A Component Model and Software Architecture for CPS

Abhishek Dubey, Gabor Karsai, Nagabhushan Mahadevan
ISIS/Vanderbilt University

NASA Cooperative Agreement NNX08AY49A
Outline

- Software components for real-time systems
- ARINC-653 features
- CCM features
- The ARINC Component Model
  - Components and interactions
  - Modeling and generation
  - Application: Software Health Management
  - Implementation
  - An Example
- Lessons Learned / Summary
Notional Design Flow for High-Confidence Software Systems

Requirement Specification

RA

Control Design

FD

Software Architecture

SwA

Component Design

CD

Arch Mod/Sim

Code Gen.

Verif.

 Alloc./Sched.

Analysis

HwA

HW Pwr/Perf Est

System Arch. Design

SY

Latency/RT Analysis

DPL

SW Deployment

What is a ‘component’?
Hard Real-time Components?

- Need:
  - A Component Model suitable for hard real-time systems that codifies all component interactions and allows specification of timing requirements

- Real-time CORBA?
  - QoS and scheduling attributes on CCM

- MARTE UML Profile?
  - Specifications for timing properties in UML models

- AUTOSAR?
  - Component execution model? (Only recently added).
ARINC-653/APEX: Partitioning Kernel API

- **Partitions:**
  - Spatial and Temporal separation of activities – Fault isolation!
  - Partition memory size and temporal duration are fixed

- **Within a partition (shared address space):**
  - Multiple processes (static); periodic/aperciodic, with opt deadline
  - Primitives for process interactions: buffers and blackboards, semaphores and events
  - Health monitor (to restart processes)

- **Across partitions (isolated address spaces):**
  - Fixed allotment of CPU time
  - Message-based interactions via channels connecting sampling and queuing ports

- **Multiple processors (‘modules’) – few details standardized**
CORBA Component Model

- **Components**
  - Generalized ‘objects’ with state
  - Synchronous (call/return) interactions via provided/required interfaces
  - Asynchronous (publish/subscribe) interactions via publish/subscribe interfaces

- **Component homes**
  - Lifecycle and resource management for components
ACM: The ARINC Component Model

- Provide a CCM-like layer on top of ARINC-653 abstractions
- Notional model:

  Terminology:
  - Synchronous: call/return
  - Asynchronous: publish-return/trigger-process
  - Periodic: time-triggered
  - Aperiodic: event-triggered
ACM: The ARINC Component Model

- Each ‘input interface’ has its own process
  - Process must obtain read-write/lock on component
- Asynchronous publisher (subscriber) interface:
  - Listener (publisher) process
  - Pushes (receives) one event (a struct), with a validity flag
  - Can be event-triggered or time-triggered (i.e. 4 variations)
- Synchronous provided (required) interface:
  - Handles incoming synchronous RMI calls
  - Forwards outgoing synchronous RMI calls
- Other interfaces:
  - State: to observe component state variables
  - Resource: to monitor resource usage
  - Trigger: to monitor execution timing
Components interact via asynchronous/event-triggered and synchronous/call-driven connections. Example: The Trigger component is released periodically and it publishes an event upon each activation. The GPS component subscribes to this event and is triggered sporadically to obtain GPS data from the receiver, and when ready it publishes its own output event. The Display component is triggered sporadically via this event and it uses a required interface to retrieve the position data from the GPS component.
### ACM: The ARINC Component Model

- Mapping the CCM concepts to APEX in ACM

<table>
<thead>
<tr>
<th>ACM: APEX Component Model</th>
<th>APEX</th>
<th>APEX Concept Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodic</td>
<td>Periodic process</td>
<td>Process start, stop Semaphores</td>
</tr>
<tr>
<td>Sporadic</td>
<td>Aperiodic process</td>
<td></td>
</tr>
<tr>
<td>Invocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous Call-Return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodic Target</td>
<td>Co-located</td>
<td>N/A</td>
</tr>
<tr>
<td>Non-co-located</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Sporadic Target</td>
<td>Co-located</td>
<td>Caller method signals callee to release then waits for callee until completion.</td>
</tr>
<tr>
<td>Non-co-located</td>
<td>Caller method sends RMI (via CM) to release callee then waits for RMI to complete.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TCP/IP, Semaphore, Event</td>
<td></td>
</tr>
<tr>
<td>Asynchronous Publish-Subscribe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodic Target</td>
<td>Co-located</td>
<td>Callee is periodically triggered and polls ‘event buffer’ – validity flag indicates whether data is stale or fresh</td>
</tr>
<tr>
<td>Non-co-located</td>
<td>Blackboard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sampling port, Channel</td>
<td></td>
</tr>
<tr>
<td>Sporadic Target</td>
<td>Co-located</td>
<td>Callee is released when event is available</td>
</tr>
<tr>
<td>Non-co-located</td>
<td>Blackboard, Semaphore, Event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Queuing port, Semaphore, Event</td>
<td></td>
</tr>
</tbody>
</table>

- **Observe:**
  - All component interactions are realized via the framework
  - Process (method) execution time has deadline, which is monitored
Modeling Language

- **Modeling elements:**
  - Data types: primitive, structs, vectors
  - Interfaces: methods with arguments

- **Components:**
  - Publish/Subscribe ports (with data type)
  - Provided/Required interfaces (with i/f type)
  - Health Manager

- **Assemblies**

- **Deployment**
  - Modules, Partitions
  - Component → Partition
Modeling

- Needs for analysis: component internals + assembly
  - Component internal data- and control flows
  - Component Assembly Model
Background

- Project on Model-based Software Health Management
  - How to build ‘software health management functions’ into systems that monitor, diagnose, and mitigate software defects at run-time?

- Concept
  - Use model-based fault diagnostics techniques for monitoring and diagnosis
  - Use model-based software development techniques to design, analyze, and generate the code for the software health management function
Modeling Language: Monitoring

- Monitoring on component interfaces
  - Subscriber port → ‘Subscriber process’ and Publisher port → ‘Publisher process’
    - Monitor: pre-conditions and post-conditions
    - On subscriber: Data validity (‘age’ of data)
    - Deadline (hard / soft)
  - Provided interface → ‘Provider methods’ and Required interface → ‘Required methods’
    - Monitor: pre-conditions and post-conditions
    - Deadline (hard / soft)
  - Can be specified on a per-component basis
- Monitoring language:
  - Simple, named expressions over input (output) parameters, component state, `delta(var)`, and `rate(var,dt)`. The expression yields a Boolean condition.
Modeling Language: Component Health Manager

- **Reactive State Machine**
  - *Event* trigger:
    - Predefined condition (e.g. deadline violation, data validity validation)
    - User-defined condition (e.g. pre-condition violation)
  - Reaction: mitigation *action* (start, reset, refuse, ignore, etc.)
  - *State*: current state of the machine
  - (Event \( \times \) State) \( \rightarrow \) Action
ACM: A Prototype Implementation

- **ARINC-653 Emulator**
  - Emulates APEX services using Linux API-s
  - Partition → Process, Process → Thread
  - Module manager: schedules partition set
  - Partition level scheduler: schedules threads within partition

- **CORBA foundation**
  - **MICO CCM ORB**
  - No modifications
  - CLHM: Component-level Health Manager
ACM: A Prototype Implementation

- **Platform:**
  - ARINC-653 Emulator on Linux
  - MICO (open source CORBA)
  - Module manager, infrastructure

- **Code generator**
  - Produces ‘glue code’ for the component framework
  - Compiles monitoring expressions
  - Builds code for CHM

Designer supplies functional code
ACM: Model-based Development

- Graphical models are used to generate ‘infrastructure’ code
Example: Fault Detection and Mitigation scenarios

<table>
<thead>
<tr>
<th>Fault</th>
<th>Detected at</th>
<th>Fault source</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard deadline violation</td>
<td>GPS Trigger interface</td>
<td>GPS Component</td>
<td>Stop and restart</td>
</tr>
<tr>
<td>Stale data (missing update)</td>
<td>NAVDisplay Subscribe port</td>
<td>GPS Component</td>
<td>Use previous value</td>
</tr>
<tr>
<td>Missing sensor event</td>
<td>GPS Subscribe port</td>
<td>Sensor Component</td>
<td>Use previous value</td>
</tr>
<tr>
<td>Rate of change is too high</td>
<td>NAVDisplay required interface</td>
<td>GPS Component</td>
<td>Use previous value</td>
</tr>
</tbody>
</table>
Lessons Learned / Summary

- Two worlds: The highly dynamic CCM and the strictly static ARINC do not mesh well
- Allocating a thread to every method is possibly a waste of resources
- For analyzability a deeper modeling of component structure and behavior is needed

- ACM: Steps towards a hard real-time component model
  - CCM: provides the essential component abstraction
  - ARINC: provides the API / platform
- Model-based configuration and code generation helps
- ACM is an experiment – work in progress