Reflex and Healing Architecture for Software Health Management

Abhishek Dubey
Nagbhushan Mahadevan
Robert Kereskenyi

First International Workshop on Software Health Management, July 2009
Rise in Software Complexity

- Software used in complex systems such as avionics has increased in size and complexity over last four decades.
- A large number of functionalities in these systems are implemented in software.
- Reliability of the system is contingent upon
  - Dependability of the embedded computing system
  - Dependability of the software implementing the logic

“Orion flight software size estimates exceed 1 million lines of code, which means that it will be more than 100 times larger than Apollo flight software and will run on avionics with thousands of times the RAM and CPU performance of the Apollo avionics.”


<table>
<thead>
<tr>
<th>Cause of outage</th>
<th>1985</th>
<th>1987</th>
<th>1989</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware failure + Environment</td>
<td>35%</td>
<td>32%</td>
<td>13%</td>
</tr>
<tr>
<td>Software failure</td>
<td>34%</td>
<td>39%</td>
<td>62%</td>
</tr>
<tr>
<td>Maintenance and operations</td>
<td>27%</td>
<td>24%</td>
<td>20%</td>
</tr>
<tr>
<td>Unknown</td>
<td>4%</td>
<td>5%</td>
<td>5%</td>
</tr>
</tbody>
</table>

“Software faults cost the U.S. economy $59.5 billion annually, representing 0.6% of the US gross domestic product” [NIST2002]
Software Health Management

- Software can have latent faults, uncaught by testing and verification. Such faults become apparent during operation when unforeseen modes and/or (system) faults appear.
- SHM: Apply concepts and techniques of physical system health management such as FDIR to software integrated systems.
- There are differences between physical systems and software systems

1. Physical systems (continuous) are governed by differential equations
2. State space is implicitly specified (symbolically).
3. Faults appear as change in value of parameters. For incipient faults, the discrepancy in observation is gradual.
4. State space, even though infinite is symbolically explored using real analysis.
5. Prognosis involves predicting the future system states based on change in parameters and environment input

1. Software is typically governed by a timed discrete event system model.
2. Representing state space is a potential challenge. Often a combination of symbolic and explicit specification is used.
3. Faults appear when the system transitions to a faulty state. This transition is instantaneous.
4. State space is often very large and has to be usually explored via model-checking using various abstractions.
5. Prognosis involves predicting the set of traces that might eventually lead to a failure state.
Reflex and Healing Approach

- Developed for managing the filter farm for a large physics experiment. The farm had to process ~1-2TB/s of data over 2x10^7 channels, and use ~2500 embedded processors with an additional ~2500-node commodity cluster.
- Strict real-time deadlines, as the interactions within the particle accelerator are produced by a well known periodic process.
- Data is reduced over several “levels” by the data collection system in an effort to store only the most scientifically interesting interactions.
- Reflex (Reactive): A fast, involuntary, preprogrammed, localized response to stimuli
  - Using the timed discrete event system model of software.
  - Depending upon history of executed events, the reflex action is achieved by driving the system through externally issued events to a desirable state.
  - e.g. changing the resolution parameter of a software sensor component if it is taking too long to process.
- Healing (Planning): Coordinated, long term adaptation of systems to recover from adverse conditions.

Reflex Demonstrations

• First, a system was modeled and deployed whereby components and dataflow interactions were modeled and deployed on a network of 16 embedded DSP processors
  – The mitigation behaviors of this system were static and created by hand
  – The system was able to recover from a variety of failure scenarios. A live demonstration of this system, including the tools used to create and deploy it, was given at the Fermilab/SLAC exhibit floor at SuperComputer2003 (SC2003)

• Then, another system of larger scale was proposed, built, and deployed on a 84-node cluster located at Fermi National Laboratory.
  – Cluster was comprised of systems which had been decommissioned from other clusters due to known failures
  – A set of fault mitigation scenarios were successfully demonstrated at the FALSE-II workshop, co-located with the Real Time Applications Symposium (RTAS 2005).
  – Behaviors which implemented all of the recovery actions were modeled in the design tools as reflexes within a RH Architecture
  – Components to implement the behaviors were automatically generated and deployed throughout the system exclusively from models

• Recently, we have also implemented this architecture for large scientific computing clusters located at Fermi National Laboratory.
Reflex and Healing Architecture

• Sensors are software as well as hardware components generating **timed event traces**.
• Reflex engines have two components: monitors and mitigators.
• Hierarchical arrangement with strict interaction constraints for fault containment and scalability.
• Fault are detected as discrepancies
  – Violation of the timed property used to create the monitor.
• Mitigation via action scripts associated with the timed state machines of actuators.
  – A timeout is generated if the action script does not finish in the bounded time.

A real time reflex engine $E_r$ is a tuple $< S, B, \mathcal{I}, \mathcal{I}', \mathcal{X}_i, \mathcal{X}_o, T >$, where $S$ is the scheduler, $B$ is a buffer, $\mathcal{I}$ is a parallel composition of all the enabled strategies and $\mathcal{I}'$ is the set of disabled strategies, $\mathcal{X}_i$ is the set of all the possible inputs to a reflex engine, $\mathcal{X}_o$ is the set of all the possible outputs generated by the reflex engine and $T$ is the set of timers used to generate timeout events.
Example: Heartbeat

Used for ensuring the computing cluster is online.

1. **Heartbeat Generator**
   - TimeOut1?/ResetTimer1?/Heartbeat!
   - Init RequestPong(Nodeld)
     - if (ping(Nodeld) != success) return 1
     - else return 0

2. **Heartbeat Filter**
   - Heartbeat?/ResetTimer2?
   - H = 1
   - Initial
   - TimeOut2?/ResetTimer2?/Heartbeat!
   - H = 0

3. **Timer1** (Generator)
   - [x < 300] TimeOut1!
   - Initial
   - X = 300
   - ResetTimer1?/x = 0

4. **Timer2** (Filter)
   - [x = 600] TimeOut2!
   - Initial
   - X = 600
   - ResetTimer2?/x = 0

X is the clock used to measure Timeout.

**Heartbeat Monitoring**

- Normal
- Rescue needed
- Heartbeat H = 0

**Reflex Engine**

- Heartbeat H = 1
- MarkOnline [H = 1]

**Node Down**

- CallHBMitigator
- Inoperable
- Detected
- Failed

**Heartbeat received**

- PBS job running?
- Is re-runnable?
- Yes
- Restart job
- No

**Mark inoperable**

- Is IPMI OK? (can we talk to BMC?)
- Yes
- Reset IPMI
- Is IPMI still OK?
- Sleep 10 min
- Is machine up? Is heartbeat working?
- Yes
- Mark online
- No (3 times)
- No

**At any point if timeout happens the node is marked inoperable.**
Healing

• When designing reflex and healing systems a great deal of information is **locked up in the models**

• When the requirements or environment experience an **unforeseen change**, it becomes necessary to go back to the model
  – Example: A critical number of boards in a rack fail and the resource model from which the runtime was built is no longer representative
  – In this case, a new resource model is fashioned and the existing tasks (wherever possible) are mapped onto the diminished resources
  – This often requires a systematic exploration of alternatives as either a **constraint satisfaction problem** or an **optimization problem**.

• For large computing clusters at Fermi Lab, this requires the workflow management subsystem to reevaluate the workflow dependencies and stop a workflow if it cannot be completed.
Diagnosis

- Initial research on RH was based on mapping discrepancies to mitigation.
- Diagnosis: produce hypotheses (cause of fault) inferred from observed discrepancies.
- Diagnosis of software systems is difficult
  - Lack of mathematical models
  - Cause and effect relationships are dynamic and difficult to capture for a software system due to large state space.
  - Software components scheduled on the set of nodes change during system runtime
  - Interaction of multiple actors complicate this even further in distributed systems.
  - This problem is simplified in the safety critical system by assuming a static configuration of software components.

Example: The large state space of faults in clusters used for scientific computing

- Vanderbilt University researchers have previously developed an approach for modeling failure propagations in complex dynamic systems using Timed Failure Propagation Graphs.
- We are investigating its use with RH architecture for diagnosing software faults.
Timed Fault Propagation Graphs

- Simple causal failure model
  - Enriched with temporal and multi-mode constraints.
  - Handles Multiple faults.
  - Robust against alarm failures.
  - Intermittent Alarms.
- \( F \): set of failure modes. \( D \): set of discrepancies. \( V = F \cup D \)
- Discrepancy attributes:
  - Type: \{AND, OR\}
  - Condition: \{Monitored, unmonitored\}
- \( E \): set of edges. Edge attributes:
  - Propagation interval: \([t_{\text{min}}, t_{\text{max}}]\)
  - [Active Mode(s)]

TFPG Hypothesis

TFPG Hypothesis - estimation of the current state of system

- **Directly**, points to failure modes that “best” explain the current set of observed alarms.

- **Indirectly**, points to failed monitored discrepancies; those with a state that is inconsistent with the (hypothesized) state of the failure modes

- **Structure**
  - List of possible Failure Modes
  - List of alarms in each set (Consistent (C)/ Inconsistent (IC)/Missing (M) / Expected (E))

- **Metrics**
  - Plausibility = \( \frac{C}{C + IC} \)
  - Robustness = \( \frac{C + IC}{C + IC + E} \)
TFPG and Distributed Software Systems

Using TFPG in large software systems pose two challenges:

- System components can operate in different modes, which leads to **dynamic software dependency chains during runtime**. This might require large-scale rewiring of models based on the concepts of modes supported by TFPG.
  - This is true for a large-scientific computing clusters where a workflow management system can be executing a different job at a given time that might interact with other jobs on the system. This essentially changes the TFPG from time to time
  - However, in a safety critical environment that uses an ARINC 653 compliant OS processes cannot be created dynamically. Effectively, this will lead to a smaller and mostly fixed cause and effect chain for the software system.

- **Search space of a global TFPG model could be very large**. Hence, we will need a component-based distributed reasoning approach where regional reasoners generate regional hypothesis, which could subsequently be improved by a global reasoner.

- A distributed and hierarchical TFPG reasoning scheme is currently being developed at Vanderbilt University to address this problem. The TFPG model is split into a global model and multiple regional models where each region model accounts for a specific sub-system.
Distributed TFPG: TFPG-D
Distributed Reasoner – Model Input

Failure Propagation across Regional Models
I/O Ports interaction

Global Reasoner (GR) Initialization

Regional Reasoner(s) (RR) Initialization
Hypothesis Update

• Regional Reasoner
  – Update hypothesis for regional events or events across the boundary through I/O Ports.
  – Communicate any boundary propagation to Global Reasoner

• Global Reasoner
  – Ensures transmission of failure effects across Regional reasoners.
  – Assimilates Update from Regional Reasoners.

• Current Assumptions
• No Network/Node Failure
• Strict Communication constraint
  – Regional TFPG do not communicate to each other
Work in progress

- We are working towards developing software health management solution for safety-critical systems that builds over Reflex and Healing architecture and Timed Fault Propagation Graphs.

- Software units abstracted as components.
  - 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.

- Platform
  - ARINC-653 like RTOS emulated on Linux

- Reflex architecture mapped to ARINC-653 specification.

Table 1: Mapping RH hierarchy to ARINC-653 specification

<table>
<thead>
<tr>
<th>RH Specification</th>
<th>ARINC Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local-level</td>
<td>Process-level</td>
</tr>
<tr>
<td>First Regional-level</td>
<td>Partition-level</td>
</tr>
<tr>
<td>Second Regional-level</td>
<td>Module-level</td>
</tr>
<tr>
<td>Global-level</td>
<td>System-level</td>
</tr>
</tbody>
</table>
Summary

- Software is a complex engineering artifact.
  - Typically abstracted as a timed discrete event model.
  - We encapsulate software entities as components.

- Reflex and Healing Architecture
  - Sensors, Monitors, Mitigators
  - Can be mapped to an ARINC-653 compliant system.

- Timed Fault Propagation Graph
  - Restricting dynamic process creation simplifies the global TFPG model.