MODEL-BASED SOFTWARE HEALTH MANAGEMENT
NASA Cooperative Agreement NNX08AY49A

Abhishek Dubey

Gabor Karsai

Nag Mahadevan

Institute for Software-Integrated Systems, Vanderbilt University
Topics

- Objective of Research
- Problem Statement
- Progress to-date
  - Key ideas
  - Recap: Components and model-based development
  - Component-level Health Management
    - Monitoring
    - Mitigation
  - System-level Health Management
    - Monitoring
    - Diagnosis
    - Mitigation
  - Model-based deliberative reasoning for autonomous fault management
  - Experiments
- Next steps
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.
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 System (Vehicle) Level.
Key ideas

• Use *software components* as units of fault management: detection, diagnosis, and mitigation
  – Components must be observable, provide fault isolation, and be capable of mitigation

• Use a two-level architecture:
  – Component level: detect anomalies and mitigate locally
  – System level: received anomaly reports, isolate faulty component(s), and mitigate on the component

• Use models to represent
  – anomalous conditions
  – fault cascades
  – mitigation actions (when / what)

• Use model-based generators to synthesize code artifacts

  → Developer can use higher-level abstractions to design and implement the software health management functions of a system
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.
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.
ACM: The ARINC Component Model

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

  ![Diagram of ARINC Component Model]

  - **Terminology:**
    - Synchronous: call/return
    - Asynchronous: publish-return/trigger-process
    - Periodic: time-triggered
    - Aperiodic: event-triggered

  - **Note:**
    - *All* component interactions are realized via the framework
    - Process (method) execution time has deadline, which is monitored
• 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 call
  – Forwards outgoing synchronous RMI call

• Other interfaces:
  – State: to observe component state variables
  – Resource: to monitor resource usage
  – Trigger: to monitor execution timing
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:
- Based on CCM implementation/s
- No modification in CCM

ACM component interactions:
- Mainly implemented via APEX
- RMI interactions use threads
# Implementation: Mapping ACM 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 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</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>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></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>Non-co-located</td>
<td></td>
<td>Blackboard</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></td>
<td>Blackboard, Semaphore, Event</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 Language - Summary

Software Assembly Specification

Component Library

Component Parts

Component Health Management

Receptacle

Publisher

Facet

Consumer

Interface Definition

DataType

Integer, Char, Boolean, Aggregate

Map To

Computation System Specification

Platform Specification

Module Specification

Partition Specification

Semaphore, QP, SP, BB, Process, Channel, Inter-Module Channel, Partition

Specifications Hierarchy
Implementation: Model-driven Development

- **Platform:**
  - ARINC-653 Emulator on Linux
  - Open source CCM(s)
  - Module manager, infrastructure

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

Designer supplies functional code
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
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.
Component Monitoring

- Monitor arriving events
- Monitor incoming calls
- Monitor state
- Monitor resource usage
- Monitor published events
- Monitor outgoing calls
- Monitor control flow/triggering
Basic component monitoring:

- Monitor method execution time
- Monitor pre- and post-condition on methods
- Monitor code execution for exceptions

Conditions are defined over values and rates of method parameters and component state variables

→ Insufficient to determine whether the component is invoked correctly or whether it is in a correct state.

Example: Component c supports operations x(), y(), z() and the legal sequences of these operations are expressed using the following regular expression: ( x; y+; z; )*  
→ Pre/post conditions are insufficient
Extended component monitoring:
- Track component state with an observer automata
- The observer automata can also capture the fault mitigation logic

Example: ‘File’ component

Legal sequences of operations start with an open(), followed by arbitrary number of read() or write() operations, followed by a close().

Observer automata:
- States: nominal and mitigation states
- Transitions: triggered by operations, guarded by conditions
- Actions: Health management actions

Anomalies detected:
- Incorrect state of component
  - Cause: component itself
- Incorrect usage
  - Cause: ‘caller’ component
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.
## Health Mitigation Actions

<table>
<thead>
<tr>
<th>HM Action</th>
<th>Semantics</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE</td>
<td>Continue as if nothing has happened</td>
<td>Component</td>
</tr>
<tr>
<td>ABORT</td>
<td>Discontinue current operation, but operation can run again</td>
<td>Component</td>
</tr>
<tr>
<td>USE_PAST</td>
<td>Use most recent data (only for operations that expect fresh data)</td>
<td>Component</td>
</tr>
<tr>
<td>STOP</td>
<td>Discontinue current operation</td>
<td>Component,</td>
</tr>
<tr>
<td></td>
<td>Aperiodic methods: operation can run again</td>
<td>System</td>
</tr>
<tr>
<td></td>
<td>Periodic operations: operation must be enabled by a future START HM action</td>
<td></td>
</tr>
<tr>
<td>START</td>
<td>Re-enable a STOP-ped periodic operation</td>
<td>Component,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System</td>
</tr>
<tr>
<td>RESET</td>
<td>Stop all operations, initialize state of component, clear all queues, start</td>
<td>System</td>
</tr>
<tr>
<td></td>
<td>all periodic operations</td>
<td></td>
</tr>
<tr>
<td>CHECKPOINT</td>
<td>Save component state</td>
<td>System</td>
</tr>
<tr>
<td>RESTORE</td>
<td>Restore component state state to the last saved state</td>
<td>System</td>
</tr>
</tbody>
</table>
Modeling Language: Extensions to Component Monitoring

- **Component Health Manager**
  - HM: Local health manager logic
  - HM Publisher: reports component health
  - HM Consumer: receives system-level health management commands

- **Local health management logic:**
  - Events and actions (responses)
  - States: Observer + Fault Manager

- **Observer:**
  - Tracks component state

- **Fault Manager:**
  - Local mitigation actions

- **State Machine paradigm:**
  - Statecharts with timeouts

The Health Manager: Monitoring and decision making logic
System-level Health Management

• Focus issue: Cascading faults
  – Hypothesis: Fault effects cascade via component interactions
  – Anomalies detected on the component level are not isolated → 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?
Modeling Cascading Faults

- The cascades are automatically computed from the component assemblies, if the anomaly types and their interactions are known.

- Sources:
  - Component internal data- and control flows
  - Component Assembly Model

Example fault cascade within component:
Modeling Cascading Faults

- Fault Propagation Graph for GPS Example
  - Here: hand-crafted, but it is generated automatically in the system
Background: Fault diagnosis

- **Model:** Timed Failure Propagation Graphs

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

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

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

**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
Fault diagnosis algorithm:

- 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
System-level Fault Mitigation

- Model-based system-level mitigation engine
  - Model-based diagnoser is automatically generated
  - Designer specifies fault mitigation strategies using 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
Functional components

• 1. Aggregator:
  – Integrates (collates) health information coming from components (typically in one hyperperiod)

• 2. Diagnoser:
  – Performs fault diagnosis, based on the fault propagation graph model
  – Ranks hypotheses
  – Component that appears in all hypotheses is chosen for mitigation (TBD: OK?)

• 3. Response Engine:
  – Issues mitigation actions to components based on diagnosis results
  – Based on a state machine model that maps diagnostic results to mitigation actions

The Health Management Approach:
1. Locally detected anomalies are mitigated locally first. – Quick reactive response.
2. Anomalies and local mitigation actions are reported to the system level.
3. Aggregated reports are subjected to diagnosis, potentially followed by a system-level mitigation action.
4. System-level response commands are propagated to components.

Currently these are modeled as components, but they can be generated automatically.
System-level Fault Mitigation

• Experience: It is very tedious and error-prone to specify system-level fault mitigation actions using the reactive state machine paradigm – something simpler is needed

• Purpose of health management: to restore functionality in case of degraded software

• Rough idea:
  – Design-time:
    • Model the functional decomposition of the system
    • Model how functions are allocated to combinations of software components
  – Run-time:
    • Mark necessary functions as ‘active’
    • In case of software component failure search for an alternative set of components that restore the desired functions
    • Reconfigure
Functional decomposition

- Functions enumerated as simple AND-trees

  ![Diagram]

  - The InertialPosition function requires the GPSPosition function and the PositionTracking function, the latter requires the BodyAccelerationMeasurement function.
  - In the running system one or more such function trees are active
  - A lower level function may be required in multiple trees
Function mapping

- Functions require component/s
  - AND of some components (all)
  - XOR of some components (one of N)
  - M-of-N of some components
Example: IMU function mapping

4 out of 6 accelerometers are needed. There is a backup ADIRU, and a backup GPS.
Example: IMU function mapping

Modeling redundancy:
- Acceleration measurement can be supported by one of the primary or secondary ADIRUs
- ADIRUs have 3 sub-groups, all required (AND)
- The *AccelerometerGroup* of the ADIRU has 6 Accelerometers, 4 of which are required
States of components and groups

• Each component can be in state:
  – isActive = (true, false)
  – isFaulty = (true, false)

• Each group node in the function mapping tree can be:
  – isActive = (true, false)
  – isUsable = (true, false)

Functional decomposition is an ‘AND’ tree that relates top level aggregate functions to lower-level functions. If F are the set of all immediate children of a function node, $f_p$ in the functional decomposition tree, then

$$isActive(f_p) = (\forall f \in F)(isActive(f))$$
Run-time Computing the state

- **isUsable function**

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>$isUsable(c) \iff \neg isFaulty(c)$</td>
</tr>
<tr>
<td>And Group</td>
<td>$isUsable(g) \iff (\forall x \in child(g))(isUsable(x))$</td>
</tr>
<tr>
<td>XOR Group</td>
<td>$isUsable(g) \iff (\exists x \in child(g))(isUsable(x))$</td>
</tr>
<tr>
<td>MofN Group</td>
<td>$isUsable(g) \iff (\exists X \subseteq child(g))(\vert X \vert \geq M)(\forall x \in X)(isUsable(x))$</td>
</tr>
<tr>
<td>Function</td>
<td>$isUsable(f) \iff (\forall x \in child(g))(isUsable(x))$</td>
</tr>
</tbody>
</table>

Table 1: $isUsable$ semantics: g means group. c means component. Operator parent (x) returns the set of all immediate parents of x in the function allocation DAG. Operator child(x) returns the set of immediate children of x. Note: $\vert . \vert$ is the cardinality operator

- **isActive function**

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>$isActive(c)$ is marked by the deployment scheme and any previous action of the reasoner</td>
</tr>
<tr>
<td>And Group</td>
<td>$isActive(g) \iff (\forall x \in child(g))(isActive(x))$</td>
</tr>
<tr>
<td>XOR Group</td>
<td>$isActive(g) \iff (\exists x \in child(g))(isActive(x))(\forall y \in child(g)) (y \neq x)(\neg isActive(y))$</td>
</tr>
<tr>
<td>MofN Group</td>
<td>$isActive(g) \iff (\exists X \subseteq child(g))(\vert X \vert \geq M)(\forall x \in X)(isUsable(x))(\forall y \in child(g)/X)(\neg isUsable(y))$</td>
</tr>
<tr>
<td>Function</td>
<td>$isActive(f) \iff (\forall x \in child(g))(isActive(x))$</td>
</tr>
</tbody>
</table>

Table 2: $isActive$ semantics.
Run-time Computing the reconfiguration

- Update the DAG with faulty components

**Algorithm 1: Mark as faulty**

**Given:** Allocation DAG T. Operator parent (x) returns the set of all immediate parents of x in the function allocation tree. Operator child(x) returns the set of immediate children of x.

**Given:** Global set GRC : Set of Possible Reconfig Commands

**Given:** Global set GRN : Set of Possible Reconfig nodes in the Tree T.

**Given:** RN is an empty set.

**Input:** Faulty Component c

1. c.isFaulty = true
2. visitParent(c, RN)
3. if isempty(RN) then
   4. output-command = RESET(c)
5. else
   6. output-command = STOP(c)
7. GRN.add(RN)
8. GRC.add(output-command)
9. end if
Run-time Computing the reconfiguration

- Propagate fault information to parents

Algorithm 2 visitParent

Given: Allocation DAG T. Operator parent (x) returns the set of all immediate parents of x in the function allocation tree. Operator child(x) returns the set of immediate children of x.

Given: Global set GRC : Set of Possible Reconfig Commands

Given: Global set GRN : Set of Possible Reconfig nodes in the Tree T.

Input: Node N, Set of Reconfig Node RN

Output: Set of Reconfig Node RN

1: \( P = parent(N) \)
2: for \( p \in P \) do
3:   if isUsable(p) then
4:     RN.add(p) {add a usable node to possible reconfig nodes}
5:   else
6:     if \( p \in RN \) then
7:       RN.remove(p)
8:     end if
9:     visitParent(p, RN)
10: end if
11: end for
12: return RN

- Returns the parent nodes that are still usable and can be used for reconfiguration
Run-time

Execute the reconfiguration

• Run all reconfiguration actions

Algorithm 3 RunReconfig

Given: Global set GRN : Set of Possible Reconfig nodes in the Tree T.
1: for n ∈ GRN do
2:   Reconfig(n)
3: end for
4: GRN.clear()
5: return
Run-time
Execute the reconfiguration

For a given node:
- Component: START
- AND: reconfigure all children
- XOR: find one other usable child and reconfigure
- M-of-N: find children and reconfigure

```
Algorithm 4 Reconfig
Given: Allocation DAG T. Operator parent (x) returns the set of all immediate parents of x in the function allocation tree. Operator child(x) returns the set of immediate children of x.
Given: Global set GRC ; Set of Possible Reconfig Commands
Input: Node N
1: if isUsable(N) then
2:   if N.type() ==COMPONENT -isActive(N) then
3:     N.isActive=true
4:   Output-Command = START(N)
5:   GRC.add(Output-Command)
6: end if
7: if N.type() == MoNGROUP then
8:   CN=child(N)
9:   for x ∈ CN do
10:      Reconfig(x)
11: end for
12: N.isActive=true
13: end if
14: if N.type() == ANDGROUP then
15:   if isUsable(N) then
16:     CN=child(N)
17:     for x ∈ CN do
18:        Reconfig(x)
19:      end for
20:   N.isActive=true
21: end if
22: if N.type() == XORGROUP then
23:   CN=child(N)
24:   for x ∈ CN do
25:     if isUsable(x) then
26:       Reconfig(x)
27:     for y ∈ CN and y ≠ x do
28:        if isActive (y) then
29:          output-command=STOP(y) {if y is a group. This generates commands for all children components of the group}
30:         GRC.add(output-command)
31:       end if
32: end for
33: N.isActive=true
34: return {It will return as soon as the first is usable child is found}
35: end if
36: end for
37: end if
38: end if
39: return
40: end if
```
System-level Fault Mitigation
Deliberative reasoning engine

• Examples:
  – GPS with simple backup failover
  – IMU Example
    • ADIRU with internal redundancy
    • Backups for ADIRUs and GPS receivers
Next steps
(NCE)

• Relax constraint on XOR group:
  – Component can be in >1 XOR groups

• Experiment with a SAT solver
  – Identify situations where current approach is weak and a SAT solver would be needed (e.g. XOR groups above)
  – Pre-compute solution at design-time, cache it for use at run-time

• Allow DAGs instead of trees:
  – Currently intermediate group nodes have a single group parent. The knowledge of a “similar”/”same” group is not captured and used in the reasoner

• Component choices may have dependencies.
  – What is the ‘best’ recovery action sequence?
  – How to order the sequence?

• How to search for the recovery action given limited resources? (e.g. time)

• A more complex demonstration


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/ACMTOOLSUITET