Introduction to Embedded Systems

Edward A. Lee
Alberto Sangiovanni-Vincentelli

UC Berkeley
EECS 149/249A
Fall 2014

Lecture 2: Design Methodologies:
V-model, Model-Based Design, Platform Based Design
Challenges in Automotive Electronics Development

Increasing functionality:
- Safety (active/passive)
- Fuel efficiency (hybrid)
- Reduced emissions (less CO2)
- Comfort

Increasing quality:
- 2000: ~1000 - 10ppm (per ECU)
- 2010: ~1 - 0ppm (per ECU)

Increasing value:
- Electronic Share (value):
  - 2004: 20% -> 2015: 40%
- Software Share (value):
  - 2000: 4.5% -> 2010: 13%

Reduce time to market:
- 2000: ~ 20 – 26 months
- 2010: < 18-20 month
From federated to integrated architectures

Today: Federated Architectures

Each time a new function is required, the OEM starts a request to suppliers for a new ECU (an integrated HW/SW device realizing the function) to be integrated on the existing networks.

The device is developed by the supplier with its own choice of HW, RTOS, device drivers and communication layers (with some standardization).

The result is:
- Proliferation of ECUs (reaching 100)
- Complex distributed architectures with the need of high bandwidth and therefore multiple networks and gateways
- Complex functional and not-functional (timing) dependencies across the network, which OEMs struggle to control
- Missing opportunities for common set of libraries and (sub)functions
- Limited standardization, flexibility and extensibility
- Limited control on the execution platform by OEMs
From federated to integrated architectures

The execution architecture is completely selected and planned by the OEM. OEMs are free to standardize HW, drivers, RTOS and communication layers, leveraging competition among suppliers.

Each time a new function is required, the OEM starts a request to suppliers for new functional content (SW) to be integrated on the existing platform.

The challenges are:
- Moving from specifications of ECUs with message interfaces to the specs of SW components
- Standardize interoperability among components
- Standardize access to the platform services
- Define models that allow to predict the result of the composition (functional and not-functional)
How Did we Cope with Complexity?

Abstractions  Methodologies  Tools
(Freedom from Choice)
Automotive V-Models: a ‘Linear’ Development Process

Development of Car System

Development of Sub-System

Development of Mechanical Part(s)

ECU Development

ECU SW Development

ECU HW Development

ECU SW Implementation

Car System Sign-Off!

Sub-System(s) Integration, Test, and Validation

Sub-System Sign-Off!

ECU/ Sens./Actrs./Mech. Part(s) Integration, Calibration, and Test

ECU Sign-Off!

ECU HW/SW Integration and Test

ECU HW Sign-Off!

ECU SW Integration and Test

ECU SW

ECU HW

ECU: Electrical Control Unit
“Finding and fixing requirements errors consumes between 70% - 85% of total project rework costs.”
Platform Models for Model Based Development

Development of Distributed System

Distributed System Requirements

Distributed System Partitioning

Sub-Systems Model Based Development

Sub-Systems Requirements

Network Protocol Requirements

Sub-System(s) Sign-Off!

Network Communication Protocol Sign-Off!

Virtual Integration of Sub-System(s) w/ Network Protocol, Test, and Validation

Sub-System(s) Integration, Test, and Validation

Sub-System(s) Implementation Models Sign-Off!

Distributed System Sign-Off!
Platform Based Design

A “meet-in-middle” design method

- Platform: an abstraction layer that hides the details of several possible implementation refinements of the underlying layers

Function model:

- abstraction of what the system is supposed to do

Architecture model:

- lower level of abstraction describing how the system realizes the function

Mapping:

- Process by which function and architecture meet
- Propagates constraints from above to meet performance estimations from below
- Phases: Allocation, binding, scheduling
Separation of Concerns

Development Process

Analysis
Specification
Implementation

Performance Analysis
Refinement
Mapping
Evaluation of Architectural and Partitioning Alternatives

Behavior Components
Virtual Architectural Components

IPs

Behavior

f1
f2
f3

C-Code
Matlab
Dymola

CPUs
Buses
Operating Systems

ECU-1
ECU-2
ECU-3
Bus

EECS 149, UC Berkeley: 11
Platform-Based Design

Application Space

Architectural Space

Application Instance

Platform Instance

Platform Mapping

Platform Design-Space Export

- Platform: library of resources defining an abstraction layer with interfaces that identify legal connections
- Resources do contain virtual components i.e., placeholders that will be customized in the implementation phase to meet constraints
- Very important resources are interconnections and communication protocols
Fractal Nature of Design

Platform Instance
Platform Design-Space
Export

Function Space
Function Instance
Mapped
Platform (Architectural) Space
Platform Instance

Function Space
Function Instance
Mapped
Platform (Architectural) Space
Platform Instance

© Alberto Sangiovanni-Vincentelli. All rights reserved.
Design Methodology

Vehicle level
- Resources
- Topologies
- Solution Patterns
- Functional Networks
- Functions

Sub-system level
- Resources
- Domain Topologies
- Solution Patterns
- Functional Sub-Nets
- Functions

ECU node level
- Resources
- ECU Design
- Solution Patterns
- Functions on ECUs
- Functions

Silicon level
- Resources
- HW-Components
- Solution Patterns
- Functions (Physical Layer)
- Functions

Requirement Propagation and Platform Selection

Platform Constraints Propagation
Cyber-physical control system

Driver

Yaw/Slip controller

Brake controller

Engine controller

Wheel dynamics

Engine dynamics

Vehicle dynamics

ECU1

ECU2

ECU3

Assume execution parameters such as jitter, latency, accuracy, WCET

Assume load and utilization levels, resource usage

Network and computing elements
Summary

- Virtual engineering is taking over: instead of testing on design prototypes, virtual testing on mathematical models to decrease costs and improve quality
- Top-down linear processes are not effective. Verification comes too late in the process
- Design is a meet-in-the-middle process: physical properties of platforms must be considered
- The essential part of any design methodologies is MODELING!!
What is Modeling?

Developing insight about a system, process, or artifact through imitation.

A model is the artifact that imitates the system, process, or artifact of interest.

A mathematical model is model in the form of a set of definitions and mathematical formulas/objects.
What is Model-Based Design?

1. Create a **mathematical model** of all the parts of the embedded system
   - Physical world
   - Control system
   - Software environment
   - Hardware platform
   - Network
   - Sensors and actuators

2. Construct the implementation from the model
   - Goal: automate this construction, like a compiler
   - In practice, only portions are automatically constructed
The Key to Platform Based Design

- Components
- Composition rules
- Refinement rules
- Abstraction rules

Contracts
Introducing Contracts

Provide a methods to formalize requirements

- (semi-)formal – Patterns
  - Allows to apply analysis techniques
  - Reduce/avoid ambiguities
  - Supporting traceability

- Distinguish between
  - What is assumed from the environment → Assumptions
  - What must be guaranteed by the component → Promises
Component and Contracts

Enrich Components with Contracts

Assumption
- Determine boundary conditions on design context under which component is promising its services

Promise
- Provide guarantees if component is used in assumed design context

Contracts cover several aspects
- Assumptions and Promises are organized in viewpoints
  - Behaviour, Safety, Real-Time, Power ....
  - Shared information allows specification of cross viewpoint dependencies

Contracts specify the behavior at the interface of the component
- completely defines implementation space of components covering all viewpoints
- E.g. characterizes when component upgrades during product lifetime are permissible

A component with its contracts is called Rich Component
Modeling Techniques in this Course

Models that are abstractions of **system dynamics** (how things change over time)

Examples:
- Modeling physical phenomena – ODEs
- Feedback control systems – time-domain modeling
- Modeling modal behavior – FSMs, hybrid automata
- Modeling sensors and actuators – calibration, noise
- Modeling software – concurrency, real-time models
- Modeling networks – latencies, error rates, packet loss
Modeling of Continuous Dynamics

Ordinary differential equations, Laplace transforms, feedback control systems, stability analysis, robustness analysis, ...
An Example: Modeling Helicopter Dynamics

The Fundamental Parts of any Helicopter

©2000 HowStuffWorks
Modeling Physical Motion

Six degrees of freedom:
- Position: x, y, z
- Orientation: pitch, yaw, roll
Notation

Position is given by three functions:

\[ x : \mathbb{R} \to \mathbb{R} \]
\[ y : \mathbb{R} \to \mathbb{R} \]
\[ z : \mathbb{R} \to \mathbb{R} \]

where the domain  \( \mathbb{R} \) represents time and the co-domain (range)  \( \mathbb{R} \) represents position along the axis. Collecting into a vector:

\[ \mathbf{x} : \mathbb{R} \to \mathbb{R}^3 \]

Position at time  \( t \in \mathbb{R} \) is  \( \mathbf{x}(t) \in \mathbb{R}^3 \).
Notation

Velocity

\[ \dot{x} : \mathbb{R} \rightarrow \mathbb{R}^3 \]

is the derivative, \( \forall t \in \mathbb{R}, \)

\[ \dot{x}(t) = \frac{d}{dt} x(t) \]

Acceleration \( \ddot{x} : \mathbb{R} \rightarrow \mathbb{R}^3 \) is the second derivative,

\[ \ddot{x} = \frac{d^2}{dt^2} x \]

Force on an object is \( F : \mathbb{R} \rightarrow \mathbb{R}^3. \)
Newton’s Second Law

Newton’s second law states $\forall t \in \mathbb{R}$,

$$ F(t) = M \ddot{x}(t) $$

where $M$ is the mass. To account for initial position and velocity, convert this to an integral equation

$$ x(t) = x(0) + \int_{0}^{t} \dot{x}(\tau) d\tau $$

$$ = x(0) + t\dot{x}(0) + \frac{1}{M} \int_{0}^{t} \int_{0}^{\tau} F(\alpha) d\alpha d\tau, $$
Orientation

- Orientation: $\theta : \mathbb{R} \rightarrow \mathbb{R}^3$
- Angular velocity: $\dot{\theta} : \mathbb{R} \rightarrow \mathbb{R}^3$
- Angular acceleration: $\ddot{\theta} : \mathbb{R} \rightarrow \mathbb{R}^3$
- Torque: $T : \mathbb{R} \rightarrow \mathbb{R}^3$

\[
\theta(t) = \begin{bmatrix}
\theta_x(t) \\
\theta_y(t) \\
\theta_z(t)
\end{bmatrix} = \begin{bmatrix}
\text{roll} \\
\text{yaw} \\
\text{pitch}
\end{bmatrix}
\]
Angular version of force is torque. For a point mass rotating around a fixed axis:

- radius of the arm: \( r \in \mathbb{R} \)
- force orthogonal to arm: \( f \in \mathbb{R} \)
- mass of the object: \( m \in \mathbb{R} \)

\[
T_y(t) = rf(t)
\]

Just as force is a push or a pull, a torque is a twist.

Units: newton-meters/radian, Joules/radian

Note that radians are meters/meter (2\(\pi\) meters of circumference per 1 meter of radius), so as units, are optional.
Rotational Version of Newton’s Second Law

\[ T(t) = \frac{d}{dt} \left( I(t) \dot{\theta}(t) \right), \]

where \( I(t) \) is a 3 \times 3 matrix called the moment of inertia tensor.

\[
\begin{bmatrix}
T_x(t) \\
T_y(t) \\
T_z(t)
\end{bmatrix} = \frac{d}{dt} \begin{bmatrix}
I_{xx}(t) & I_{xy}(t) & I_{xz}(t) \\
I_{yx}(t) & I_{yy}(t) & I_{yz}(t) \\
I_{zx}(t) & I_{zy}(t) & I_{zz}(t)
\end{bmatrix} \begin{bmatrix}
\dot{\theta}_x(t) \\
\dot{\theta}_y(t) \\
\dot{\theta}_z(t)
\end{bmatrix}
\]

Here, for example, \( T_y(t) \) is the net torque around the \( y \) axis (which would cause changes in yaw), \( I_{yx}(t) \) is the inertia that determines how acceleration around the \( x \) axis is related to torque around the \( y \) axis.
Simple Example

Yaw dynamics:

\[ T_y(t) = I_{yy} \dot{\theta}_y(t) \]

To account for initial angular velocity, write as

\[ \dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau. \]
Feedback Control Problem

A helicopter without a tail rotor, like the one below, will spin uncontrollably due to the torque induced by friction in the rotor shaft.

Control system problem: Apply torque using the tail rotor to counterbalance the torque of the top rotor.
Actor Model of Systems

A system is a function that accepts an input signal and yields an output signal.

The domain and range of the system function are sets of signals, which themselves are functions.

Parameters may affect the definition of the function $S$. 

\[ x: \mathbb{R} \rightarrow \mathbb{R}, \quad y: \mathbb{R} \rightarrow \mathbb{R} \]

\[ S: X \rightarrow Y \]

\[ X = Y = (\mathbb{R} \rightarrow \mathbb{R}) \]
Actor model of the helicopter

Input is the net torque of the tail rotor and the top rotor. Output is the angular velocity around the $y$ axis.

Parameters of the model are shown in the box. The input and output relation is given by the equation to the right.

\[
\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau
\]
Composition of actor models

\[
x = T_y
\]

\[
y = x'
\]

\[
\forall t \in \mathbb{R}, \quad y(t) = ax(t) \quad y'(t) = i + \int_0^t x' (\tau) d\tau
\]

\[
y = ax
\]

\[
a = 1/I_{yy}
\]

\[
i = \dot{\theta}_y(0)
\]
Actor models with multiple inputs

\[
S: (\mathbb{R} \to \mathbb{R})^2 \to (\mathbb{R} \to \mathbb{R})
\]

\[
\forall t \in \mathbb{R}, \quad y(t) = x_1(t) + x_2(t)
\]

\[
(S(x_1, x_2))(t) = y(t) = x_1(t) - x_2(t)
\]
Proportional controller

\[ e(t) = \psi(t) - \dot{\theta}_y(t) \]

\[ T_y(t) = Ke(t) \]

\[ \dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau \]

\[ = \dot{\theta}_y(0) + \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau \]

Note that the angular velocity appears on both sides, so this equation is not trivial to solve.
Behavior of the controller

Desired angular velocity:

\[ \psi(t) = 0 \]

Simplifies differential equation to:

\[ \dot{\theta}_y(t) = \dot{\theta}_y(0) - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau \]

Which can be solved as follows (see textbook):

\[ \dot{\theta}_y(t) = \dot{\theta}_y(0) e^{-Kt/I_{yy}} u(t) \]
Exercise

Reformulate the helicopter model so that it has two inputs, the torque of the top rotor and the torque of the tail rotor.

Show (by simulation) that if the top rotor applies a constant torque, then our controller cannot keep the helicopter from rotating. Increasing the feedback gain, however, reduces the rate of rotation.

A better controller would include an integrator in the controller. Such controllers are studied in EECS 128.
Questions

- Can this controller be implemented in software? How does the behavior change when it is implemented in software?

- How do we measure the angular velocity?
Other Modeling Techniques we will talk about

- **State machines**
  - sequential decision logic

- **Synchronous/reactive concurrent composition**
  - concurrent computation
  - composes well with state machines

- **Dataflow models**
  - exploitable parallelism
  - well suited to signal processing

- **Discrete-event models**
  - explicit about time

- **Time-driven**
  - suitable for periodic, timed actions

- **Continuous-time models**
  - models of physical dynamics
  - extended to “hybrid systems” to embrace computation
Discretized Model
A Step Towards Software

Numerical integration techniques provided sophisticated ways to get from the continuous idealizations to computable algorithms. Discrete-time signal processing techniques offer the same sophisticated stability analysis as continuous-time methods.

But it is not accurate for software controllers (fails on correctness)

In general, \( z \) is an \( N \)-tuple, \( z = (z_1, \cdots, z_N) \), where \( z_i: \text{Reals}_+ \rightarrow \text{Reals} \). The derivative of an \( N \)-tuple is simply the \( N \)-tuple of derivatives, \( \dot{z} = (\dot{z}_1, \cdots, \dot{z}_N) \). We know from calculus that

\[
\dot{z}(t) = \frac{dz}{dt} = \lim_{\delta \to 0} \frac{z(t+\delta) - z(t)}{\delta},
\]

and so, if \( \delta > 0 \) is a small number, we can approximate this derivative by

\[
\dot{z}(t) \approx \frac{z(t+\delta) - z(t)}{\delta}.
\]

Using this for the derivative in the left-hand side of (5.50) we get

\[
z(t+\delta) - z(t) = \delta g(z(t), v(t)). \tag{5.51}
\]
Hybrid Systems – Union of Continuous & Discrete

A good starting point, but has limitations.

E.g. Consider building a hybrid system model for software running under a multitasking real-time OS.

This model gives two separate ordinary differential equations, one for each point mass attached to a spring. The ZeroCrossingDetector actor detects the collision of the point masses and emits the "touched" event.

V1 and V2 are velocities, and P1 and P2 are positions of the two masses.
Understanding models can be very challenging

An example, due to Jie Liu, has two controllers sharing a CPU under an RTOS. Under preemptive multitasking, only one can be made stable (depending on the relative priorities). Under non-preemptive multitasking, both can be made stable.

Theory for this is lacking, so designers resort to simulation and testing.
Key Concepts in Model-Based Design

- Models describe physical dynamics.
- Specifications are executable models.
- Models are composed to form designs.
- Models evolve during design.
- Deployed code may be (partially) generated from models.
- Modeling languages have semantics.
- Modeling languages themselves may be modeled (meta models)

For embedded systems, this is about
- Time
- Concurrency
- Dynamics
Model-based design: a quick assessment

Model-based design is used in industry but not to the extent that is desirable

- algorithms are designed and analyzed using block diagram-based modeling tools
- correctness of the algorithms is validated against models of the plant
- models form the basis for all subsequent development stages
  - executable specification (instead of docs)
  - automatic code generation

Advantages

- Time-saving and cost-effective
- Design choices can be explored and evaluated quickly and reliably
- Ideally, an optimized and fully tested system is obtained
Model-based design: Difficulties

However, today in industry

- model-based design is often limited to control algorithm description

- incomplete plant modeling prevents accurate validation of algorithms

- very expensive, time-consuming, bounded coverage
- due to the high cost, OEM will provide less support to experimentation in Tier-1 companies