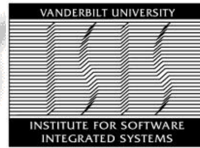


INTEGRATED VEHICLE
HEALTH MANAGEMENT

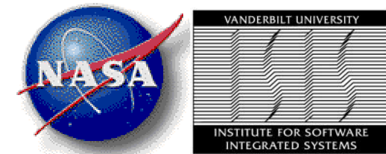
Model-based Software Health Management

A. Dubey, G. Karsai, R. Kereskenyi, N. Mahadevan
Institute for Software-Integrated Systems
Vanderbilt University

NASA Cooperative Agreement NNX08AY49A
NASA POC: Paul Miner

Aviation Safety Program Technical Conference
November 17-19, 2009
Washington D.C.

Outline



- Motivation
- Problem Statement
- Background
- IVHM milestones(s) being addressed
- Approach
- Results
- Conclusions
- Future Plans

Level 4 – *Aircraft Level*

Goal – Validated multidisciplinary integrated vehicle health management tools and techniques to enable automated detection, diagnosis, prognosis and mitigation of adverse events during flight.

IVHM 4.1 Vehicle-Level Reasoning and Ground/ Flight Test Evaluations

IVHM 4.2 Systems Analysis

IVHM 4.3 *Dashlink*

IVHM 4.4 Research Test and Integration

Level 3 – *Themes*

IVHM 3.1 Detection

IVHM 3.2 Diagnosis

IVHM 3.3 Prognosis

IVHM 3.4 Mitigation

IVHM 3.5 Integrity Assurance

Level 2 – *Subsystems*

IVHM 2.1 Aircraft Systems HM

IVHM 2.2 Airframe HM

IVHM 2.3 Propulsion HM

IVHM 2.4 Software HM

Level 1 – *Foundational*

IVHM 1.1 Advanced Sensors and Materials

IVHM 1.2 Modeling

IVHM 1.3 Advanced Analytics and Complex Systems

IVHM 1.4 Verification and Validation

Motivation: Software as Failure Source?

Qantas 72 - Oct 7, 2008 – A330 (Australia) – ATSB Report

At 1240:28, while the aircraft was cruising at 37,000 ft, the autopilot disconnected. From about the same time there were various aircraft system failure indications. At 1242:27, while the crew was evaluating the situation, the aircraft *abruptly pitched nose-down*. The aircraft reached a maximum pitch angle of about 8.4 degrees nose-down, and descended 650 ft during the event. After returning the aircraft to 37,000 ft, the crew commenced actions to deal with multiple failure messages. At 1245:08, the aircraft commenced *a second uncommanded pitch-down event*. The aircraft reached a maximum pitch angle of about 3.5 degrees nose-down, and descended about 400 ft during this second event. At 1249, the crew made a PAN urgency broadcast to air traffic control, and requested a clearance to divert to and track direct to Learmonth. At 1254, after receiving advice from the cabin of several serious injuries, the crew declared a MAYDAY. The aircraft subsequently landed at Learmonth at 1350.

The investigation to date has identified two significant safety factors related to the pitch-down movements. Firstly, immediately prior to the autopilot disconnect, **one of the air data inertial reference units (ADIRUs) started providing erroneous data (spikes) on many parameters to other aircraft systems**. The other two ADIRUs continued to function correctly. Secondly, **some of the spikes in angle of attack data were not filtered by the flight control computers**, and the computers subsequently commanded the pitch-down movements.

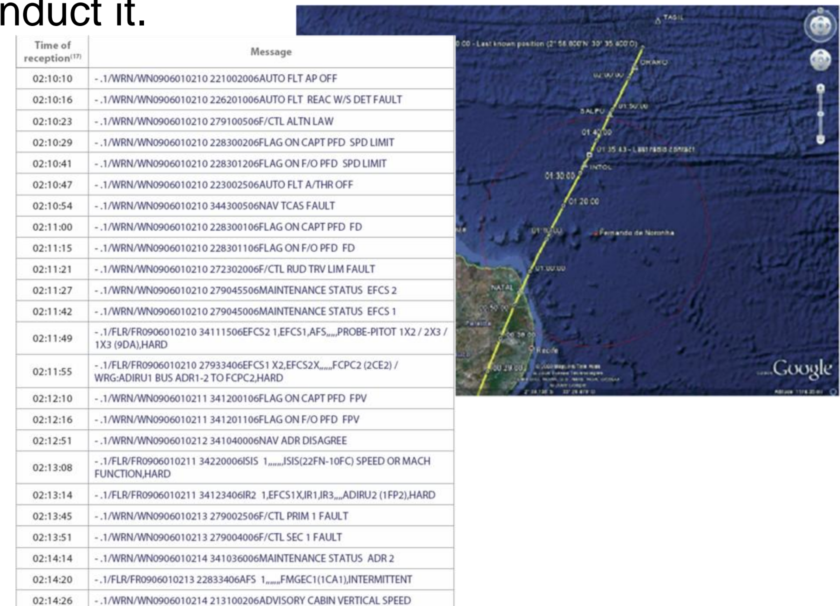
http://www.atsb.gov.au/publications/investigation_reports/2008/AAIR/pdf/AO2008070_interim.pdf

Motivation: Software as Failure Source?

AF 447 – June 1, 2009 – A330 (Atlantic Ocean) – BEA Interim Report

On Monday 1st June 2009 at around 7 h 45, the BEA was alerted by the Air France Operations Coordination Centre, which had received no news from flight AF447 between Rio de Janeiro Galeão (Brazil) and Paris Charles de Gaulle. After having established without doubt that *the airplane had disappeared in international waters* and in accordance with Annex 13 to the Convention on International Civil Aviation and to the French Civil Aviation Code (Book VII), the BEA launched a technical investigation and a team was formed to conduct it.

- ACARS messages indicate:
  - autopilot disconnect…
  - reconfiguration to alternate control law…
  - autothrust disconnect…
  - inconsistent ADR data…
  - cabin vertical speed
- History of speed sensor problems



Time of reception[17]	Message
02:10:10	- .1/WRN/WN0906010210 221002006AUTO FLT AP OFF
02:10:16	- .1/WRN/WN0906010210 226201006AUTO FLT REAC W/S DET FAULT
02:10:23	- .1/WRN/WN0906010210 279100506F/CTL ALTN LAW
02:10:29	- .1/WRN/WN0906010210 228300206FLAG ON CAPT PFD SPD LIMIT
02:10:41	- .1/WRN/WN0906010210 228301206FLAG ON F/O PFD SPD LIMIT
02:10:47	- .1/WRN/WN0906010210 223002506AUTO FLT A/THR OFF
02:10:54	- .1/WRN/WN0906010210 344300506NAV TCAS FAULT
02:11:00	- .1/WRN/WN0906010210 228300106FLAG ON CAPT PFD FD
02:11:15	- .1/WRN/WN0906010210 228301106FLAG ON F/O PFD FD
02:11:21	- .1/WRN/WN0906010210 272302006F/CTL RUD TRV LIM FAULT
02:11:27	- .1/WRN/WN0906010210 279045506MAINTENANCE STATUS EFCS 2
02:11:42	- .1/WRN/WN0906010210 279045006MAINTENANCE STATUS EFCS 1
02:11:49	- .1/FLR/FR0906010210 34111506EFCS2 1,EFCS1,AFS,,,,,,PROBE-PITOT 1X2 / 2X3 / 1X3 (9DA),HARD
02:11:55	- .1/FLR/FR0906010210 27933406EFCS1 X2,EFCS2X,,,,,FCPC2 (2CE2) / WRG:ADIRU1 BUS ADR1-2 TO FCPC2,HARD
02:12:10	- .1/WRN/WN0906010211 341200106FLAG ON CAPT PFD FPV
02:12:16	- .1/WRN/WN0906010211 341201106FLAG ON F/O PFD FPV
02:12:51	- .1/WRN/WN0906010212 341040006NAV ADR DISAGREE
02:13:08	- .1/FLR/FR0906010211 34220006ISIS 1,,,,,,ISIS(22FN-10FC) SPEED OR MACH FUNCTION,HARD
02:13:14	- .1/FLR/FR0906010211 34123406IR2 1,EFCS1X,IR1,IR3,,,,ADIRU2 (1FP2),HARD
02:13:45	- .1/WRN/WN0906010213 279002506F/CTL PRIM 1 FAULT
02:13:51	- .1/WRN/WN0906010213 279004006F/CTL SEC 1 FAULT
02:14:14	- .1/WRN/WN0906010214 341036006MAINTENANCE STATUS ADR 2
02:14:20	- .1/FLR/FR0906010213 22833406AFS 1,,,,,,FMGEC1(1CA1),INTERMITTENT
02:14:26	- .1/WRN/WN0906010214 213100206ADVISORY CABIN VERTICAL SPEED

"*Visual examination showed that the airplane was not destroyed in flight; it appears to have struck the surface of the sea in level flight with high vertical acceleration*"

http://www.bea.aero/docspa/2009/f-cp090601e1.en/pdf/f-cp090601e1.en.pdf

Problem Statement

Embedded software is a complex engineering artifact that can have latent faults, uncaught by testing and verification. Such faults become apparent during operation when unforeseen modes and/or (system) faults appear.

The problem:

- General: How to construct a *Software Health Management* system that detects such faults, isolates their source/s, prognosticates their progression, and takes mitigation actions in the *system* context?
- Specific: How to specify, design, and implement such a system using a *model-based framework*?

The larger picture:

- General: Software Health Management must be integrated with System Health Management – 'Software Health Effects' must be understood on the Vehicle Level.
- Specific: Vehicle-Level Reasoning System: New start project that builds a Vehicle-level HM technology – participation on the Boeing team.

Background

Software Fault Tolerance: Methods and techniques to implement software that can tolerate faults in itself, in the platform it is running on, in the hardware system it is connected to, and in the environment.

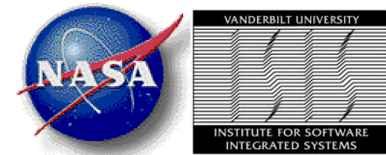
- Extends (hardware-based) fault-tolerance techniques to software
- Literature:
  - Wilfredo Torres-Pomales: Software Fault Tolerance: A Tutorial, NASA/TM-2000-210616, Langley Research Center, 2000
  - Software Fault Tolerance, Edited by Michael R. Lyu, Published by John Wiley & Sons Ltd.
- Single version techniques:
  - Component self-protection and self-checking
  - On-line checks: replication, timing, reversal, coding, reasonableness, structural…
  - Checkpointing, pair-wise operation
- Multi-version techniques:
  - Recovery blocks, n-version programming, n-self-checking, consensus-based…
- Limitations:
  - Reactive behavior + Hand-crafted construction + Limited integration with ‘system’

Background

Why 'Software Health Management' and why now?

- Complexity of systems necessitates an additional layer 'above' SFT that manages the 'Software Health'
- Embedded software ….
  - is a crucial ingredient in aerospace systems
  - is a method for implementing functionality
  - is the 'universal system integrator'
  - could exhibit faults that lead to system failures
  - complexity has progressed to the point that zero-defect systems (containing both hardware and software) are very difficult to build
- Systems Health Management is an emerging field that addresses precisely this problem: How to manage systems' health in case of faults ?
- 'Software Health Management' is **not**…
  - A replacement for existing and robust engineering processes and standards (DO-178B)
  - A substitute for hardware- and software fault tolerance
  - An 'ultimate' solution for fault tolerance

IVHM milestones(s) being worked



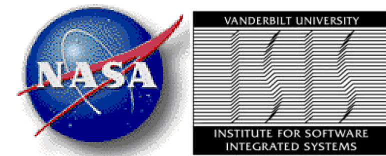
Model-based Software Health Management: Model-based techniques and tools for detecting, diagnosing, prognosticating and mitigating software-related faults.

- The work is directly related to the topic "IVHM 2.4 Software Health Management" of the IVHM Technology Plan.
  - "*The goal of the Software Health Management element is to develop the tools and techniques needed to enable the detection, diagnosis, prognosis, and mitigation of errors and related adverse events caused or contributed to by software systems in aircraft*."
- Related IVHM milestones:

Technology Level/Fiscal Year	08	09	10	11	12
Software HM	Initiate SoA Survey	Consistent Evidence Accum. Framework	S/W Malfunct. Classification		Eval. of Integrated Adapt. Reconfig.

- Award timeline: 10/1/08-9/31/11 (3 years), currently starting Year 2

Approach



- Summary:

  Build tools and techniques for model-based, generated health manager/s for software components and systems.

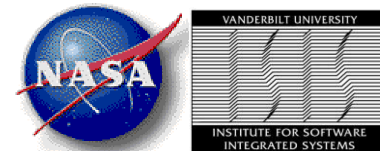
Year	Focus of effort:	Milestones:	Deadline (relative to year)
1	Literature review, concept development, component-level health management	Literature review	Y1+6mos
		Implementation plan	Y1+6mos
		Component-level health manager	Y1+10mos
		Testbed, experiment, demonstration	Y1+11mos
2	Software health management for cascading faults	Modeling language	Y2+3mos
		System-level health manager (v1)	Y2+10mos
		Testbed, experiment, demonstration	Y2+11mos
3	Software health management for systemic faults	Modeling goal trees	Y3+3mos
		System-level health manager (v2)	Y3+10mos
		Experiment/demonstration	Y3+11mos

Phase I: Literature Review

Main conclusions:

- **Faults in systems and** in **software** are well recognized and both detection and mitigation techniques are available.
- For SHM, the most **relevant** HM/FDIR concepts and techniques are: run-time fault detection, diagnosis, and containment and mitigation.
- Fault detection for software is **difficult**, because it is often **hard to define** what the **correct** behavior is. Hence, techniques are needed for modeling the correct behavior of components and subsystems, under all foreseeable scenarios.
- Fault diagnosis in the classical sense ('fault source isolation') may be problematic, as it is **hard to foresee the exact faults** in software. It may be more *pragmatic* to indict solely the faulty software component, with some model of how it has failed.
- **Fault containment** techniques could be used to provide the primary protection from failures that can possibly propagate into high-criticality components, and such techniques are needed for **protecting** the SHM system as well.
- Fault mitigation and recovery should be placed on a more **systematic** and **formal** basis such that faults and failures are anticipated in the software development process, and **appropriate verifiable mitigation actions** are designed into the system. One can take advantage of the existing results; however a general, reusable framework for the approach is still a research challenge.

Phase I: Implementation Plan

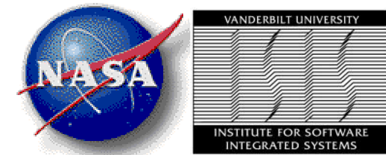


- Phase I – Implementation Plan
  - Use a component-based approach: software components and/or component containers as the *primary* units as SHM
  - 1. Select platform:
    - Representative OS: – ARINC-653 / IMA emulated on Linux
    - Component System: Lightweight Component Framework Implementation
  - 2. Prototype technology:
    - *Monitoring* on the level of components
    - Monitors detect *discrepancies*
    - Health manager reacts with *mitigation actions*
  - 3. Demonstrate technology on selected examples
    - Small component-based system
    - Small UAV flight software – two controllers
      - Available from VU/CMU/UCB MURI project

Prototype Technology

- Phase I – Implementation
  - Technology approach: Component-level health management
    - *Monitoring* the component
      - Interfaces (synchronous/asynchronous calls)
      - Component state
      - Scheduling and timing (WCET)
      - Resource usage
    - *Detection*:
      - Pre/post conditions over call parameters, rates, and component state
      - Conditions over timing properties
      - Conditions over resource usage (e.g. memory footprint)
      - Combinations of the above
    - *Mitigation*:
      - Given detected anomaly and state of the component take action
      - Can be time- or event-triggered
      - Actions: restart, initialize, block call, inject value, inject call, release resource, modify state; combination of the above

Notional Component Model



Parameter

Monitor arriving events

Monitor published events

Subscribe (Event)

Publish (Event)

Component

Provided (Interface)

Required (Interface)

Monitor incoming calls

Monitor outgoing calls

Monitor control flow/ triggering

Observe state

State

Resource

Trigger

Monitor resource usage

A component is a unit (containing potentially many objects). The component is *parameterized*, has **state**, it consumes **resources**, publishes and subscribes to **events**, provides interfaces and requires interfaces from other components.

Publish/Subscribe: Event-driven, asynchronous communication (publisher does not wait)

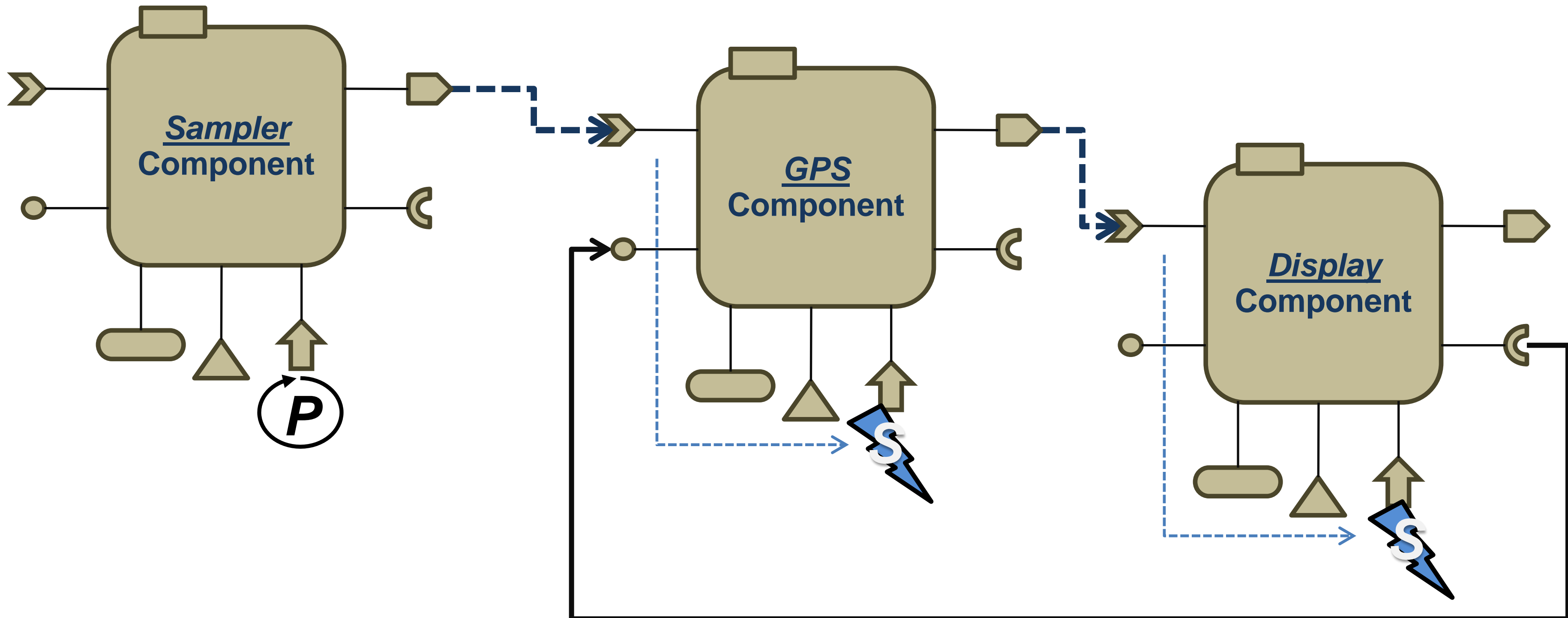
Required/Provided: Synchronous communication using call/return semantics.

Triggering can be periodic or sporadic.

Extension of a Component Model defined by OMG (CCM) : state, resource, trigger interfaces.

The CM allows monitoring of various properties of the software component, at run-time.

Example: Component Interactions

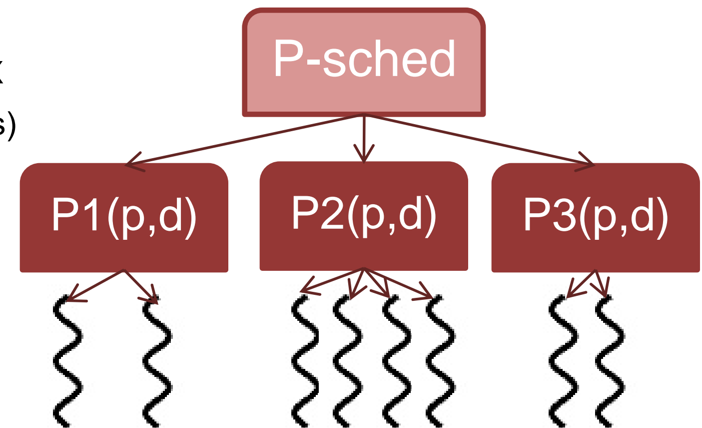


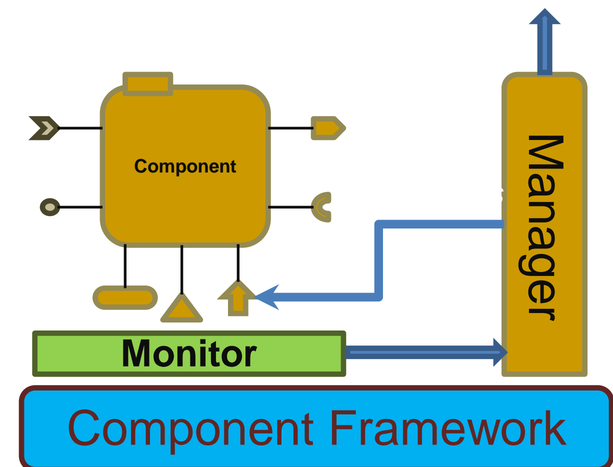
Components can interact via asynchronous/event-triggered and synchronous/call-driven connections.

Example: The *Sampler* component is triggered 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.

Component-level Health Management: 1/2

- Phase I – Component-level Health Manager
  - Platform:
    - ARINC-653-like RTOS emulated on Linux
      - Linux process → 'Partition' (controlled via signals)
      - Linux thread → 'Process' (monitored exec time)
    - Lightweight **Component Framework**
      - Open source CCM (MICO)
      - Modified to work on SW emulator
  - Monitoring:
    - On interface → CCM "interceptors"
    - On state → Required 'state access' methods
    - Timing → thread monitoring
    - Resources → API calls
  - Mitigation:
    - Given detection event (condition, state, timing, resource usage)
    - Given time since last event
    - → Take mitigation action





→ Independently designed, implemented, and verified.

Component-level Health Management: 2/2

Manager's behavioral model:

- Finite-state machine
- Triggers: monitored events, time
- Actions: mitigation activities

Manager is local to component container (for efficiency) but shall be protected from the faults of functional components

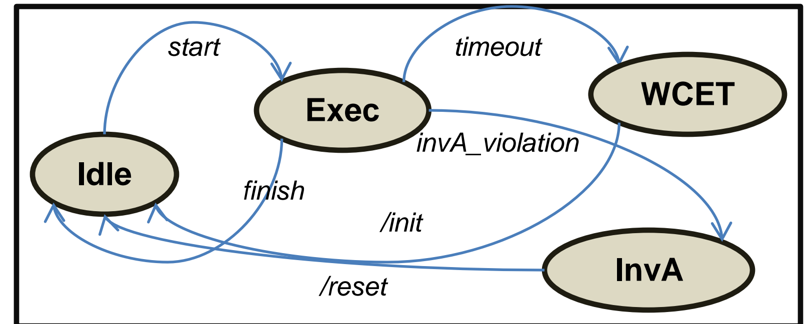
Notional behavior:

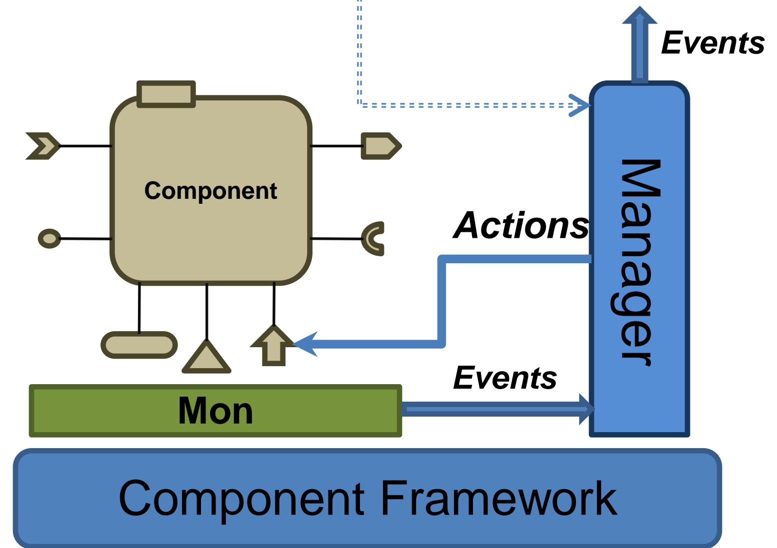
Track component state changes via detected events and progression of time

Take mitigation actions as needed

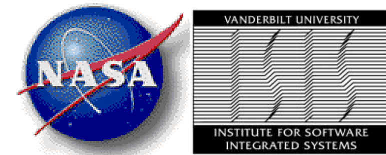
Design issues:

- Co-location with component (fault containment)
- Local detection may implicate another component

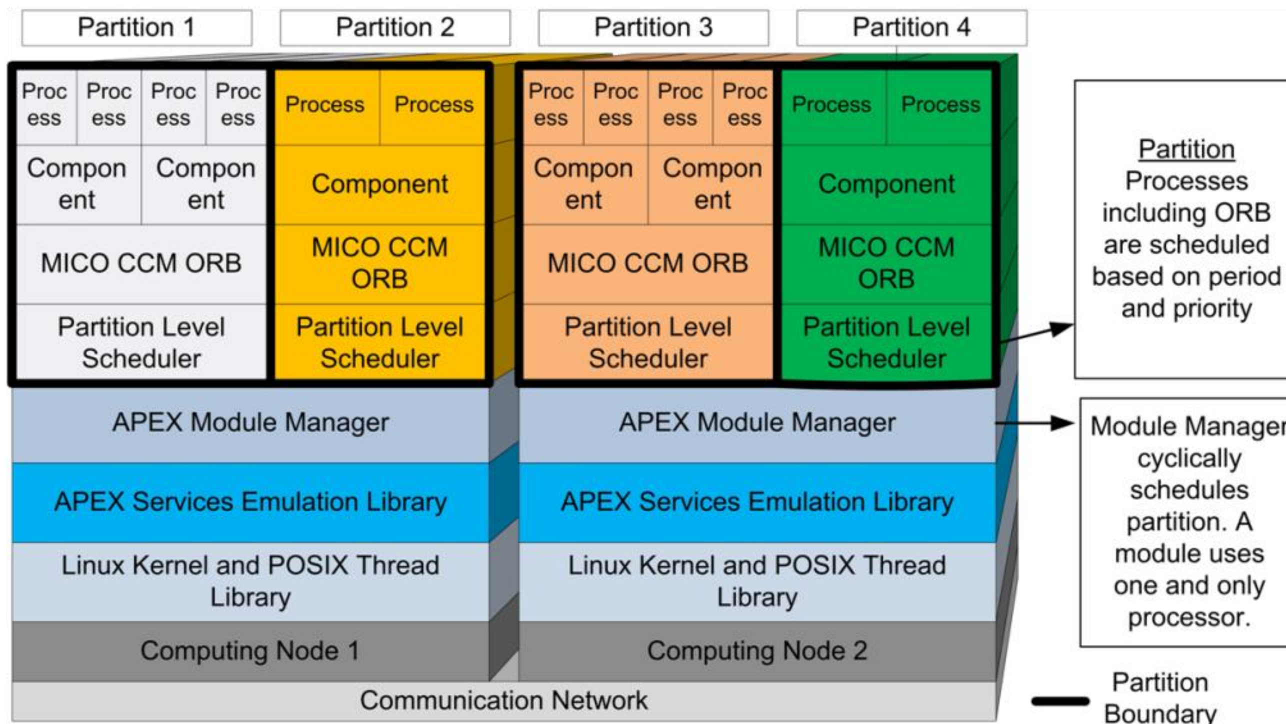




Results



- Implementation platform for experimentation
  - ARINC-653 emulator on Linux: Partial implementation of the APEX calls
  - Lightweight Component Framework: MICO/CCM
    - Minor changes in the source code
    - Periodic and sporadic methods on components
    - Extensions to the IDL-generated code to enforce execution time monitoring



Results

Phase I – Component-level Health Manager - Status

- Platform:
  - Prototype is operational on selected test examples
  - Able to detect WCET violations, evaluate pre- and post-conditions on interface calls
  - Examples use hand-coded conditions and mitigation actions
- Monitoring:
  - Specification language is in progress – will be embedded in an architecture modeling language
  - Changes to IDL generator for CCM is prototyped
- Mitigation:
  - Modeling (specification) language is in progress – will be integrated with the component framework
  - Code generator is in progress

Results – Lessons learned about CM + RTOS

Features of CM and RTOS:

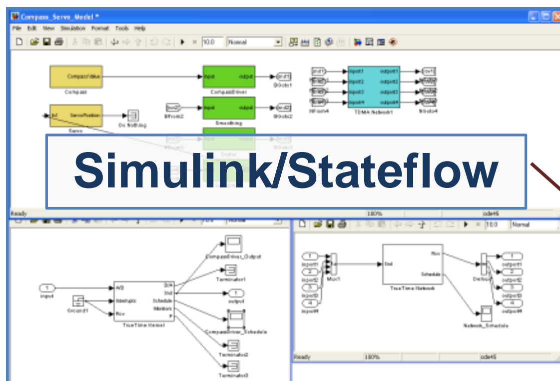
- (CM) Singleton components are needed - session-based components are *not feasible* (instantiated per client request)
- (CM) CM relies on dynamic memory allocation that *contradicts* with the ARINC-653 static allocation principles
- (CM) Chained component invocations *must not* result in self-deadlocks
- (CM) Has *no support* for monitoring the resource usage or the state of the component
- (CM) Exception handling and RTOS error handling must be *harmonized*, and exception cleanup must release all resources and locks
- (ARINC 653) Health monitoring must be *extended* to monitor and manage health of each component in (CM).

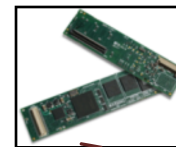
Issues:

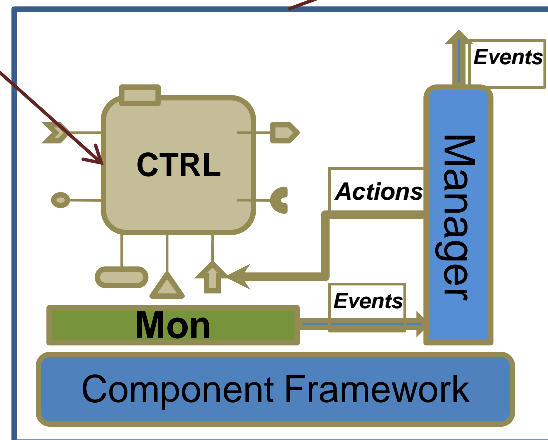
- In ARINC-653 process WCET is fixed and processes *cannot be created dynamically* → Every component method needs be implemented as a separate process [inefficient].
- (CM) event channels – as implemented by many – are not truly asynchronous, as they use the same thread → CM implementations must be *modified*.
- Non-(CM) intra-partition communication must not be used → A *disciplined* software development process is required.
- (CM) inter-partition communication should be mapped to sampling ports and queuing ports. → May lead to *inefficiencies* (over TCP/IP sockets).
- Mitigation actions need to consider that components can have multiple processes (one per method).

Progress

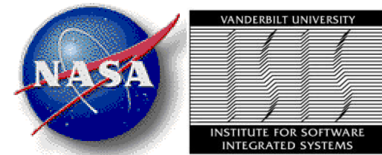
- Phase I – Testbed, experiment, demo - Status
  - In the MURI Project “Frameworks and Tools for High-Confidence Design of Adaptive, Distributed Embedded Control Systems” (VU, CMU, UCB, Stanford) we have developed prototype flight control software for a small UAV
    - Original vehicle design and controller by UCB/Stanford (C. Tomlin)
  - Low + high-level controller, available as Simulink model
  - Plan: use this model (2-component controller) as a vehicle to demonstrate a functional Component-level Health Manager.





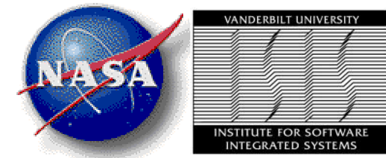


Conclusions



- Component-level health management *can* be implemented in a model-based component framework
- The component framework *shall* be tightly integrated with and based on a robust RTOS platform (e.g. ARINC-653)
- Prototype shows that such integration is feasible but several differences between traditional component frameworks and the RTOS must be *resolved*
- CCM-like abstractions (interfaces) *allow* the construction of the 'anomaly detection' services for health-management
- Generation of a component-level health manager from high-level models is feasible and such generation can reduce the coding effort needed
  - Models are used the generate code for and configure the component infrastructure

Next Steps



Immediate:

- Platform:
  - Specification language for condition monitoring
  - Extensions to the (CCM) IDL compiler and generator
  - Finalize ARINC emulator
- Model-based health manager:
  - Modeling language/tool for component architecture, monitors, and health manager behavior
  - Generator for component-level health manager
- Continuous integration, testing, and evaluation

Year 2:

- Develop modeling approach for cascading faults
- Design and develop system level health-manager
- Prototype and evaluate approach

Publications

- Abhishek Dubey, Gabor Karsai, Robert Kereskenyi, Nag Mahadevan: Technologies for Software Health Management, Technical Report for the project.
- Abhishek Dubey, Nag Mahadevan, Robert Kereskenyi: Reflex and Healing Architecture for Software Health Management; Extended Abstract for Workshop on Software-Health Management, SMC-IT, July, 2009.
- Abhishek Dubey, Gabor Karsai, Robert Kereskenyi, Nag Mahadevan: Towards a Real-time Component Framework for Software Health Management, submitted to RTAS 2010.
- Collaboration with SHM community: 1st International Workshop on Software-Health Management, at SMC-IT, Pasadena, CA, July, 2009 (http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm)