MODEL-BASED SOFTWARE HEALTH MANAGEMENT FOR REAL-TIME SYSTEMS

NASA COOPERATIVE AGREEMENT NNX08AY49A

Abhishek Dubey
Gabor Karsai
Nag Mahadevan

Institute for Software-Integrated Systems,
Vanderbilt University,
Nashville, TN
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.

Problem Statement

- Software is a complex engineering artifact that can have latent faults, uncaught by testing and verification.
- Faults appear during operation when unforeseen modes and/or (system) faults appear.
- Techniques like Voting and Self-Checking pairs have shortcomings
  - e.g. Common Mode Fault, Fault Cascades

Our Goal
Extend classical techniques with solutions for software health management using model-based software development techniques. This requires a comprehensive approach to modeling and managing faults in software in the context of the system it is applied. Specifically,
1. Detection: Identify anomalies.
2. Fault Isolation: Identify source(s)
3. Mitigation: Reduce/remove the affect of faults. (Prevent software fault from causing system failure.)
4. Prognostication: Identify future source of faults/failures
Overview of our Approach

• **Component-based software construction**: Developed for ARINC-653 based systems.

• *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

• System-Level Diagnosis
  - Generate causal fault models from the knowledge of component interconnections.

• *Mitigation*: **Component Level and System-Level**
  - Take action given detected anomaly and state of the component
  - Actions: restart, initialize, block call, inject value, inject call, release resource, modify state; combination of the above

• All of the above are facilitated by model-based software construction techniques.
ARINC-653 Component Model

- A CORBA Component Model-like layer on top of ARINC-653 abstractions.
- Synchronous Interfaces: call/return
  - Periodic: time-triggered
  - Aperiodic: event-triggered
- Asynchronous Interfaces: publish-subscribe
  - Periodic: time-triggered
  - Aperiodic: event-triggered
- Additional Monitoring interfaces
  - State: to observe component state variables
  - Resource: to monitor resource usage
  - Trigger: to monitor execution timing

Built using MICO ORB and ARINC 653 primitives
Component Monitoring

- Post/Pre condition violations: threshold, rate, custom filter (moving average)
- Resource Violations: Deadline
- Validity Violation: Stale data on a consumer
- Concurrency Violations: Lock Time Outs.
- User code violations: reported error conditions from application code.
  - ARINC-653 style errors.

Port Monitors
- Monitor arriving events
- Monitor incoming calls
- Monitor published events
- Monitor outgoing calls

Non-Port Monitors
- Observe state
- Monitor resource usage
- Monitor control flow/triggering

Model-Based Software Health Management
Example: Assembly of 3 components deployed on two partitions

- The Sensor component is triggered periodically and it **publishes** an event upon each activation.
- The GPS component **subscribes** to this event and is triggered periodically to obtain GPS data from the receiver. It publishes its own output event.
- The Nav Display component is triggered sporadically via this event and it uses a **required** interface to retrieve the position data from the GPS component.

Specified Monitoring Conditions:

- Validity(GPS.data_in) < 4 ms
- \( \text{Delta}(\text{Nav.data_in.time}) > 0 \)
- Rate(gps_data_src.data) > 1

<table>
<thead>
<tr>
<th>Period (secs)</th>
<th>Component</th>
<th>Port</th>
<th>WCET (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.004</td>
<td>Sensor</td>
<td>data_out</td>
<td>0.004</td>
</tr>
<tr>
<td>0.004</td>
<td>GPS</td>
<td>data_in - after processing sends an event to Display</td>
<td>0.004</td>
</tr>
<tr>
<td>Sporadic</td>
<td>Display</td>
<td>data_in - after processing calls GetGPS-Data</td>
<td>0.004</td>
</tr>
<tr>
<td>Sporadic</td>
<td>GPS</td>
<td>gps_data_src.GetGPSData</td>
<td>0.004</td>
</tr>
</tbody>
</table>
Formally Specified Component-Level Mitigation

e1 (deadline violation) of the periodic consumer data_in of GPS Component
e2 (Validity Monitoring) defined on the periodic consumer data_in. Validity implies age of data < 4 ms

e1 (post condition) defined on data_in consumer
= (DELTA(data_in.time) > 0)
e2 (post condition) defined on
gps_data_source.get_gps_data
required interface method = (RATE(gpsData) > 1)
Fault Scenario: Injected fault such that sensor stops publication
Background: Fault diagnosis

- **Model**: Timed Failure Propagation Graphs

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

**Edges:**
- Propagation delay: \([\text{min}, \text{max}]\)
- Discrete Modes (activation)

**Modeling variants**
- Untimed, causal network (no modes, propagation = \([0..\infty]\))
- 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
Modeling Inter-Component 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)

**Example**
Modeling Intra-Component Cascading Faults

- Inter-component propagation is regular – always follows the same pattern

- Intra-component propagation depends on the component! → Need to model internal dataflow and control flow of the component.

Note: Could be determined via source code analysis.
Example: GPS Assembly

The System TFPG is generated from the given assembly model.
Diagnosis Example

===[ Hypothesis Group 1 ]=====

Fault: FM_Sensor_data_out_USER_CODE time: [0.000000, 24.341104]
Supporting Alarms: VALIDITY_FAILURE (GPS DataIn), POSTCONDITION_FAILURE (NavDisplay GetGPSData)
Plausibility: 100.000000 Robustness: 100.000000 FRMetric: 0

===[ Hypothesis Group 2 ]=====

Fault: Sensor_LOCK_PROBLEM time: [0.000000, 24.341104]
Supporting Alarms: VALIDITY_FAILURE (GPS DataIn), POSTCONDITION_FAILURE (NavDisplay GetGPSData)
Plausibility: 100.000000 Robustness: 100.000000 FRMetric: 0
Summary

- **Faults in systems and in software** are well recognized and both detection and mitigation techniques are available.
- For SHM, we require systematic software construction. One approach is to use reusable **software components** with limited/but fully defined interaction semantics.
- Fault detection for software is **difficult**, because it is often hard to define what the correct behavior is. Hence, techniques are needed for specifying 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 is more **pragmatic** to indict solely the faulty software component, with some model of how it has failed.
- Fault mitigation and recovery is 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. Our approach tends toward a, reusable framework for the approach is still a research challenge.
Questions?