MODEL-BASED SOFTWARE
HEALTH MANAGEMENT
NASA Cooperative Agreement NNX08AY49A

Abhishek Dubey

Gabor Karsai

Robert Kereskenyi

Nag Mahadevan

Institute for Software-Integrated Systems, Vanderbilt University
Topics

- Objective of Research
- Hypothesis
- Progress to-date
  - Component-level Health Monitor
  - Modeling language
  - Testbed, experiment, demo (1)
  - System-level Health Manager
  - Testbed, experiment, demo (2)
- Next steps
Objective of Research

To develop solutions for software health management using model-based software development techniques

Goals
1. Review state-of-the-art in software health management
2. Develop a comprehensive approach to modeling and managing faults in software in the context of the system it is applied
3. Construct (a) a prototype modeling tool, and (b) a prototype multi-layered software health management engine that implements the principles involved
4. Demonstrate the feasibility of the approach through experiments using realistic problems, systems, and scenarios.

Perspective:
Model-based software tools that assist software developers in building complex airborne software that is robust against systemic faults – have positive impact on safety and supportability.
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 hypothesis:

A. A Software Health Management system *can* detect such faults, isolate their source, prognosticate their progression, and take mitigation actions.

B. Detection, isolation, prognostics, and mitigation shall be engineered in a model-based framework.

The work is directly related to the topic “IVHM 2.4 Software Health Management” of the IVHM Technology Plan.
Progress

• Phase I – Summary
  – Task 3: Component-level Health Manager
  – Task 4: Testbed, experiment, demo

• Phase II – Summary
  – Task 1: Modeling Language
  – Task 2: System-level Health Manager – in progress
  – Task 3: Testbed, experiment, demo – in progress
Recap: Component Health Management

• What are the components?
  – Component model

• How do we monitor them?
  – Monitor interfaces

• How do we detect anomalies?
  – Based on specs for anomalous/correct behavior

• How do we mitigate?
  – React to detection events / current state of health manager

• How do we implement all these?
  – Use a modeling language + generate code as needed
Recap: Notional Component Model

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
Required/Provided: Synchronous communication using call/return semantics. Triggering can be periodic or sporadic.
Recap: 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.
Recap: Component Monitoring

**Purpose:** to detect anomalies in the context of components.

Component monitoring must happen on the ‘right’ level of abstraction such that meaningful conditions for the component’s health could be formulated. Low-level, packet-oriented monitoring is ineffective.

Monitor events: trigger on the appearance of the event, access data associated with the event (if any)

Monitor interfaces: trigger on the execution of the method call, before and after execution of the method, access call parameters

Observe state: query values of state variables.

Monitor resource usage: keep track of dynamic and stack memory usage, and keep track of generic resource allocation/de-allocation operations (including timing)

Monitor control flow/triggering: detect invocation and return, keep track of timing (execution time, invocation frequency)

*Goal: Minimally intrusive monitoring – as little overhead as possible.*
Recap: Component Monitoring

- Monitor arriving events
- Monitor incoming calls
- Monitor published events
- Monitor outgoing calls
- Observe state
- Monitor resource usage
- Monitor control flow/triggering
How to build it?

• Model-based Software Development and Model-Integrated Computing:
  – Use a domain-specific modeling language to capture salient, functional and non-functional aspects of a system
  – Use software generators to produce code whenever feasible and/or necessary
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 $\rightarrow$ Partition
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, \texttt{delta}(var), and 
  \texttt{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 X State) → Action
# Implementation: Mapping a CM to APEX

<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</td>
</tr>
<tr>
<td>Sporadic</td>
<td>Aperiodic process</td>
<td>Semaphores</td>
</tr>
<tr>
<td>Invocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call-Return</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodic Target</td>
<td>Co-located</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Non-co-located</td>
<td>N/A</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></td>
<td>Non-co-located</td>
<td>Caller method sends RMI (via CM) to release callee then waits for RMI to complete.</td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>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></td>
<td>Non-co-located</td>
<td>Blackboard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sampling port, Channel</td>
</tr>
<tr>
<td>Sporadic Target</td>
<td>Co-located</td>
<td>Callee is released when event is available</td>
</tr>
<tr>
<td></td>
<td>Non-co-located</td>
<td>Blackboard, Semaphore, Event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Queuing port, Semaphore, Event</td>
</tr>
</tbody>
</table>
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
## Fault Detection and Mitigation scenarios

### Table of Fault 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>
Experiment: No Fault Scenario

Part 1 execution

Part 2 execution

Non-Linear Scale

1 Second 1 Second
Experiment: Bad GPS Data

Rate of change of GPS Data is above a threshold

Fault injection
Detection
Mitigation
Non-Linear Scale
• Focus issue: Cascading faults
  – Hypothesis: Fault effects cascade via component interactions
  – Anomalies detected on the component level are not isolated \(\rightarrow\) can be caused by other components

• Problem:
  – How to model fault cascades?
  – How to diagnose and isolate fault cascade root causes?
  – How to mitigate fault cascades?
Background: Fault diagnosis

• Model: Timed Failure Propagation Graphs

Modeling variants
- Untimed, causal network (no modes, propagation = [0..inf])
- Modal networks: edges are mode dependent
- Timed models
- Hierarchical component models

Nodes:
- Failure modes
- Discrepancies
  - AND/OR (combination)
  - Monitored (option)

Edges:
- Propagation delay: [min, max]
- Discrete Modes (activation)

Example models (#components,#failuremodes,#alarms)
- Trivial examples
- Simplified fuel system (~30,~80,~100)
- Realistic fuel system (~200,~400,~600)
- Aircraft avionics (~2000,~8000,~25000) – generated
Background: Fault diagnosis

• Fault diagnosis algorithm:

  • Outline:
    – Check if new evidence is explained by current hypotheses.
    – If not, create a new hypothesis that assumes a hypothetical state of the system consistent with observations.
    – Rank hypotheses for plausibility and robustness metrics.
    – Discard low-rank hypotheses, keep plausible ones.

  Metrics:
  
  *Plausibility:* how plausible is the hypothesis w.r.t. alarm consistency.
  
  *Robustness:* how likely is that the hypothesis will change in the future.

Fault state: ‘total state vector’ of the system, i.e. all failure modes and discrepancies.

Alarms could be

- Missing: should have fired but did not.
- Inconsistent: fired, but it is not consistent with the hypothesis.

Robust diagnostics: tolerates missing and inconsistent alarms.
Modeling Cascading Faults

• We don’t. The cascades can be computed from the component assemblies, if the anomaly types and their interactions are known.

• Component ‘elements’
  Methods belong to one of these (7)

• Fault cascades within component
  (A few of the 38 patterns)
Modeling Cascading Faults

- Inter-component propagation is regular – always follows the same pattern
- Intra-component propagation depends on the component! \( \rightarrow \) Need to model internal dataflow and control flow of the component.

Note: Could be determined via source code analysis.
Modeling Cascading Faults

- Fault Propagation Graph for GPS Example
  - Hand-crafted, but can/will be generated automatically
Mitigation – Work in Progress

- Plan: Construct a model-based system-level mitigation engine
  - Designer specifies fault mitigation strategies in a reactive state machine

Advantages:
- Models are higher-level programs to specify (potentially complex) behavior – more readable and comprehensible
- Models lend themselves to formal analysis – e.g. model checking
System-level Health Management

- **Modeling**: Generate the TFPG model automatically using (1) ‘templates’, and (2) component and assembly models
  - Lesson learned: TFPG discrepancy is a pair: (anomaly, hm-action)
- **Diagnosis**: Use a (simplified) TFPG engine for root cause determination: which component and/or method was the source of a fault cascade
- **Mitigation**: Build a (model-based) System-level Health Manager (SLHM) that takes mitigation action/s based on (1) reports from CLHM-s, (2) results of fault diagnosis, and (3) current state.
Fault Cascade Scenario

<table>
<thead>
<tr>
<th>S. No</th>
<th>Anomaly</th>
<th>Detected at</th>
<th>Fault source</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inject a fault to stop publishing data</td>
<td>Not detected</td>
<td>Sensor Component</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>Validity Failure</td>
<td>GPS Subscribe port</td>
<td>GPS Component</td>
<td>Use past data</td>
</tr>
<tr>
<td>3</td>
<td>Post Condition Violation</td>
<td>Nav Display Subscribe port</td>
<td>Nav Display Component</td>
<td>Ignore</td>
</tr>
</tbody>
</table>
Experiment: Fault cascades

Monitor info received in the Diagnosis Engine Component
Status: System-level HM

• Modeling:
  – Modeling language defined and tool is available
  – Fault cascade templates identified (38)

• Model generation:
  – Prototype generator for producing TFP graph models is functional
    but more work is needed

• Mitigation engine:
  – Concepts developed
  – Modeling language and generators - in progress

• Integration:
  – Hand-crafted prototype is functional
  – Fully automated integration - in progress
Testbed, experiment, demo

- Need: component-based avionics software example
- DARPA/MoBIES Program: Bold Stroke example
  - A suite of examples for component-based avionics software systems: typical mission computing scenarios
  - No sensitive code (only stubs)
  - Component assembly models exist (in another DSML)
- Plan:
  - Rebuild models for ACM
  - Create seeded fault scenarios
  - Demonstrate system-level health manager in action
Testbed, experiment, demo

• Testbed for experimentation:
  – 9 CPU-s with quad-core 2.4G Xeon, 4GB mem/320GB HD, 3xGBit Ethernet,
  – 24 port Gbit Ethernet programmable switch
  – 4 programmable 100Mb routers
  – 3x 19in LCD monitors (KVM switch)
  – Reconfigurable subnet + 1 developer node

• Provided by an AFOSR DURIP grant
Next steps

• Finish System-level Health Manager
  – Language for SHM Mitigation Models
  – Generator for SHM logic
  – Integrate and test

• Develop demo example
  – Bold Stroke architecture model/s
Publications

During the past 12 months:


• NASA Aviation Safety Technical Conference, November 2009, Washington DC [Talk given at the IVHM Software health Management Session]

• Software Health Management Technical Interchange Meeting, November 2009, Washington DC [Presentation and Demonstration]

• Software health management tutorial, Annual Conference of the Prognostics and Health Management Society 2009, San Diego, CA

Software:

• ACM Tool suite (ARINC Component Model tools including: modeling tool, generators, run-time platform – ARINC-653 emulator, documentation, experimental sample, etc.)

https://wiki.isis.vanderbilt.edu/mbshm/index.php/ACMTOOLSUITE