Prototype Implementation of a Goal-Based Software Health Management Service

Dr. Matthew Barry
Kestrel Technology

Gregory Horvath
Jet Propulsion Laboratory
California Institute of Technology

Copyright 2009. All Rights Reserved.
Agenda

• Concept Overview and Prior Work
• Project Requirements
• FAILSAFE Architecture
• Prototype Design and Implementation
• Lessons Learned
Overview - Failsafe

• **Idea**: Integrate various models of software performance and fault recovery to provide system-level software health management capabilities under ARINC 653 for high-assurance applications

• Integrate models of
  • System components and interface behavior
  • Component-level resource usage
  • Data and characteristic constraints

• Define formal and expected properties in standard modeling frameworks
Project Requirements
NASA Safety Requirements

- NASA’s Constellation Program Computing System Requirements
  - Control system safety requirements
    - Loss of function
    - Inadvertent activation
  - System safety specifications
    - Hardware/software interactions
    - State conditions, paths, interlocks
    - I/O queue drains and updates
    - Stale displays, intervals, content
    - Etc.

Ref: NASA JSC Constellation Program Computing Systems Requirements (CxP-70065)
FAILSAFE Architecture
Developing meta-models of elements of this framework will enable various architecture, design, and implementation details to be explored and prototypes evaluated during Phase II.
MDS has three parts:
An information and control system architecture
A systems engineering methodology
Reusable and adaptable framework software
This goal...

- Simple rules guided by state effect models define elaborations

• Projections add predicted effects to state time lines
• Models of effects between states and into the future handle all resource usage and other conflicts

Ref: MDS web site http://mds.jpl.nasa.gov/public/
Prototype Design and Implementation
Application Software Layer

\[ P = \{p_1, \ldots, p_n\} \]

Application Partition 1
- Process \( q_1 \)
- \( \text{IPC}_Q \)

Application Partition \( n \)
- Process \( q_n \)
- \( \text{IPC}_Q \)

\( \text{IPC}_Q = \) messages (buffers or blackboards) or synchronization (semaphores or events)

\( \text{IPC}_P = \) messages over channels as (sampling ports) or (queuing ports)

Core Software Layer

APEX Interface

System Partition 1

System Partition \( m \)

O/S Kernel

System Specific Functions

Hardware
Software Under Control

Partitions $P = \{ p_1, \ldots, p_{n-1}, p^* \}$

Application Partition 1
- Process $q_1$
- Process $q_q$
- Error Handler Process $q_i^*$
- SHM Queuing Port

Application Partition $n-1$
- Process $q_1$
- Process $q_q$
- Error Handler Process $q_i^*$
- SHM Queuing Port

Application Health Monitor Partition $p^*$
- SHM Queuing Port

APEX Interface

$IPC_Q = \text{messages (buffers or blackboards) or synchronization (semaphores or events)}$

$IPC_P = \text{messages over channels as (sampling ports) or (queuing ports)}$

Hardware
**Failsafe**

**Software Under Control**

1. Application Partition 1
   - Process q₁
   - IPC_Q
   - Error Handler Process q₁* 
   - Health Data
   - Repair Commands

2. Application Partition n-1
   - Process q₁
   - IPC_Q
   - Error Handler Process q₁* 
   - Health Data
   - Repair Commands

**Software Health Management**

3. Application Health Monitor Partition p*
   - Process q₁
   - IPC_Q
   - Error Handler Process q₁* 
   - Health Data
   - Repair Commands
   - SHM Queuing Port

**APEX Interface**

- SHM Sampling Port
- IPC_P Ports

---

**Notes:**

- IPC_Q = messages (buffers or blackboards) or synchronization (semaphores or events)
- IPC_P = messages over channels as (sampling ports) or (queuing ports)
SOFTWARE SAFETY DEMO

• Demonstration application: *Space Shuttle Abort Control Sequencer*
  • Performs OMS/RCS propellant dumps during launch aborts
  • Dumps necessary to manage
    • aerodynamic center and aerosurface loads
    • touchdown loads on main and nose gear
  • Several software processes involved
  • Good candidate for software health management
Software SAFETY CONSTRAINT

NOTES:
[1] VERIFIED AERO BOUNDARY.
[2] DESIGN ENVELOPE AFT LIMIT.
[3] FORWARD LIMIT FOR EOM, ATO, AND AOA (X=1075.2 INCHES).
[4] FORWARD LIMIT FOR GRTLS FOR ANGLE OF ATTACK < 30 DEGREES AND FOR TAL (X=1076.7 INCHES).
[5] FORWARD LIMIT FOR GRTLS FOR ANGLE OF ATTACK > 30 DEGREES (X=1079.0 INCHES).
[6] INITIAL CG INSIDE, MOVES INSIDE NOMINAL AT DUMP STOP.
[7] INITIAL CG INSIDE, STAYS INSIDE CONTINGENCY AT DUMP STOP.
[8] INITIAL CG INSIDE, MOVES OUTSIDE CONTINGENCY AT DUMP STOP.

NOMINAL CG ENVELOPE REFLECTS A 1-INCH UNCERTAINTY FOR X CG AND A 0.5-INCH UNCERTAINTY FOR Y CG. ©091098-6715
Software Management Problem

Manage software to manage propellant to manage CG imbalance

Less propellant on right moves CG leftward, potentially driving out of envelope
Health Monitor Development

- Began by performing a full state analysis for the application
  - State analysis artifacts form the basis for the software design and implementation
- State effects diagram took on a hierarchical structure
  - Also closely mirrored the ARINC 653 structure
- Software topology – specified in the collaboration diagram – derived from the state effects diagram
- MDS software framework components map directly back to the state analysis artifacts
  - Frameworks also implement some canonical patterns in some areas (e.g. goals on discrete state variables)
    - These patterns were leveraged in the prototype
Health Monitor State Effects Diagram
Software Collaboration Diagram

- Depicts main control loops (no redundancy shown here)
- Arrows indicate direction of data flow
- Methods called out indicate required interfaces for each component
Technical Lessons Learned

1) Partitioned RTOS platforms require a more involved build process than a traditional RTOS
   • Invest in faster build machines and standalone environment, plan builds

2) Platform support for certain language features should not be assumed
   • Verify all required features are supported, invest in newer platforms/toolchains

3) Initial integration tests should exercise all required software libraries and functions at a minimum, including all third party and externally developed libraries.
   • Design initial tests with all dependencies in mind, verify compatibility early on

4) Developing software for an ARINC 653-based partitioned OS requires much more thought, planning, and design to function properly, as compared to a traditional embedded RTOS platform
   • Plan effort appropriately, use existing metrics where available/applicable

5) For the domain of software health management (SHM), it is clear that the possibility for confusion arises due to its place among the other system elements
   • Clearly define terminology and boundaries of all system elements
Team

• **Kestrel Technology**
  • Matt Barry, Principal Investigator

• **Jet Propulsion Laboratory**
  • Greg Horvath, Fault Protection Engineer
  • Dave Wagner, Senior Software Engineer
  • Hui Ying Wen, Software Engineer
  • Others including lab support
Concept Overview and Prior Work
Modeling

Mission Data System
Canonical model for RTES
Elaborated via state analysis
Suitable for behavior identification...
Suitable for constraint monitoring...
Suitable for reasonableness...
Suitable for repair...
Other TBD...
Overview - STTR Phase I

- Phase I activity elaborated a general concept for software health management into specific functionality that:
  - Is realizable as a software product
  - Is specific to the needs of NASA's missions
  - Supports integration into current products
  - Uses modest amounts of computational resources
  - Is certifiable
Overview - STTR Phase II

- Phase II activity will translate the elaborated concept into software product prototypes that:
  - Realize the software health management concept
  - Leverage prior work and standards
  - Are applicable to the needs of NASA’s missions
  - Integrate with commercial and OSS products
  - Are certifiable
Project Requirements
Project Requirements

From contract

- Prototype Requirements
  - Functional Requirements
  - Verification Requirements
  - Implied Requirements

From NASA CxP

- Demo Requirements
  - Safety Requirements
  - Control System Requirements

- Failsafe Requirements
FailSafe Requirements — Demo Requirements — Safety Requirements — D3 Sensing

- Source map
- Unused sensors
- Homogeneous data
Prototype Design and Implementation
Flight Software Test Lab

- Constructed for JPL-led RTOS evaluation in FY06
- Contains all necessary hardware and software for our development purposes
- Two commercial Motorola PowerPC CompactPCI 750 Single Board Computers (MCP750 SBC)
- One board will be dedicated to Failsafe development
- Two Sun Blade workstation hosts running Solaris 9 and 10
- Configuration-controlled development environment
- Several TB of H/W RAID storage for test data archival
Target Application Development

• Originally specified as a set of nominal and off-nominal scenarios
  • Later distilled into a Java application
  • Java specification used as basis for implementation
• Final version ported to the Vx653 platform
  • Processes allocated among two ARINC 653 partitions
  • Communication between partitions using ARINC 653 ports
  • Communication between processes in the same partition using ARINC 653 objects (events, semaphores, blackboards)
  • Task management using ARINC 653 methods
• Each partition required a new thread of execution not specified in the Java version
  • HealthMonitorReporter thread provides partition-local data to the health monitor partition for state estimation
Health Monitor Integration Details

- MDS frameworks had not been run on an embedded system since major refactoring effort, undertaken in 2005.
- Several issues encountered during development
  - MDS frameworks rely on the ACE (Adaptive Communication Environment) frameworks for platform isolation layer
    - ACE had not yet been ported to the Vx653 platform
    - Team had to write a minimal version of ACE to provide the subset of features used by the HM adaptation
  - The typical MDS build process generates shared object libraries
    - Vx653 does not support C++ in shared libraries
    - MDS build system was changed to output static libraries, and target make rules were adjusted to link against the static libraries and to meet the requirements of the platform (can only specify a single object filename in the ARINC 653 configuration)
  - Refactored frameworks had not been run on a real-time platform, only as a standalone application on Linux workstation
    - Needed to come up with a simple design to integrate the real-time portions of MDS with the platform real-time elements (details in backup slides)
MDS Real-Time Executive Design
MDS Real-Time Execution Design

• ISSUE: MDS05 C++ frameworks were never run in a real-time multithreaded environment.
  • Previous MDS baselines were run on VxWorks, RTLinux, GreenHills, etc, but this was never ported
  • Take some cues from the MDS Java frameworks real-time execution design and implement in Vx653.
MDS Real-Time Execution Design (cont)

- Main Threads of Execution
  - Single thread which executes all of the achievers
    - Achievers are added to a rate group container
    - Executed sequentially
  - Single thread for MPE components
    - Executive
    - XGoalChecker
    - Goal Scheduler
      - Some uncertainty here due to current implementation of Scheduler in C++ frameworks
The goal scheduler in MDS 05 frameworks is currently implemented using a “yield scheduler”
- Idea stolen from gaming community
- Simulates multithreaded behavior in a single thread of execution by adding calls to a yield() method in the main thread
- When yield() is called, a loop is invoked which executes a small portion of the work required by some other job, then returns control to the main thread
- Large jobs are accomplished in small bits across many calls to yield().
- The work that happen within the yield() call essentially amounts to a “background thread” of execution, but without the overhead of context switching required in an actual multithreaded implementation.

Never run in a real-time environment, but shouldn’t be a problem
Compile time flag exists to compile out the yield() method
- In this case, the goal scheduler would simply run as another thread
- The frameworks have been compiled this way, but never run like this
Technical Lessons Learned
Lesson 1: Time Required for Build

• LESSON
  • Partitioned RTOS platforms require a more involved build process than a traditional RTOS due to the extra steps involved to ensure proper memory layout, etc. This in itself is a good thing; however, it can increase the time required to perform a build
    • As a result, full system builds can run significantly longer than for a traditional (non-partitioned OS) platform
      • For our small (compared to a full embedded control system) application, system builds took ~15 minutes

• RESOLUTION
  • Invest in faster build machines (multicore processor, more RAM)
  • Allow sufficient time for builds; schedule points when builds are to be done and plan for testing around those points
  • Invest in standalone test environment where application algorithms can be compiled and debugged rapidly
Lesson 2: Platform Language Support

• LESSON
  • Platform support for certain language features should not be assumed.
    • VxWorks PSC ARINC 653 (Vx653) does not support C++ application code in shared libraries.
    • Overall support for C++ on the Vx653 platform appears to lag support for C, as well as C++ support on other platforms.

• RESOLUTION
  • Do not assume all language features will be supported. Verify all required features before committing.
  • Investigate other platforms/toolchains (Green Hills Integrity, etc), which may provide better support for newer languages (C++, Java)
  • Investigate use of RTSJ for embedded applications
Lesson 3: Initial Integration Testing Goals

• LESSON
  • Initial integration tests should exercise all required software libraries and functions at a minimum.
  • Care should be taken to ensure that all external dependencies – e.g. third party libraries, frameworks, etc – are exercised during initial integration and compatibility testing.

• RESOLUTION
  • Design tests with all dependencies in mind.
  • Verify performance of a critical subset of the required features provided by the external software early on.
  • Verify that the resultant object code contains the references that are expected based on the test build configuration – i.e. if you expect a set of symbols to be exported from an external library into the final object that gets loaded to the target, verify that they do exist in that module using standard tools (GNU binutils, etc).
Lesson 4:
Software Development for ARINC 653 Platforms

• LESSON
  • Developing software for an ARINC 653-based partitioned OS requires much more thought, planning, and design to function properly, as compared to a traditional embedded RTOS platform
    • This does not imply any arcane qualities of the platform itself, rather the time required to familiarize one’s self with the mechanics of the platform and to design application targeting the platform
  • Interfaces between partitions must be well defined
  • Application memory usage must be tightly tracked and understood in order to maintain/update system configuration files
  • Legacy software must be assessed to determine which portions, if any, must be re-written to adhere to and employ standard platform-provided mechanisms

• RESOLUTION
  • Do not underestimate the time required for such an undertaking
  • Metrics for working with such systems are hard to come by, making labor planning/estimation difficult; to the extent possible, empirical metrics should be used in labor planning, especially for a group working with an ARINC 653-based platform for the first time
Lesson 5: MDS and State Analysis for SHM

• LESSON
  • For the domain of software health management (SHM), it is clear that the possibility for confusion arises due to its place among the other system elements
    • When interacting with stakeholders whose primary experience is with control system software, there is a very real possibility for confusion during technical exchanges
  • An SHM system is effectively a meta-control system, whose purview may include the elements of the control system software, hardware elements in the system under control, or a combination of the two.

• RESOLUTION
  • Clearly define the terminology used to describe the SHM system, and the terms used within the SHM to describe elements external to it
  • Clearly define the boundaries of all elements of the system – including the SHM system, the control system, and the system under control – as well as how each of these may interact with one another
ARGIN 653-1 Partition Mode Definitions

1a Mp1 = COLD_START ∧ Sp1 = START_CONDITION :: Mp2 = Mp1
1b Mp1 = WARM_START ∧ Sp1 = START_CONDITION :: Mp2 = Mp1
2 Mp1 = WARM_START ∧ Sp1 = START_CONDITION :: Mp2 = COLD_START
3a Mp1 = COLD_START ∧ (SET_PARTITION_MODE (IDLE) ∨ Ar = shutdown)) :: Mp2 = IDLE
3b Mp1 = WARM_START ∧ (SET_PARTITION_MODE (IDLE) ∨ Ar = shutdown)) :: Mp2 = IDLE
3c Mp1 = NORMAL_START ∧ (SET_PARTITION_MODE (IDLE) ∨ Ar = shutdown)) :: Mp2 = IDLE
4a Mp1 = COLD_START ∧ SET_PARTITION_MODE (NORMAL) :: Mp2 = NORMAL
4b Mp1 = WARM_START ∧ SET_PARTITION_MODE (NORMAL) :: Mp2 = NORMAL
5a Mp1 = NORMAL ∧ Sp1 = STARTCONDITION :: Mp2 = COLD_START
5b Mp1 = NORMAL ∧ Sp1 = STARTCONDITION :: Mp2 = WARM_START
6a Mp1 = IDLE ∧ A1 ∨ Ae :: Mp2 = COLD_START
6b Mp1 = IDLE ∧ A1 ∨ Ae :: Mp2 = WARM_START

Mp = Partition Mode = (COLD_START, WARM_START, NORMAL, IDLE) = OPERATING_MODE
Sp = Partition Status = (START_CONDITION) from GET_PARTITION_STATUS()
Ae = Action External to Partition = (power interrupt, hardware module reset, application reset)
Ar = Action recovery from health monitor = (shutdown)
ARINC 653-1 Process State Transitions Versus Partition Mode Transitions

1. \( Mp = \text{NORMAL} \land P1 = \text{DORMANT} \land \text{process started action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{READY} \)
2. \( Mp = \text{NORMAL} \land P1 = \text{READY} \land \text{process stopped action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{DORMANT} \)
3a. \( Mp1 \in \{\text{COLD_START}, \text{WARM_START}\} \land (Pt = \text{aperiodic} \land \text{process started}) \land \neg \text{process suspended action} :: \text{Mp2} = \text{NORMAL} \land P2 = \text{READY} \)
3b. \( Mp1 = \text{NORMAL} \land P1 = \text{READY} \land \text{Et} :: \text{Mp2} = \text{NORMAL} \land P2 = \text{READY} \)
4. \( Mp1 = \text{NORMAL} \land P1 = \text{READY} \land \text{process suspended action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{WAITING} \)
5. \( Mp1 = \text{NORMAL} \land P1 = \text{RUNNING} \land \text{(process waits on zero} \lor \text{process preempted action}) :: \text{Mp2} = \text{Mp}1 \land P2 = \text{READY} \)
6. \( Mp1 = \text{NORMAL} \land P1 = \text{RUNNING} \land \text{process selected action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{RUNNING} \)
7. \( Mp1 = \text{NORMAL} \land P1 = \text{RUNNING} \land \text{(process suspended action} \lor \text{wait} \in \text{We}) :: \text{Mp2} = \text{Mp}1 \land P2 = \text{WAITING} \)
8. \( Mp1 = \text{NORMAL} \land P1 = \text{RUNNING} \land \text{process stop action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{DORMANT} \)
9a. \( Mp1 = \text{NORMAL} \land P1 = \text{WAITING} \land \{ \text{wait} \in \text{We} \land \text{process suspended action} \lor \text{(process suspended action} \land \text{Et}) \} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{WAITING} \)
9b. \( Mp1 \in \{\text{COLD_START}, \text{WARM_START}\} \land P1 = \text{WAITING} \land \{ (Pt = \text{aperiodic} \land \text{process started}) \lor (Pt = \text{aperiodic} \land \text{process suspended}) \lor (Pt = \text{aperiodic} \land \text{process delayed}) \} :: \text{Mp2} = \text{NORMAL} \land P2 = \text{WAITING} \)
9c. \( Mp1 \in \{\text{COLD_START}, \text{WARM_START}\} \land P1 = \text{WAITING} \land \text{process suspended} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{WAITING} \)
10. \( Mp1 \in \{\text{COLD_START}, \text{WARM_START}\} \land P1 = \text{WAITING} \land \text{process stopped action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{DORMANT} \)
11a. \( Mp1 \in \{\text{COLD_START}, \text{WARM_START}\} \land P1 = \text{DORMANT} \land \text{process started action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{WAITING} \)
11b. \( Mp1 = \text{NORMAL} \land P1 = \text{DORMANT} \land \text{process started action} :: \text{Mp2} = \text{Mp}1 \land P2 = \text{WAITING} \)
12. \( Mp1 \in \{\text{COLD_START}, \text{WARM_START}\} \land P1 = \text{DORMANT} \land \text{process started action} :: \text{Mp2} = \text{NORMAL} \land P2 = \text{DORMANT} \)

\( Mp = \text{Partition Mode} = \{\text{COLD_START, WARM_START, NORMAL, IDLE}\} \land \text{OPERATING_MODE} \)
\( P = \text{Process State} = \{\text{DORMANT, WAITING, READY, RUNNING}\} \)
\( Pt = \text{Process Type} = \{\text{periodic, aperiodic}\} \)
\( \text{We} = \text{Waiting for Event} = \{\text{semaphore, event, message, delay}\} \)
\( \text{Et} = \text{process resumed action} \lor \{\text{resource available} \lor \text{timeout expires}\} \)
JPL FSTL
Hardware and Block Views

Images from JPL
RTOS Study Final Report, Len Day et al.
Health Management Tactics

- Fault-tolerant software design reuse
  - Architectural Patterns
  - Detection Patterns
  - Error Mitigation Patterns
  - Error Recovery Patterns
  - Fault Treatment Patterns

Each pattern definition contains applicability conditions and design rules for implementing its associated features.

See backup material for additional pattern type examples.
Candidate Detection Patterns

Failsafe architecture tactic implementation candidates.

Ref: Hanmer, “Patterns for Fault Tolerant Software”
Candidate Mitigation Patterns

Failsafe architecture tactic implementation candidates.

Ref: Hanmer, “Patterns for Fault Tolerant Software”
Candidate Recovery Patterns

Failsafe architecture tactic implementation candidates.

Ref: Hanmer, “Patterns for Fault Tolerant Software”