Embedded Systems: Challenges and Views of Embedded Software Development

Cristina Seceleanu
Senior Lecturer

Mälardalen University
Mälardalen Real-Time Research Centre
Västerås, Sweden
cristina.seceleanu@mdh.se

PROGRESS
A national Swedish Strategic Research Centre
School of Innovation, Design and Engineering

Design Science
- Product Development
- Innovation
- Information Design

IPR
Innovation and Product Realization

ISS
Intelligent Sensor Systems

MRTC
Mälardalen Real-Time Research Centre

“Applied academic research in Industrial Software Engineering and Real-Time Embedded Systems with strong industrial links.”

-13 Professors
-20 Senior Researchers
-45 PhD students

Mälardalen University
Västerås, Sweden
www.mrtc.mdh.se

Industrial partners include:

Swedish Nat'l Embedded Software Systems Strategic Research Centre

ABB
ERICSSON
SCANIA
STOCKHOLM
VÄSTERÅS
ESKILSTUNA
What are Embedded Systems (ES)?

• ES are computer systems
  • designed to perform one or a few dedicated functions
  • are part of a complete device often including hardware and mechanical parts.
  • dedicated software (may be user-customizable)
  • controlled by one / more main processing cores
    - microcontroller or DSP
  • may require very powerful processors
  • are quite diverse; no one statement applies to all cases

• Embedded systems control many devices in common use today.
What’s Different About ES?

• ES have different design constraints than general purpose computers
  • cost may matter more than speed
  • long life cycle: dominates design decisions
  • reliability/safety: constrain design decisions
  • scarcer resources: constrain design decisions

• Applying general purpose digital CAD tools is not sufficient
  • system-level view must go beyond digital electronics
  • must deal with functional and extra-functional requirements
  • must optimize across a broad span of product life-cycle

• Business model varies by domain
  • desktop computing has few domains
    (PC, workstation, mini-supercomputer, entertainment)
## Typical Embedded System Functions

- **Control laws**  
  *(e.g., PID control, fuzzy control)*

- **Sequencing logic**  
  *(e.g., FSM, mode changes/switching between control laws)*

- **Signal processing**  
  *(e.g., voice, video)*

- **Fault response**  
  *(e.g., detection, reconfiguration)*

- **Application specific user interface hardware**  
  *(e.g., buttons, bells, lights)*
ES: Reactive Systems

- **Reactive systems** – maintain an ongoing interaction with their environments.
  - E.g. – air traffic control systems
    – ongoing processes within nuclear reactors, etc.

- Such systems must be specified in terms of their **behaviours**.

- **Challenge in controller construction**
  • possibly unexpected changes of inputs.
ES: Real-Time Systems

A system where correctness not only depends on the logical order of events but also on their **timing**!!

**E.g.:**
- Air Bags, Cruise Control, ABS
- Process Control, Production Lines, Robots
- Real-time Protocols
- DVD/CD Players
EMBEDDED SOFTWARE – WHY BOTHER?
The competitiveness of products is implemented in software.
Automotive

- In 5 years, vehicles will contain as much memory as a PC of today
- Complexity is increasing
- Development and maintenance costs are increasing

History repeats itself…

Source: Audi
Some more insights

- 90% of new innovations is realized by software
- 23% of total cost, estimated to increase to 35% in 2011
Telecommunication Evolution

**Ericsson:**
The World’s 5th largest Software Company

80% R&D related to Software

**SonyEricsson:**
Software doubled every 18 months
CHALLENGES IN EMBEDDED SOFTWARE DEVELOPMENT
Embedded SW challenge

Specific requirements (extra functional properties)

- Real-time / reactive operation
- Systems are resource constrained (space, power, computation, ...)
  - Small size, low power, limited cooling
- Real-time (sometimes very stringent)
- Dependability (sometimes safety-criticality, reliability)
- Moderate to extreme cost sensitivity

No uniform set of reqs. → Specialized domain specific methods are needed

Quality assurance at a reasonable effort is a constant challenge
- fast may be more important than precise
Safety & Security
Can open innovation be used in development of embedded systems?

- Can quality be guaranteed?
  - Can I trust components developed by "anyone"?

- Can confidentiality be guaranteed?
  - Can "anyone" develop my components without knowing my secrets?
Moores curse for software development
(our ability to handle SW does not increase as fast)

How to meet the SW complexity challenge

- Partitioning and integration
  - divide and conquer
- Abstraction
- Reuse
- Automation
- Standardisation

Moores law: Doubled computing capacity every 18th month

Methods and tools
(+architectures and platforms)
Fundamental Scientific Challenges

- Composability
  
  \[ P1 + P2 = P1+P2 \]

  - Answer: NO (typically)
    - Sometimes approximately true.
    - Tradeoff between the simplicity and accuracy.
      - Composability at the cost of reduced efficiency

- Encapsulation
  
  Software Component (proven correct in isolation)

  \[ \text{Sys1} \rightarrow \text{P1} \]

  P1 "correct"?

  \[ \text{Sys2} \rightarrow \text{P1} \]

  P1 still "correct"?

  Encapsulating firewall
  - Guaranteeing correctness
  - Allowing interactions
Observations of the practice of software engineering

• About 80% of software development deals with changing (adaptation, improvement) of existing software

• Time to market is an important competitive advantage:
  – Importance of incorporation of new innovations quickly

• System should be built to facilitate changes
  – Easy removal and addition of functionality

• Systems should be built to facilitate reuse
  – Easy integration of existing functions
Hypothesis

• By building embedded software (and systems) from reusable components
  – complexity,
  – integration, and
  – quality assurance

• can be handled in a more cost efficient and scalable way

Remains to be proven!
(at least for embedded software)
Approach: Component-Based Development

- Software built from larger reusable components

```c
int foo(int sensor)
{
    int i, j, result;
    i = 0; j = 0;
    while(i <= sensor)
    {
        if(j < 5)
            j++;
        else
            break;
        result = result + j - 2; i++;
    }
    return result;
}
```
Approach: Component-Based Development

- Software built from larger reusable components

```c
int foo(int sensor)
{
    int i, j, result;
    i = 0; j = 0;
    while (i <= sensor) {
        if (j < 5) {
            j++;
            if (j > maxval)
                break;
        }
        result = result + j - 2;
        i++;
    }
    return result;
}
```
Software Component Definition

Szyperski (Component Software beyond OO programming)

• A software component is
  – a unit of composition
  – with contractually specified interfaces
  – and explicit context dependencies only.

• A software component
  – can be deployed independently
  – it is subject to composition by third party.
ES Modeling

• Describing embedded system’s processing behavior
  – Can be extremely difficult
  • Complexity increasing with increasing capacity
    – Past: washing machines, small games, etc.
      » Hundreds of lines of code
    – Today: TV set-top boxes, Cell phone, nuclear plants etc.
      » Hundreds of thousands - millions of lines of code
  • Desired behavior often not fully understood in beginning
    – Many implementation bugs due to description mistakes/omissions
  – English (or other natural language) common starting point
    • Precise description difficult to impossible
    • Example: Motor Vehicle Code – thousands of pages
Example: trying to be precise in English

• California Vehicle Code
  – Right-of-way of crosswalks

• 21950. (a) The driver of a vehicle shall yield the right-of-way to a pedestrian crossing the roadway within any marked crosswalk or within any unmarked crosswalk at an intersection, except as otherwise provided in this chapter.
• (b) The provisions of this section shall not relieve a pedestrian from the duty of using due care for his or her safety. No pedestrian shall suddenly leave a curb or other place of safety and walk or run into the path of a vehicle which is so close as to constitute an immediate hazard. No pedestrian shall unnecessarily stop or delay traffic while in a marked or unmarked crosswalk.
• (c) The provisions of subdivision (b) shall not relieve a driver of a vehicle from the duty of exercising due care for the safety of any pedestrian within any marked crosswalk or within any unmarked crosswalk at an intersection.

  – All that just for crossing the street (and there’s much more)!
Models

• How can we capture behavior?

• Model
  – abstract representation of reality
  – facilitates the understanding of its workings

• System model
  – helps the analyst understand the functionality of the system
  – used to communicate with customers
  – if given semantics: are amenable to rigorous analysis
Models

- Different models present the system from different perspectives
  - **External** perspective: the system’s context / environment
  - **Behavioral** perspective: the overall system behavior
  - **Structural** perspective: the system’s architecture
Computation Models

• How can we precisely capture behavior?
  – *computation model* is the key

• Common computation models:
  – Sequential program model
    • Statements, rules for composing statements, semantics for executing them
  – Communicating process model
    • Multiple sequential programs running concurrently
  – State machine model
    • For control dominated systems, monitors control inputs, sets control outputs
  – Dataflow model
    • For data dominated systems, transforms input data streams into output streams
  – Object-oriented model
    • For breaking complex software into simpler, well-defined pieces
Models

• Two types of behavioral models

  – Data flow models:
    • show how data is processed as it moves through the system

  – State machine models
    • show the systems response to events

• Both of these models are required for a description of the system’s behavior
Formal Models

- **Formal** model
  - a *mathematical* model
  - a set of variables and
  - a set of logical and quantitative relationships between them.

- Formal models
  - are constructed to enable *reasoning* / *analysis*
    within an idealized *logical framework*, about systems.

- **Idealized**
  - the model may make explicit assumptions that are known to be false
    in some detail.
  - such assumptions
    - simplify the model while
    - allow the production of acceptably accurate solutions.
Models vs. languages

- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program

- Languages capture models
  - Concrete form, e.g., English, C

- Variety of languages can capture one model
  - E.g., sequential program model → C, C++, Java

- One language can capture variety of models
  - E.g., C++ → sequential program model, object-oriented model, state machine model

- Certain languages better at capturing certain computation models
Text versus Graphics

- Models versus languages not to be confused with text versus graphics
  - Text and graphics are just two types of languages
    - Text: letters, numbers
    - Graphics: circles, arrows (plus some letters, numbers)

\[
X = 1; \\
Y = X + 1;
\]

\[
X = 1 \\
Y = X + 1
\]
Introductory example: An elevator controller (EC)

- Simple elevator controller
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor

- Try capturing in C

**Partial English description**

“Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don’t change directions unless there are no higher requests when moving down or no lower requests when moving up…”
Sequential program model

Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

void UnitControl()
{
    up = down = 0; open = 1;
    while (1) {
        while (req == floor); ... req = ...
        if (req > floor) { up = ...
        } else {down = 1;}
        while (req != floor);
        up = down = 0;
        open = 1;
        delay(10);
    }
}

Partial English description

“Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don’t change directions unless there are no higher requests when moving up or no lower requests when moving down…”

System interface

One might have come up with something having even more if statements.
Finite-state machine (FSM) model of the EC

*UnitControl* process using a state machine

- **GoingUp**
  - req > floor
  - u, d, o, t = 1, 0, 0, 0
  - !(req > floor)

- **Idle**
  - req == floor
  - req < floor
  - u, d, o, t = 0, 0, 1, 0
  - timer < 10
  - !(timer < 10)

- **GoingDown**
  - req < floor
  - u, d, o, t = 0, 1, 0, 0
  - !(req < floor)

- **DoorOpen**
  - req < floor
  - u, d, o, t = 0, 0, 1, 1
  - timer < 10
  - !(req < floor)

Legend:
- u is up, d is down, o is open
- t is timer_start
Implementation

- Mapping of system’s functionality onto hardware platform:
  - captured using computational model(s)
  - written in some language(s)

- Implementation choice based on power, size, performance, timing and cost requirements

The choice of computational model(s) is based on whether it allows the designer to describe the system.

The choice of language(s) is based on whether it captures the computational model(s) used by the designer.

The choice of implementation is based on whether it meets power, size, performance and cost requirements.
Component-based vs. Model-based

- **Model-based** is focusing on abstraction
- **The model is the system**
- Automatic translation from model to system
- “Top down”

- **Component-based** is focusing on reuse
- “Bottom-up”
- Two independent processes
  - Development of components
  - Development of systems from components

**Complementary approaches**
- Components as design entities
  - Traditionally, SW components are binaries
ES Model Analysis

- Investigation conducted to provide stakeholders with information about the quality of the product or service

- Intended to discover software bugs

- Prerequisites
  - System model
  - System requirement
ES Model Analysis Techniques

- **Simulation**
  - manipulation of a model
  - compresses time and space
  - enables perceiving the interactions that would not otherwise be apparent
  - **Pro**: effective in discovering bugs in early stages
  - **Con**: does not cover all system behaviours
    » does not provide guarantees
ES Model Analysis Techniques

- Simulation

![Graph showing temperature, rod temperature, memory, and global clock over time.]
ES Model Analysis Techniques

- **Testing**
  - the process of executing a program or application with the intent of finding software bugs.

  - Software testing can also be stated as the process of validating and verifying that a software program/application/product:
    - meets the business and technical requirements that guided its design and development;
    - works as expected; and
    - can be implemented with the same characteristics.

  - Software testing
    - can be implemented at any time in the development process
    - most of the test effort occurs after the requirements have been defined and the coding process has been completed
    - never exhaustive for complex systems
ES Model Analysis Techniques

- **Formal Verification**
  - checking **ALL** system behaviours
  - exhaustive testing
  - property to check is given in some **logic**

- **Algorithmic Verification** (model checking)
  - exploration of the state space
  - scalability is still a challenge!

- **Deductive Verification**
  - uses axioms, rules to prove the correctness of a given model
  - theorem-proving
  - not fully automated (hindered application)
A – Model: Network of Timed Automata

F – Requirement: temporal logical formula, e.g.
- Invariant: something bad will never happen, something may happen
- Liveness: something will eventually happen
PROGRESS: Centre for Predictable ES Design

The PROGRESS approach

PROGRESS
A national Swedish Strategic Research Centre
“Basic values” (Philosophy/approach)

• Development of application SW for the vehicular, automation, and telecom domains

• First class citizens:
  – **Extra functional properties**
    • Timing, Resource usage, Reliability, and Life-cycle properties
  – **Reuse**
    • At multiple levels of abstraction/granularity
      – Small/Large SW components
      – Specifications (models) and analyses
  – **SW Design and Deployment**
    • Efficient development requires an interplay between these throughout the SW development (hypothesis)
**Goals:**

- Correlate a formal analysis with the deployment process
- **Analyze** functional and extra-functional behaviour for:
  - subsystems
  - components
  - their compositions
- **Assess** provided-required resource correctness (feasibility analysis), compute trade-offs
- **Optimal** resource utilization of components (good performance)

---

**PROGRESS ES Design View and Main goals**

http://www.idt.mdh.se/pride
High-level view

Component-based development

Legacy

Component model
IDE
Deployment
Predictability assurance

Legacy property prediction
Component-ization of legacy

Demonstrators

Progress
ES Analysis

- Establish correctness at design time of models

- Functional and timing properties of components
  - Formal model of functional and real-time behaviour
  - Verification of safety, liveness properties

- Difficulties/complexity:
  - hierarchical model
  - communication structure
  - functional behaviour, timing
• Software Component
  – Ports, read-execute-write
  – Function and timing
    • timed automaton with start/final location
    • ports mapped to data variables
    • analysis model

• Horizontal Composition
  • Connections

• Vertical Composition
  • Compositions
Final remarks

• Embedded systems development is not static
  – System complexity is sky-rocketing
  – Global competence is growing (and the centre of mass is moving)

• Opportunity for new players
• Threat for the established
  – IBM, Microsoft, Google, …
The applied (externally motivated) researcher’s opportunity and threat

- Problems are constantly evolving (moving target)
- Solutions are becoming obsolete
  - (and possibly: obsolete solutions can be recycled?)
- Assumptions must constantly be questioned
- Research must target future industrial reality (2020?)
- … better stick to basic research ;(-)
Those that master Software Intensive Embedded Systems are winners today... and tomorrow...

PROGRESS
A national Swedish Strategic Research Centre