On Control in Cyber-Physical Systems

Bran Selić

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

(selic@acm.org)
WHAT'S WHAT:
CONTROL AND CYBER-PHYSICAL SYSTEMS
**Definitions: Control [INCOSE*]**

- **CONTROL** – A means or a device to regulate a process or sequence of events

- **CONTROL SYSTEM** – A system which responds to input signals from the process and/or from an operator and generates output signals causing the EUC [equipment under control] to operate in the desired manner

![Diagram of control system](Image)

**Q:** Who controls the control system while the control system is busy controlling?
Definitions: Cyber-Physical Systems (CPS)

- **Cybernetics** - the scientific study of control and communication in the animal and the machine [Norbert Weiner, 1948]
  - Greek root: κυβερνητική = governance/control

- **Cyber-physical system** - integrations of computation with physical processes [Ed Lee, 2008]
  - i.e., systems in which control is achieved (in part) using computer software
  - A holistic view of systems (vs. embedded/real-time, which focuses on the software)

“In the physical world, the passage of time is inexorable and concurrency is intrinsic. Neither of these properties is present in today’s computing and networking abstractions.”

The Case of the Mars Climate Orbiter

Mars Climate Orbiter

"The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software..."

-- NASA report, 1999

Conventional programming languages have no first-order concept of physical processes or dimensions

e.g., delay(100);
force:Force = 225;
What is a “First-order” Language Concept?

- A concept whose semantics are part of the definition of the language and which are directly supported by associated computer-based tools (e.g., compilers, interpreters)

- Example:
  - Compilers “understand” the concept of “real” number but not the concept of “force”

```plaintext
p, q, s : Real;
s := p + q;

enum FUnit: {N, kN};
deftype Force{
    value: Real,
    u: FUnit};
p, q, f : Force;
f := p + q;
```

⇒ A compiler cannot neither properly validate programs that include second-order types nor generate the appropriate code
CHALLENGES IN CONTROLLING CYBER-PHYSICAL SYSTEMS

Two primary sources:

- The physical world
- The software
The “Nasties” of the Physical World

- **Concurrency**: multiple temporally overlapping phenomena of interest
- **Asynchrony**: events of interest that occur outside expected or desired order
  - ...including occurrences of **partial failures** of software and/or equipment
- **Physical constraints**: limits on the speed and reliability of communication and computation, limited resources (e.g., memory, processors, energy), etc.

Q: Can't we simply mask those out using appropriate “transparency mechanisms” (reliable communications, fault masking, virtual memory, etc.)?
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.

The reliability requirements of an application can only be fully realized by mechanisms that are aligned with the specific needs of that application.

- The application [writer] is in the best position to decide how to deal with reliability issues (e.g., failures).
Construction materials and tools can have a fundamental impact on design choices in traditional engineering.

Q: What is software made of and does it matter?
A Quick Quiz

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

A: None of them

(a) MITS Altair 8800 (8080 CPU) 4KB
(b) Sinclair ZX81 (Z80 CPU) 8KB
(c) Lenovo ThinkPad X61 (Intel® Core™2 Duo CPU) 1GB
The Functional vs “Non-functional”* Separation

A parable...

... Can your grandmother swim?
... Yes.
... Excellent! Then let’s put her on the Olympic Swimming Team.
... But, but... she’s not a very fast swimmer.
... That’s OK, we’ll work on that once she’s on the Team.

- Can be a very dangerous separation of concerns
  - ...because the “non-functional” can often have a major (e.g., architectural) impact on the functional
- Separate only if justified following adequate up front analysis

* A particularly ill-chosen term:
  - Says nothing about what it is
  - Suggests that this is a second-order concern
Platforms: The Raw Material of Software

Platform: The full complement of supporting software and hardware required for a given application program to execute correctly.
Platforms are the mediators through which software interacts with its environment (including the physical world).

Everything that the software senses and performs is influenced by the physical characteristics of the platform.
If we are at all concerned about the quality of service (QoS) characteristics of our software application (e.g., response time, resource usage), we must factor in the effects of its underlying platform in the course of design.
But, What About Platform Independence?

- An important and useful notion
  - Helps abstract away irrelevant technological detail
  - Necessary for software portability

- *Platform independence does not imply 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 computing and communications structure
2. Its relevant physical properties (size, capacity, performance, bandwidth, etc.)
3. The relationship of the application software elements to platform elements (i.e., deployment)
A MODELING LANGUAGE APPROACH TO DESIGNING SOFTWARE FOR CPS
Example: The MARTE Domain-Specific Language

- **MARTE** = *Modeling and Analysis of Real-Time and Embedded systems*
  - An OMG industry standard that supplements the UML modeling language

- **Provides:**
  - A rich model of time
  - A domain-specific language for modeling quality-of-service (QoS)-sensitive applications and platforms
  - Precise specification of QoS values and their semantics
  - Support for formal analyses and predictions of system QoS characteristics

**NB:** MARTE is discussed here merely as an example of the general approach required in designing computer languages for CPS
Many QoS specifications are related to time ⇒ need to be explicit about the assumed model of time

**Structure of Time**
- time bases
- multiple time bases
- instants, durations
- time relationships

**Access to Time**
- clocks and their characteristics (resolution, maximum value, drift, accuracy…)

**Specifying Time Values**
- Time specifications (deadlines, intervals, time-of-day, etc.)
Core Concept: Resource

  - “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 configuration of different resources*, which can be drawn upon by applications

```
Platform -- 1..* Resource

Resource -- Computing Resource

Resource -- Memory Resource

etc.
```
Core Concept: Resource Services

Resources are viewed as **service providers**
- Consequently, applications are viewed as **service clients**

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

![Diagram of Resource to Resource Service with 1..* relationship]

e.g. (platform services):
- memory provisioning
- processing power
- bandwidth
- energy
- mutual exclusion
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)

Most engineering stress analyses consist of calculating whether (QoS) supply can meet (QoS) demand
QoS Compatibility

- Qualitative compatibility
  - Compatible physical dimensions (e.g., length, bandwidth)

- Quantitative compatibility
  - Does supply meet demand

Key engineering question:
\[(\text{RequiredQoS} \leq \text{OfferedQoS})\]?

It would be very useful if this type of compatibility check could be performed automatically (e.g., by a compiler).
Why Resource Analysis is Difficult

- Because many platform resources are shared
  - ...often by *independently-designed but competing* applications

- Unfortunately, aggregation of resource service requests is resource-type specific
  - Each resource type (bandwidth, memory, CPU power, energy...) combines idiosyncratically
  - No common algebra
MARTE Resource Types

- Resource
  - Storage Resource
  - Communication Resource
  - Timing Resource
    - Synch Resource
    - Concurrency Resource
    - Computing Resource
    - Device Resource
Physical (Data) Types

- Expressed as an amount of some physical measure
- Need a means for specifying physical quantities
  - Dimension: kind of quantity (e.g., time, length, speed)
  - Unit: measurement unit (e.g., second, meter, km/h)
    - Default unit and relationships to it
  - Value: quantity
- 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
Specifying Deployment

: MyApplication

appComp1 : C1

appComp2 : C2:

: SomeOperatingSystem

appProcess : Process [256]

«allocate»
{nature = spatialDistribution, kind = structural}

c : LogicalCPU

m : LogicalMemory

«allocate»
{nature = spatialDistribution, kind = structural}

: SomePhysicalProcessor

«allocate»
{nature = spatialDistribution, kind = structural}

«allocate»
{nature = timeScheduling, kind = structural}

PhysicalCPU

PhysicalMemory

«allocate»

{nature = spatialDistribution, kind = structural}
Achieving Meaningful Platform Independence

• Inspired by the COBOL approach
  • “Acceptable platform”: a platform and deployment specification – which is included in the application model – that identifies expected and tolerated QoS ranges
  • Any platform that satisfies the abstract platform is valid
This combination of models can be formally analyzed.
For Those Interested in MARTE

More approachable MARTE “cookbook”:

Publisher: Morgan Kaufmann
ISBN: 978-0-12-416619-6
TWO ARCHITECTURAL DESIGN PATTERNS FOR CPS SOFTWARE

1. Layering (for platform modeling)
2. Recursive control (for system control)
What does it actually mean?

Should D1 have access to A3 (3 levels below)?
Making it a Bit More Concrete

Should Applications have access to File System (3 levels below)?
The primary purposes of encapsulation in software:

- Abstraction
- To reduce coupling by hiding implementation from externals and vice versa!

Q: Are there situations when it is not possible to encapsulate the complete implementation of a software component?
"Platform" Layering

- Distinguishing characteristics
  - Upper layer: realizes some desired functionality
  - Lower layer: a set of potentially shared services used to implement the upper layer = “platform”

- Example:

  ![Diagram of platform layering](image)

  - Upper layer entities depend on the services of the platform layer for their implementation
  - These interfaces are implementation specific and should be explicit (but not public)
  - Lower layer entities do not depend on the upper layer entities, but do participate in their implementation
“To understand the capabilities of a black-box component it is sufficient to know ALL of its interfaces”
In most systems, layering is a complex multidimensional relationship

- Most real system architectures cannot be described accurately by a single vertical layer stack
- Most vertical stacks are merely architectural views along a particular viewpoint
Failed Representations of Layering

- **Staircase model**

  ![Staircase model diagram]

- **Toaster model**

  ![Toaster model diagram]
Layering in Real Systems

In practice, real system architectures consist of multiple layer "stacks"

- Each stack represents a particular viewpoint (domain concern)
- The elements of individual stacks may be shared between stacks
- All stacks ultimately converge on the hardware
2. Recursive Control: Example Design Problem

- Design the software architecture of network router (switch) that supports multiple end points using the alternating-bit protocol.
The Alternating Bit Protocol: Spec 1

- A simple positive acknowledgement protocol with retransmission

Diagram:
- e1: EndUser
- :Sender
- :Receiver
- e2: EndUser

Data Flow:
- data1
  -pktA [data1]
  -ackA
  -pktB [data2]
  -ackB
  -pktA [data3]
- data1
- data2
- data3
State machines of sender and receiver ends

- Define the primary functionality of the switch
A (Simplified) Architecture

Purpose: CPS control

ABsender
ABreceiver

SWITCH

Logging Subsystem
Operator Interface

Lines Controller
Lines Diagnostic Subsystem

Database Manager

Purpose: CPS control control

Operator

Database

The control software of a CPS itself needs to be controlled!
**Software control**: The set of mechanisms and processes required to bring a software control system into desired operational states and to maintain it in those states in the face of both desirable and undesirable (i.e., asynchronous) inputs.

This includes:

- System and component start up, restart, and shutdown
- Failure detection, reporting, diagnosis, and recovery/masking
- System administration, maintenance, and dynamic evolution

Q: Who controls the control system while the control system is busy controlling?

A: Software control (software)
Implementing the Sender

 Sounds simple enough…

- Just Started
- Get Line Parameters
- Checking Hardware*
-Failed
- Analyzing Failure*
- Ready
- Wait for A data
- Wait for B data
- Wait for ackA
- Wait for ackB

* Indicates a failure condition.
A Separation of Concerns: Control vs. Function

System “modes” = states of the control automaton

Just Started

Get Line Parameters

Checking Hardware*

Ready

System “modes” = states of the control automaton

Operational

Failed

Just Started

Get Line Parameters

Checking Hardware*

Ready

Operational

Failed

NB: The same control policy could be applied to the Receiver!

Wait for A data

Wait for B data

Wait for ackB

Wait for ackA

ackB/
data/send PktB

timeout/(re)send PktB

ackA/
data/send PktA

timeout/(re)send PktA

NB: The same control policy could be applied to the Receiver!
Some Important Observations

- **Control predicates function**
  - Before a system can perform its primary function, it first has to reach its operational state

- **Software control behavior patterns are often independent of function of the software**
  - E.g., the Sender and Receiver are controlled in the same way
  - The process by which a system reaches its operational state is often the same regardless of the specific functionality of the component
Basic Design Principles

- **Separate software control from software function**
  - Separate software control components from functional components
  - Define explicit interfaces dedicated to control
  - Embed functional behavior within control behavior

- **Distinguish between control policies (i.e., control decisions) from control mechanisms (control realization)**
  - Minimizes impact of changes of control policy

- **Centralize decision making**
  - As much as possible minimize the need for distributed consensus in making control decisions
  - Define dedicated control components (decision makers) that implement control policies
The Core Architectural Pattern

- A star-like pattern with control in the centre
  - ... and distinct control interfaces

The control policies are realized by control mechanisms within in the (functional) components

All control policies are defined and dictated from here

Dedicated control interface

Central Control

Group of components that need to be controlled as a unit (e.g., a layer “stack”)

- Controlled Component [1]
- Controlled Component [2]
- ... Controlled Component [n]
**The Recursive Control Pattern**

- Scales easily to very large systems
- Simple, but ensures consistent and highly controllable dynamic software systems

Note that the controllers can also be controlled in the same way as other controlled components.
All controlled components that share the same control automaton can be subclasses of a common abstract class.

```
Controlled Component

controlPort: Ctrl

ABsender

ABreceiver

. . . etc.
```
Weaving Using Hierarchical States

- Achieves clean separation of control and functional behaviors

controlled Component

controlPort: Ctrl

Running

ABsender
**Summary**

- Cyber-physical systems are systems in which software control plays a key role.
- This software must account for the idiosyncrasies of CPS, including the impact of physical phenomena as well as software issues that need to be controlled.
- This requires domain-specific software technologies and design solutions.
- We have examined two proven aspects of solutions to some of the key aspects of the CPS design problem:
  - An approach to designing CPS-specific modeling languages.
  - Suitable architectural patterns.