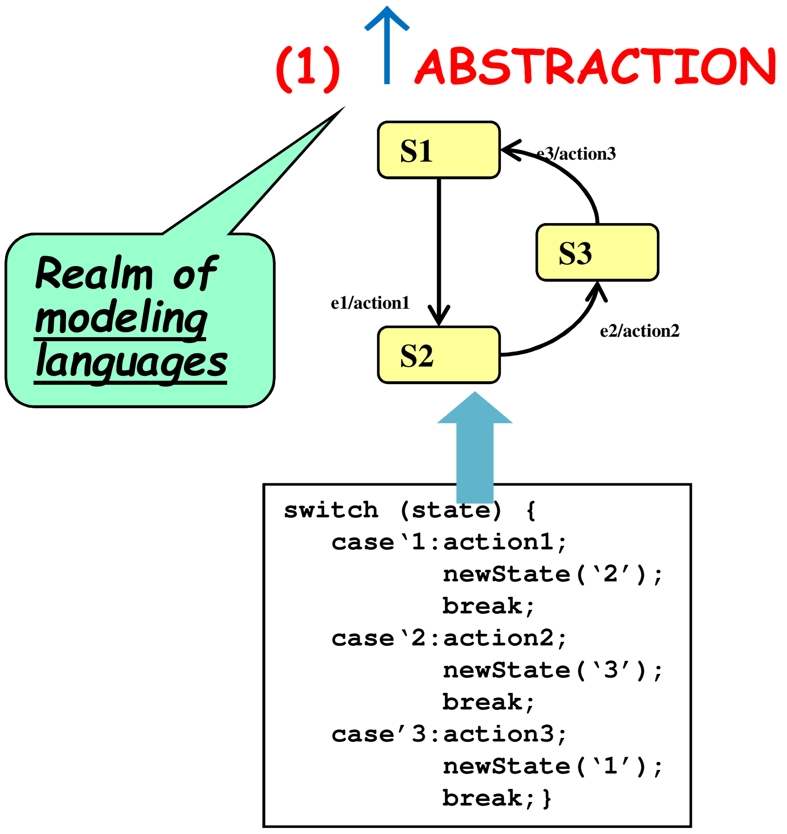
Plus the Power of Computer Automation

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

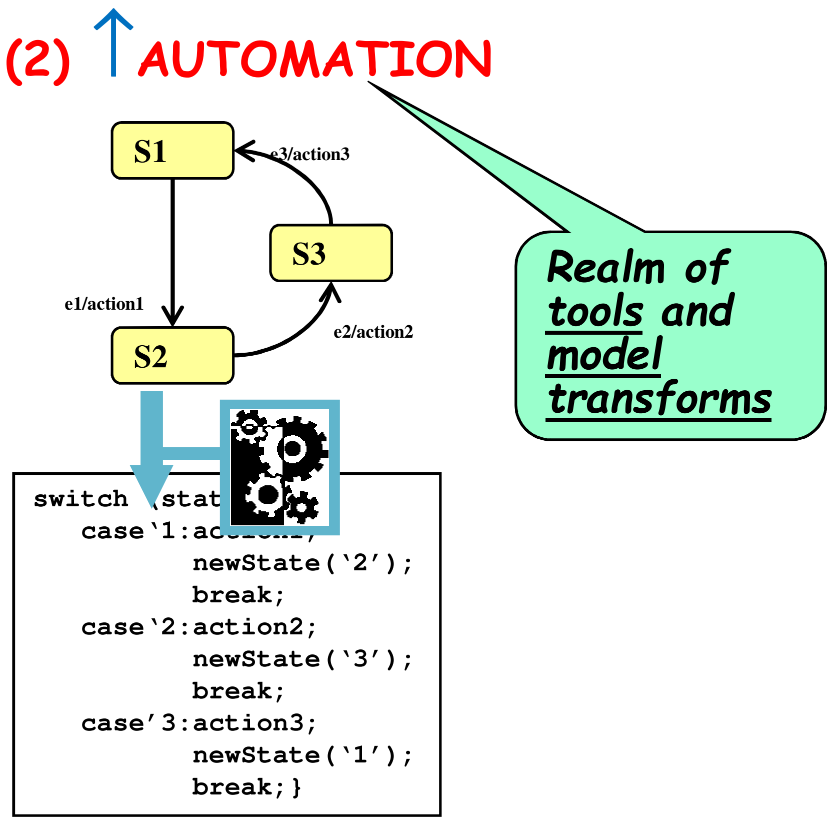


Model-Based Engineering: The Essential Coping Approach

- An approach to system and software development in which computer-based software models play an <u>indispensable</u> role
- Based on two time-proven premises:



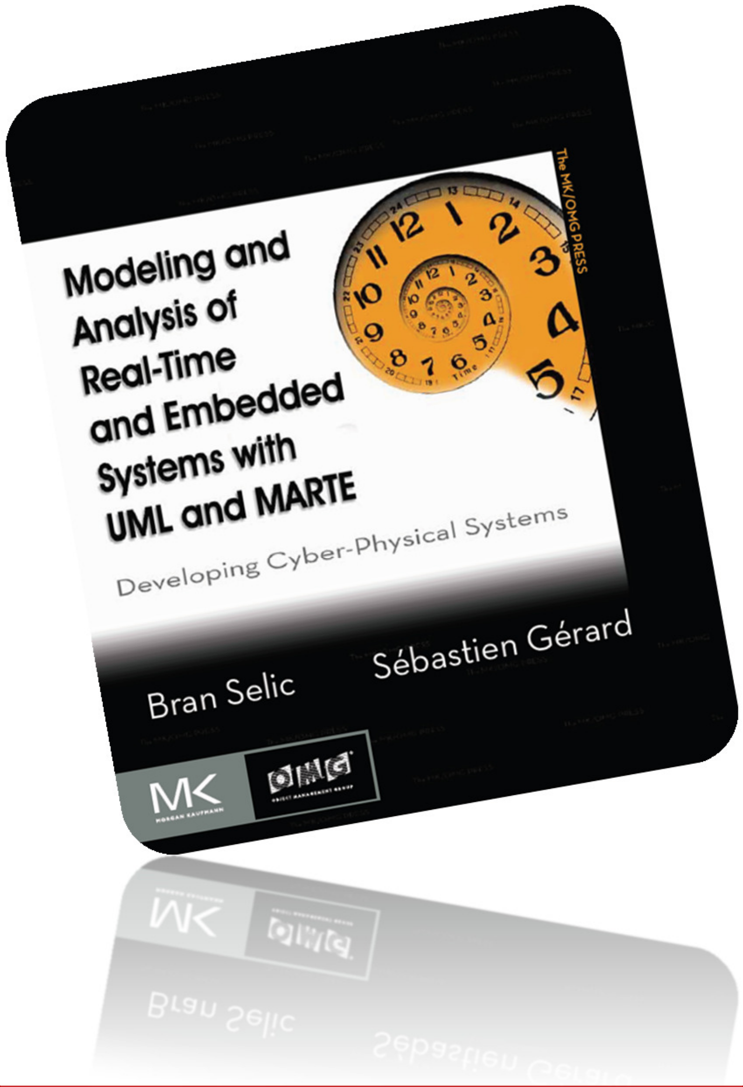
```
switch (state) {
   case`1:action1;
          newState(`2');
          break;
   case`2:action2;
          newState(`3');
          break;
   case'3:action3;
          newState(`1');
          break;}
```



```
switch (state) {
   case`1:action1;
          newState(`2');
          break;
   case`2:action2;
          newState(`3');
          break;
   case'3:action3;
          newState(`1');
          break;}
```

A shameless plug

Available from a web page/bookstore near you:

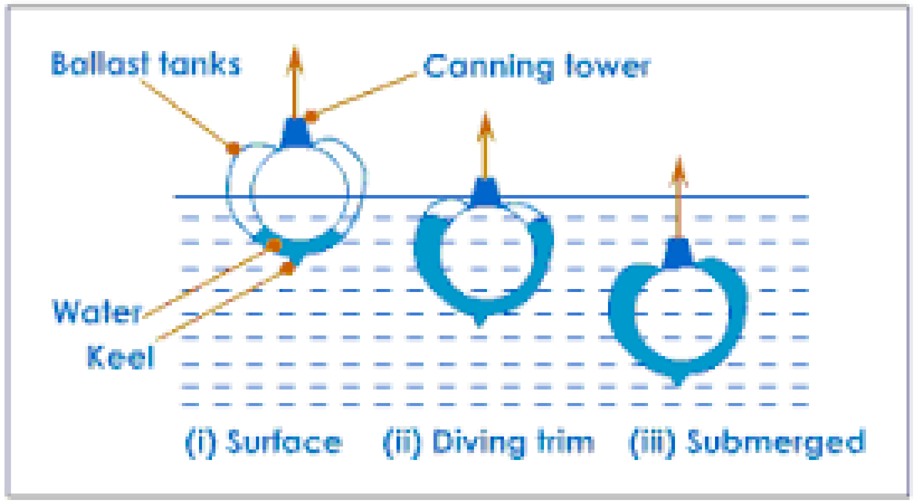

Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE
Developing Cyber-Physical Systems
Bran Selic
Sébastien Gérard
The MK/OMG PRESS
MK MORGAN KAUFMANN
OMG

Publisher: Morgan Kaufmann
ISBN: 978-0-12-416619-6

The “Software Crisis”

- **Systems of this type were designed primarily by classical engineers (mechanical, electrical, radio, etc.) and physicists**
  - Software was viewed as a simple *production* problem (i.e., writing the code) – as opposed to a *research* problem
  - *It is still a common attitude today among many traditional engineering professionals*
    - A “soft” science: difficult to make irrefutable assertions or predictions
- **But, the software problems of SAGE and similar systems exposed the difficulties of designing reliable software**
  - 1968 NATO Conference on Software Engineering ⇒ “software crisis”

Functionality vs. Engineering



Functionality (Logic)

- *Air conditioning*
- *Plumbing*
- *Electrical wiring*
- *Water recycling*
- *Waste management*
- *Steering*
- *etc.*

… and its Engineering Manifestation

But, does this paradigm apply to software?