From Control Loops to Software

Oded Maler (based on Paul Caspi)

CNRS-VERIMAG
Grenoble, France

September 2007

Let's Get Personal

Paul Caspi and myself have shared an office for 11 years. During this period Paul had to be exposed to various things, not all pleasant

Hear noisy music, multi-participant meeting on diverse topics

Feed my fish in my absence and participate in some related funerals

Bare with patience severe abuse of the French language both in terms of grammer and pronounciation

Translate to French my letters to the bank, reports to the CNRS etc.

Hear a stream of infantile provocations against many things dear to his heart, both scientifically, culturally and politically

The next slide is a succinct demonstration of what he had to go through. It should not be taken too seriously

Paul Caspi, a Living Oxymoron

Paul Caspi:

Is French... but modest

Finished a “grand ecole” ... but does not cause damage

Is called Caspi.. but does not care too much about money

Is part of the French “left” ... but is open to concrete foreigners

Works on “synchronous languages” ... but is also interested in science

Is an engineer ... but acknowledges the existence of things that do not exist

Executive Summary

Embedded systems $\approx$ realization of control systems by computers

Computers are the major medium for realizing controllers

There is a gap between the world views of control and of computation

Consequently there is no nice and coherent theory to cover the practice (sampled systems theory treats only part of the problem)

We try to remove some of the confusion (or replace it with another)

Plan

1) A high-level historical and philosophical discussion of control and computation

2) From a simple PID controller all the way to implementation

Further issues discussed in

P. Caspi, O. Maler, From Control Loops to Real-Time Programs, Handbook of Networked and Embedded Control Systems, 2005:

3) Multi-periodic control loops and their scheduling on a sequential computer

4) Discrete event (and hybrid) systems and their software implementation

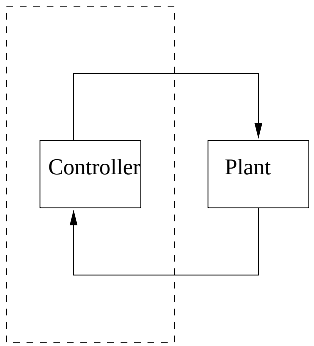
5) Distributed control and fault-tolerance

Controllers and Feedback Functions

A mechanism that interacts with part of the world (the “plant”) by measuring certain variables and exerting some influence in order to steer it toward desirable states

The rule that determines what the controller does as a function of what it observes (and of its own state) is called the feedback function

Prehistory: feedback function “computed” physically (Watt Governor)

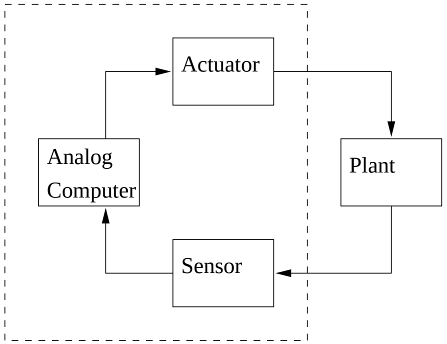


Control by Analog Computation I

Decoupling the computation of the feedback function from measurement and actuation

Physical magnitudes transformed, via sensors, into low-energy electric signals which are fed into an analog computer

The computer outputs electric signals which are converted into physical quantities and fed back to the plant

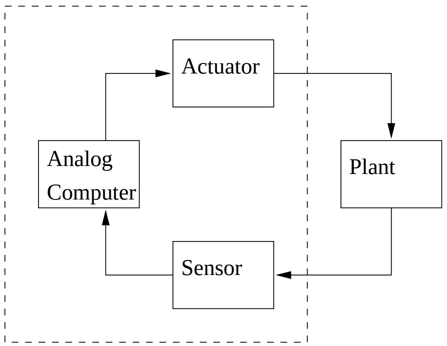


Control by Analog Computation II

This new architecture poses no conceptual/mathematical problems. The plant is viewed as a continuous dynamical system $\dot{x} = f(x, d, u)$ with state x, disturbance d and control input u

The electrical analog controller can be viewed as a system that computes u according to $\dot{u} = g(u, x, x_0)$ (x_0 is a reference signal)

The closed loop system is obtained by combining both systems into a good old continuous system where feedback is computed "continuously"

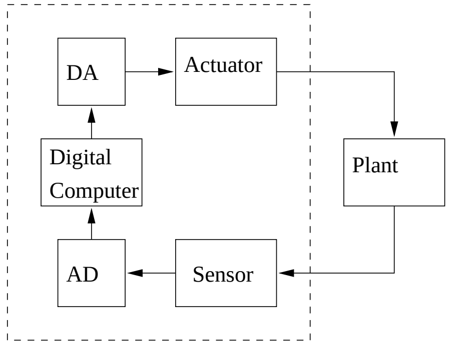


Digital Control I

Computing a function by digital means is an inherently discrete process

Numbers are represented by bits rather than by physical magnitudes

Sensor readings are transformed from analog to digital before the computation, and the results of the computation are transformed back from digital to analog



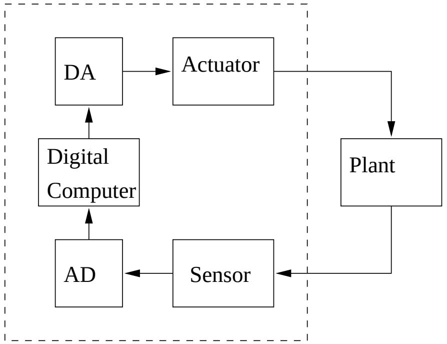
Digital Control II

Something completely different

Computation is done by a sequence of discrete steps that take time

Electrical values on wires are meaningless until the computation terminates

It makes no sense to connect the computer to the plant continuously



Digital Control: Sampling

The interaction of the controller and the plant is restricted to sampling points, a (typically periodic) discrete subset of the real-time axis

At these points sensors are read, the values are digitized and handed over to the computer which computes the value of the feedback function

The outcome is converted to analog and fed back to the plant via the actuators

Between sampling times the output is kept constant, or interpolated by the actuator. There is no feedback

From the control point of view, the sampling rate is determined by the dynamics of the plant. Faster and more complex dynamics requires more frequent sampling

No real theory

Digital Control: the Computer Role

The computer should be able to compute the value of the feedback function (including the A/D and D/A conversions) fast enough, that is, between two sampling points

This requirement is the origin of the term real-time computation

Once this is guaranteed, the control engineer can regard the computer as yet another (discrete time) block in the system and ignore its computerhood

This is true for simple SISO systems, but becomes less and less so when the structure of the control loops becomes more complex

Computation

Prehistory: batch programs for payroll or intensive numerical computations

No interaction with the external world during execution

“Transformational” programs: read their input at the beginning, embark on the computation process and output the result upon termination

Fundamental theories of computability and complexity are tailored to this type of “autistic” computation:

What functions can and cannot be computed (computability)

How the number of computation steps grows asymptotically with the size of the input (complexity)

Remark: The Relativity of Real Time

Even computations of this type are “embedded” in some sort of a larger process

A payroll program is embedded in the “control loop” of the organization, a process of filling time sheets and getting salary at the end of the month

If the program execution time was in the order of a month , this could be considered as real-time programming

So it is always a matter of comparison between time scales of the computation and some external processes

Interactive Computing I

With the advent of time-sharing operating systems computation became more interactive

Typical examples: text editor, a command shell or any other program interacting with one or more users via keyboards and screens

What is the function that such an interactive program "computes"?

Mathematically speaking it can be formulated as a sequential function, mapping sequences of input actions to sequences of responses

The crucial point: the process of computation is not isolated from the input/output process but is interleaved with it

Interactive Computing II

The user types a command, the computer computes a response (and possibly changes its internal state) and so on. These were called "reactive" systems by Harel and Pnueli

It differs from batch programs but still, the environment on the other side is restricted; typically a human user or a computer program following some protocol

The user waits for the computer response before entering the next input

Of course, if you type faster than your editor or transmit faster than the receiver it becomes "real time" (buffer overflow)

Control-Loop Computing

Implementations of control systems interact with the physical world

This player is assumed to be governed by differential equations, and which evolves independently of whether the computer is ready to interact with it

A slow computer may ignore sensor readings or not update actuator values fast enough

In many “time-critical” systems, the ability of the computer to meet the rhythm of the environment is the key to the usefulness of the system

Failing to do so may lead to catastrophic results or to severe degradation in performance

Real-time: tight coupling between the internal time inside the computer and the time of the external world

From Mathematical Descriptions to Programs

Algorithms can be described at various levels of abstraction, for example an abstract graph algorithms can contain a statement: “for every node do”

A more concrete program should specify the data-structure in which the mathematical object is stored, retrieved, etc.

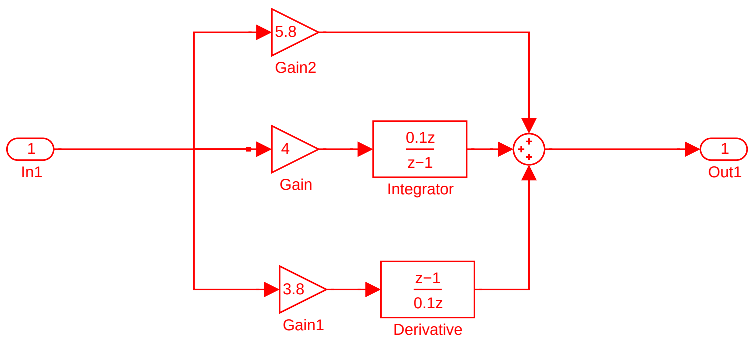
And there is a longer chain of concretizations (assembly, machine code, micro architecture) until the implementation

One of the main achievement of computer science: automatic (and semi-automatic) semantics-preserving transformations between levels

PID Controller from the Control Viewpoint

A PID controller: It takes the input signal I, computes its derivative D and integral S and computes its output O as a linear combination of I, S and D

The controller can be represented using the following block diagram



PID Controller: the Semantics

The controller produces an output sequence O_n as a function of the input sequence I_n

The relation between them is defined via the following recurrence equations:

$$S_{-1} = I_{-1} = 0.0$$

initialization

$$S_n = S_{n-1} + 0.1 \cdot I_n$$

integration

$$O_n = 5.8 \cdot I_n + 4 \cdot S_n + 3.8 \cdot 10.0 \cdot (I_n - I_{n-1})$$

derivative and summation

PID Controller: State and Memory

The state variables of the system include the integral S and an auxiliary variable J memorizing the last input in order to compute the derivative

The controller has memory that has to be maintained and propagated between successive invocations of the program

The appropriate programming construct is a class in an object-oriented language, but we use instead a C program with global variables

These variables continue to exist between successive invocations of the program (like latches in sequential digital circuits)

PID Controller: the Program

$$S_{-1} = I_{-1} = 0.0$$
$$S_n = S_{n-1} + 0.1 \cdot I_n$$
$$O_n = 5.8 \cdot I_n + 4 \cdot S_n + 3.8 \cdot 10.0 \cdot (I_n - I_{n-1})$$

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O,J_1,S_1;

  I = Input();

  J_1 = I;
  S_1 = S + 0.1 * I;
  O = I * 5.8 + S_1 * 4.0 + 10.0 * 3.8 * (I-J);
  J = J_1;
  S = S_1;

  Output(O);
}
```

Optimizing the Program

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O,J_1,S_1;

  I = Input();

  J_1 = I;
  S_1 = S + 0.1 * I;
  O = I * 5.8 + S_1 * 4.0 + 10.0 * 3.8 * (I-J);
  J = J_1;
  S = S_1;

  Output(O);
}
```

optimization

$\longrightarrow$

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O;

  I = Input();

  S = S +0.1 * I * 4.0;
  O = I * 5.8 + S + 10.0 * 3.8 * (I-J);
  J = I;

  Output(O);
}
```

Automatic Code Generation

Saving two variables and two assignment statements is not much, but for complex control systems that should run on cheap micro-controllers such savings can be significant

Writing, modifying and optimizing such programs manually is error-prone and it would be much safer to derive it automatically from the high-level block diagram model

We have generated the program automatically using our Simulink-to-Lustre-to-C translator. From there is can be compiled to machine code

The Platform

The transformation to a working controller is not yet complete

The execution platform should support the I/O functions and be properly connected to the machinery for conversion between digital and analog data

Program correctness depends crucially on its being invoked every T time units, where T is the sampling period of the discrete time system used to derive the parameters of the controller

Not adhering to this sampling period may result in a strong deviation of the program behavior from the intended one

To ensure the correct periodic activation of the program we need access to a real-time clock that triggers the execution every T time units

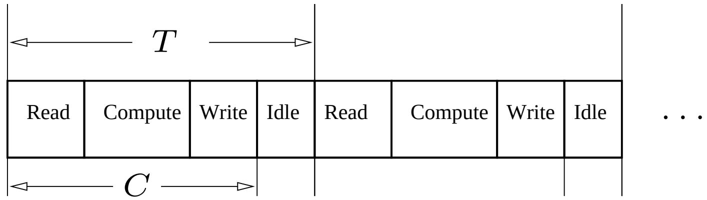
Computation Time

But this is not enough

An abstract mathematical function is timeless but a the corresponding program takes some time to compute

The condition $C < T$ should hold, where C is its worst case execution time (WCET). Otherwise the program will not terminate before its next invocation.

Computing WCET is not an easy task for modern processors



Final Implementation

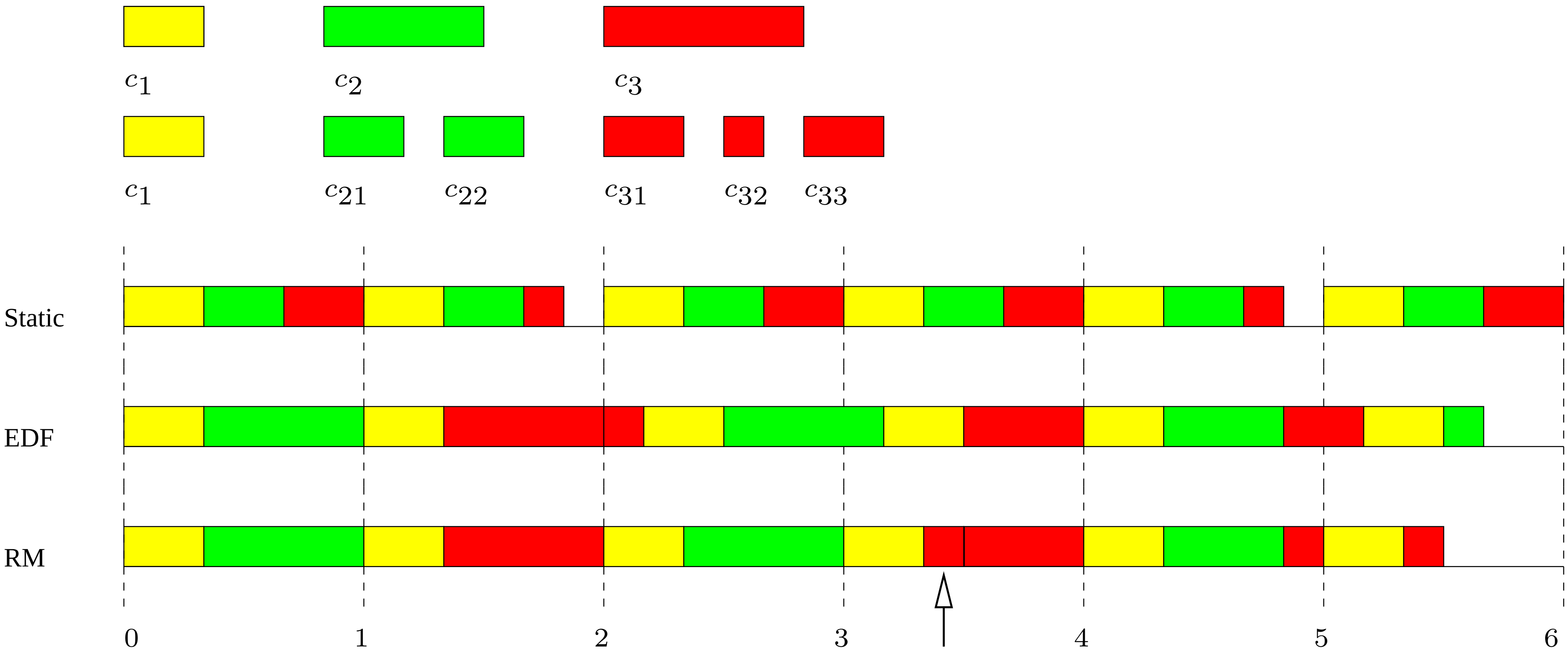
Historically, such controllers were first implemented on a bare machine, without using any operating system (OS)

The real-time clock acts as an interrupt that transfers control to the program. If the scheduling condition $C < T$ is satisfied, this interrupt occurs after the program has terminated and the computer is idle

No preemption or context switch. A simple and reliable solution that need not rely on a complex piece of software like an OS

Today real-time OS (RTOS) technology is more developed and the role of monitoring the real-time clock and dispatching the program for execution can be delegated to an OS

Preview: Multi-Periodic Controllers and Scheduling



Something Completely Different: Lustre and Temporal Logic

There are many formalisms for defining sets of sequences or functions from sequences to sequences

In general, any function $f : X \rightarrow Y$ can be "lifted" to a function $F : X^* \rightarrow Y^*$ or $F : X^\omega \rightarrow Y^\omega$

These are pointwise (instantaneous, memoryless) functions such that

$$\beta = F(\alpha) \text{ if } \forall t\ \beta[t] = f(\alpha[t])$$

Memory which is introduced through flip-flops and latches (sequential machines), states of automata, variables, etc. can be expressed by the delay operator

The Delay Operator (the pre of Lustre)

The function $D : X^\omega \to X^\omega$ is defined as

$$\beta = D(\alpha) \text{ iff } \forall t > 0 \; \beta[t] = \alpha[t-1]$$

$$\begin{array}{ll} \alpha : & 0110010001100\cdots \\ D(\alpha) : & *0110010001100\cdots \end{array}$$

This is equivalent to the previously operator of past temporal logic whose semantics is defined as:

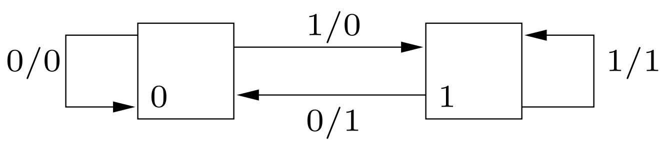
$$(\xi, t) \models \ominus\varphi \quad \leftrightarrow \quad (\xi, t-1) \models \varphi$$

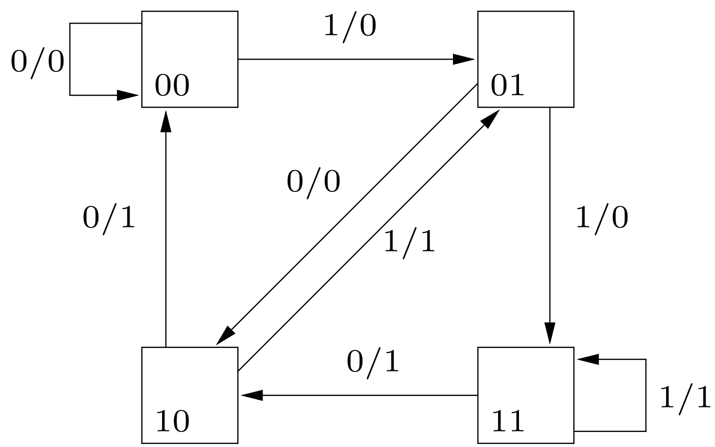
Automaton: the Shift Register

We can build an automaton (transducer) that for each input sequence α outputs $D(\alpha)$ or $D(D(\alpha))$ or $D^k(\alpha)$

This automaton is called a shift register. It remembers the last k inputs and outputs the value of the oldest among them

$k = 1$





$k = 2$

In temporal logic it can be viewed as a tester for $\ominus^k \varphi$: its input at time t indicates whether φ holds at t and its output says whether $\ominus^k \varphi$ holds at t

But what about the Future?

In future temporal logic you use the next operator whose semantics is

$$(\xi, t) \models \bigcirc\varphi \quad \leftrightarrow \quad (\xi, t+1) \models \varphi$$

Which corresponds to the inverse D^{-1} of D

$$\begin{array}{ll} \beta : & *0110010001100\cdots \\ D^{-1}(\beta) : & 0110010001100\cdots \end{array}$$

But this function is not causal! It has to output at time t something based on its input at time $t+1$

The Solution: Guessing and Aborting

We want to build an automaton which reads the sequence of truth values of φ and outputs the truth values of $\bigcirc\varphi$

This is part of a procedure to build automata from future TL formulae

The idea: guess and branch into two runs, one that predicts 0 and one that predicts 1

In the next step you abort the run that made the wrong prediction

For every infinite input sequence there is only one infinite output sequence

And the Automaton?

Just take the automaton for the delay and reverse the direction of the arrows

The past automaton was deterministic and complete and the future automaton is non-deterministic (guessing) and incomplete (abortion)

It is perhaps too late to speak of unbounded temporal operators or derive philosophical insights from the exercise, so let's conclude with an aphorism attributed to Niels Bohr:

Prediction is very difficult, especially about the future