What’s Ahead for Embedded Software?
- Edward A. Lee (2000)

Wednesday November 10, 2010
Hokeun Kim
## Contents

1. **Introduction**
2. **Frameworks**
3. **HW/SW Partnership**
4. **Real-time Scheduling**
5. **Interfaces and Types**
6. **Metaframework**
7. **Conclusion**
**Introduction**

- **Embedded Software**
  - Main task is to engage the physical world
  - Interact with sensors and actuator
  - Ex)

- **Research For Embedded Software**
  - Past days: too small and retro area
  - Nowadays: complex and pervasive enough
Introduction

Questions

- How to reconcile a set of domain-specific requirements with the demands of interaction in physical world?
- How to adapt software abstractions to meet requirements like these?
  - Real-time constraints
  - Concurrency
  - Stringent safety consideration
Frameworks

- **Component**
  - Any kind of building block
    - Ex) Program, process, subroutine, etc

- **Framework**
  - Defines models which governs the interaction
  - Supports component interaction mechanism
    - Ex) OS, Programming language, etc.

- **In other words, A Framework is**
  - A set of constraints on components
  - A set of benefits that derive from those constraints
Frameworks

Four service categories

- **Ontology**
  - What it means to be a component
  - Ex) subroutine, process, object

- **Epistemology**
  - State of knowledge
  - Ex) shared information, connectivity

- **Protocols**
  - How components interact
  - Ex) rendezvous, semaphores

- **Lexicon**
  - Vocabulary of component interaction
  - Ex) possible messages
Frameworks

- **Level of Constraints**
  - The more constraints, the more specificity
  - The more specificity, the more benefits
  - Examples
    - Unix pipe
      - Not support feedback structure, but no deadlock
    - The internet
      - Constraints on lexicon (byte stream), protocol (HTTP), but provides platform independence

- **KEY: To invent framework that better match the application domain**
  - Reintroduction of time
  - Recognize of essential properties when components become an aggregate
Frameworks

Concurrency

- Concurrency For Embedded System Design
  - Useful because some computations in parallel
  - But, concurrency complicates system design

- Von Neumann Framework
  - Sequential computation
  - Total ordered time -> Correctness

- Distributed System
  - Partially ordered at best
  - Difficult to maintain "Global system state"
Network Embedded Systems
- Various communication bandwidth and latencies
- Previous approaches (for synchronous digital circuit design) poorly suited

Prof. Gul Agha of UIUC
- Actors
  - Encapsulates control thread
  - Have interfaces with other actors
- Interaction patterns
  - Lack of structure
  - Interaction policy
So far, most designers are exposed to only one or two frameworks.

But, design practices are changed.
- Level of abstraction and domain-specificity rise.

Rich variety of frameworks.
- Myriad views being offered.
- Designers will need some way to reconcile those.

Example: different view of ‘Time’.
- Explicit – real number.
- Abstract – discrete.
- Constraints imposed by causality.
A grand unified approach to modeling
- Seek a framework that serves all purpose

Possible approaches
- Union of all the frameworks
- Choosing one concurrent framework
- ADL (Architecture description language)
- Heterogeneously mixed frameworks
<table>
<thead>
<tr>
<th>Frameworks</th>
<th>Mixing frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union</strong></td>
<td></td>
</tr>
<tr>
<td>• Complex and hard to use</td>
<td></td>
</tr>
<tr>
<td>• Designing would be difficult</td>
<td></td>
</tr>
<tr>
<td><strong>Choosing one</strong></td>
<td></td>
</tr>
<tr>
<td>• Feasible way relatively easy to use</td>
<td></td>
</tr>
<tr>
<td>• Doesn't acknowledge each model’s strength and weakness</td>
<td></td>
</tr>
<tr>
<td><strong>ADL</strong></td>
<td></td>
</tr>
<tr>
<td>• Describe the component interactions</td>
<td></td>
</tr>
<tr>
<td>• Provide a way to get good insight into the design</td>
<td></td>
</tr>
<tr>
<td>• Poor match could be appear</td>
<td></td>
</tr>
<tr>
<td><strong>Mixed</strong></td>
<td></td>
</tr>
<tr>
<td>• Preserve distinct identity of each framework</td>
<td></td>
</tr>
<tr>
<td>• Finite time machines + continuous time model</td>
<td></td>
</tr>
</tbody>
</table>
In history
- Functionality has steadily shifted from H/W to S/W

Software
- Primarily sequential execution
- Shared H/W resources multiplexed in time

Hardware
- Primarily parallel execution
- Hardly shared H/W resources

Most Embedded System Involve Both
- Designer’s task: To explore the balance between sequential and parallel styles
Hardware-Software Partnership

- **Hard real-time examples**
  - Signal processing
    - Concurrent tasks on distinct processors
  - Speech coders & Radio modems in cell phone
    - Use distinct processors
    - Primarily S/W, have H/W nature in that

- **In Theory**
  - Embedded processors’ improving performance
  - Until then designers must use
    - Dedicated H/W to handle hard-real-time tasks
    - Processors that so greatly exceed minimum performance
Hardware-Software Partnership

- **Real-time OSes cannot yet reliably handle hard-real-time tasks**
- **Rethink multitasking**
  - Component interface definitions need to declare temporal properties
    - Not just a fixed priority
    - Need to declare the dynamics (phases of execution, exception handling, etc...)
  - Composition of components must have consistent and non-conflicting temporal properties
    - Viewing all running processes as part of a single application
Real-time Scheduling

- **Real-time scheduler**
  - Provides some assurances of time performance given certain component properties
    - Ex) Component invocation periods, Task deadlines

- **Rate-monotonic scheduling**
  - Translate the invocation period into properties
  - Properties may be based on semantic information, reflecting the criticality
Real-time Scheduling

**Problem**
- Most methods are not compositional
  - Can provides assurance individually
  - There is no systematic way to provide assurance for the aggregate
  - For example, Priority inversion

**Priority inversion**
- A Phenomenon that low priority task is scheduled before high priority tasks
- Due to shared resource contention
- Possible way can make deeper failure
Real-time Scheduling

- Priority Inversion Example

- From Victor Giddings, Measuring Distributed and Local Priority Inversions in Real-Time ORBs

- Need for a entirely different scheduling mechanism
Interface and types

- **Type system**
  - Ensure software’s correctness
  - Provides a vocabulary for talking about larger structure

- **Type systems only handle static structure for embedded software**
  - Syntax of procedural programs
  - Nothing about concurrency or dynamics

- **Work with active object and actors moves a bit in the right direction**
  - Not enough about interface to safety, liveness, consistency, or real-time behavior
Interface and types
Type system techniques

❖ **Type system constraints**
  - What a component can say about its interface
  - How to ensure compatibility

❖ **How a type system works**
  - Data-level system type
    - Subtyping relation or lossless convertibility
  - System-level type
    - Dynamic properties using nondeterministic automata
Interface and types
How a type system works

Data-Level Type System

- General
- String
  - Boolean
  - Scalar
  - Complex
    - Long
    - Double
  - Integer
- NaT

Integer < Double
“Integer can be converted to Double without Information Loss”
Interface and types
How a type system works

- System-Level Type

  - Domain polymorphic
  - Process network
  - Discrete events
  - Rendezvous
  - Dataflow
  - Continuous time
  - NaT

Continuous Time < Dataflow
“Dataflow can simulate Continuous time”
Interface and types
The case for strong typing

- **Strong typing (Java, ML)**
  - Enables static type checking
  - Prone to dynamic errors
    - Array out of bound

- **Without strong typing (Lisp, Tcl)**
  - Modularity and reusability
  - Difficult to identify the source of problems
Interface and types
The case for strong typing

- One solution to obtain modularity and reusability with strong typing
  - Polymorphism
  - Reflection – observing and modifying structure and behavior at runtime
  - Runtime type inference and type checking

- Components must give their dynamic properties as part of interface definition
Meta framework

- **Stronger constraints**
  - Stronger benefits
    - Frameworks become rather specialized as they seek these benefits
  - Drawbacks of specialized frameworks
    - Cannot solve all the problems for any complex system

- **Not to give up the benefits,**
  - Mix frameworks heterogeneously
    - Through specialization (= subtyping)
    - Mix frameworks hierarchically
Meta framework

- **Ptolemy project @ UCB**
  - Hierarchical approach
  - Domain polymorphism
    - Domain – A framework using S/W infrastructure
    - Can operate in multiple domains
  - Initially,
    - Built in an ad hoc way, defining an interface as unspecific as possible
  - Recently,
    - Interfaces using nondeterministic automata
    - A component can operate within a domain if its interface automata simulate those of the domain
There are more problems

- Human-computer interaction
  - Transparent, mediating a natural and intuitive interaction with the physical world
- Configurable hardware
  - Selecting appropriate computational models
- Networking problem
  - Providing QOS guarantees in case of unreliable resources
- H/W & S/W design technique
  - Minimizing power consumption on portable devices
Conclusion

- **This paper focused on**
  - Constructing embedded S/W, since time became a first-class of programming

- **And the focus must move beyond**
  - A program’s functional correctness

- **The key problem in the future**
  - Identifying the appropriate abstractions for representing the design
Thank You!