Toward Monitoring Fault-Tolerant Embedded Systems

Alwyn E. Goodloe
National Institute of Aerospace

Lee Pike
Galois, Inc
Characterizing the Systems

The systems we focus on must be ultra-reliable, and so demand catastrophic-failure rates of $\leq 10^{-9}$ per hour of operation.

- Fault-tolerant system - provides required functionality in the presence (bounded number) of faults
  - Should not allow a single point of failure
    - Replicated hardware & distributed architectures
    - Fault Containment Regions - Isolate components
    - Fault-management SW
  - Hard real-time
- Problems usually stem from the physically possible, but logically unanticipated
The Problem: Motivation

Space Shuttle

- In 2008, a pre-launch failure of STS-124 was reported in the Space Shuttle’s data processing system.
- Components:
  - **FA 2**: the flight-aft mux/demux card
  - **GPC n**: general-purpose computers \( n \)
- The incident:
  1. A diode fails on FA 2
  2. GPC 4 receives bad data from FA 2; in the data comparisons with GPC 1-3, it is voted out.
  3. Then similarly for GPC 2.
  4. GPC 3 also determined to be faulty.
  5. With only one GPC remaining, the system was powered-down.
- Described as a “non-universal I/O error”
Hybrid Fault Model

- **Fault models** capture faults that the system should handle
  - Many explicit and implicit assumptions
- Classify nodes (Thambidurai and Park)
Fault-Model Violations

- A maximum fault assumption (MFA) states the maximum number of each kind of fault a system designed to withstand.
- An MFA along with the fault-arrival rate gives you its hypothesized reliability.
- Too often hypothesized reliability > actual reliability:
  - Design errors (i.e., systematic faults) cause the actual MFA to be a subset of the hypothesized MFA.
  - Designers underestimate the MFA required to achieve the desired reliability. The Shuttle incident arguably resulted from an underestimated MFA.
Monitors

- **System Under Observation (SUO)** – system being monitored
- Correctness property $\phi$
  - Temporal logics
  - Regular expressions
  - State machines
- Monitors – observe the SUO and detect violations of $\phi$
  - Accept all execution traces satisfying $\phi$
  - RT Distributed Systems give rise to false positives and false negatives
Previous Efforts Focus

Previous research on monitors mostly focuses on systems lacking **one or more** characteristics of ultra-reliable systems.

- Focus on Java programs, e.g.,
  - Run-time Monitoring and Checking (MaC) – Insup Lee et al.
  - Monitoring-Oriented Programming (MOP) – Grigore Rosu et al.

Specification-logics that capture properties about program traces, e.g.

- Variants of past-time temporal logic
- Compile specifications to efficient monitors
  - RuleR – Barringer, Rydeheard, and Havelund
Needed for this Domain

- Focus on Hard real-time systems.
  - Several case studies of soft real-time systems
- Focus on fault-tolerant systems.
  - Well constructed, extensive certification testing, subtle errors often arise in deployment arising from assumptions and hardware failures.
- Focus on synchronous distributed systems.
  - For instance timed-triggered systems.
Related Work

A few efforts have touched on aspects of safety-critical embedded systems. Representative efforts include:

- MOP extensions to monitor distributed programs using a past-time modal logic.\(^1\)
- BusMOP: synthesizing high-level specs onto FPGAs for zero-overhead bus monitoring.\(^2\)
- Logics for monitoring real-time systems (particularly distributed Java programs).\(^3\)
- Runtime Verification of C Programs.\(^4\)

\(^1\)[Sen, Vardhan, Agha, Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems, ICSE’04.]

\(^2\)[Pellizzoni, Meredith, Caccamo, Rosu. Hardware Runtime Monitoring for Dependable COTS-based Real-Time Embedded Systems, RTSS’08.]

\(^3\)[Mok and Liu. Efficient Run-Time Monitoring of Timing Constraints. RTAS97.]

\(^4\)[Haveland. Runtime Verification of C Programs, ]
Consensus Properties

• **Consensus** – Nonfaulty nodes agree on a value
  - Inexact consensus agree within $\varepsilon$
    - Clock Synch.

• **We propose to monitor for consensus in distributed systems.**
  - A monitor that can observe consensus (or the lack thereof) between distributed components.

• This principally means observing classes of asymmetric or Byzantine faults (including omissive faults).
Consensus Properties

- What faults can be couched in terms of consensus?
  - Fault-model violations
  - Timing violations
  - Point-to-point error-checking (Haskell Symposium 2009)
Monitors for Consensus

- Inputs are local state projections
- Data are fault-arrive probabilities and state-collection times
  - Occurrence frequencies
- Outputs are consensus violations
Monitor Architecture Requirements

- **Functionality**: the monitor does not change the functionality of the *system under observation* (SUO), unless the SUO violates its specification.
  
  - **Unintentional**: safe-guards must be in place to ensure that monitor faults do not affect the SUO’s functionality
  
  - **Intentional**: the monitor must signal a reset, etc. to the SUO only if the SUO has (probably) violated its specification
Monitor Architecture Requirements

- **Schedulability**: the monitor architecture does not cause the SUO to violate its hard real-time guarantees.
- **Reliability**: the reliability of the SUO in the context of the monitor architecture is greater or equal to the reliability of the SUO alone.

A monitor might reduce the SUO’s reliability for some class of faults of (improbable) faults and yet increase the system’s overall reliability.
Architectural Considerations

• Where does the monitor “go”?
  • Two architectural approaches:
    – **Distributed**: monitors at the distributed nodes, and interchange “consensus data”
    – **Central**: nodes send “consensus data” to a central monitor.
  
• Resulting in various reliability/cost tradeoffs

• Want to be able to synthesize multiple architectures
Monitor Bus Traffic

- Can be applied to legacy systems
  - BusMoP
  - Internet traffic monitoring
  - No instrumenting code needed
- Cannot detect faults beyond level of SUO itself
Dedicated Bus Single Monitor

- Code must be instrumented to copy state information to the monitor bus
  - Must ensure this does not affect timing
Distributed Monitors

- Monitors are local to each node
- Exchange values to monitor consensus
- This architecture can be used to **bolt on** byzantine fault tolerance
Monitor Software Requirements

• Since the SUO is a real-time embedded system
  • The monitor must run in constant time
  • The monitors must run in constant space
    – No dynamic memory allocation (No garbage collector)
• The monitors should be written in a language with low-level control over time and memory
  • Most likely will be written in C
Atom DSL

- Writing monitors in a modern high-level language such as Haskell is less error prone and closer to a specification than C
- Atom: Subset of Haskell → Embedded C
- Designed to synthesize the embedded real-time control systems for off-road vehicles
  - Tom Hawkins of Eaton
  - Designed to automate aspects of scheduling and synchronization
- Initial experiments are promising
Future Work

- Designing a specification language tailored for consensus properties
  - Spec $\rightarrow$ Atom $\rightarrow$ C
- Pick a platform for monitoring
- Monitors detecting and diagnosing faults in consensus
  - Real or transient faults
Consensus Properties

- A monitor can observe consensus (or the lack thereof) between distributed components.
- This principally means observing classes of asymmetric or Byzantine faults (including omissive faults).
- It appears that Byzantine faults are also the most “malicious” and least accounted-for faults.
- Example: non-universal I/O error in the Shuttle!
- Monitors are bound by the “laws” of distributed-system observation (given real-time clocks). This means there's some probability of false-positives and false-negatives.

Example:
Consensus Properties: Timing

- Constraints talk about real-time (i.e., wall-clock time).
- For example: here’s a clock drift-rate constraint:

\[(1 - \rho) \cdot (t_1 - t_2) \leq C'(t_1) - C'(t_2) \leq \[(1 + \rho) \cdot (t_1 - t_2)\]\]

- But violations of constraints will manifest themselves as systematic faults (i.e., greater than the expected fault-arrival rates)
- And faults are likely to be slightly-out-of-spec timing errors
- Challenge: determining when a fault is frequent enough to be a systematic fault
- Techniques for probabilistic runtime checking in soft real-time systems are applicable

\[\text{[Sammapun, Lee, Sokolsky, Regeher. Statistical runtime checking of probabilistic properties, RTV'07.]}\]
Fault-Tolerance Definitions

- **Failure** – system unable to provide required functions
- **Error** – system state liable to lead to subsequent failure
- **Fault** - Adjudged cause of error
Central Monitor Shared Memory

- Central monitor uses shared variables to observe system state
- The monitors only watch
  - Unless steering in response to detected fault
- Currently under study
Consensus Properties: Timing

Violated timing assumptions

Hard realtime systems have timeliness guarantees, provided system timing assumptions hold

- The timing assumptions are constraints on clock drift, skew, message delays, resynchronization, etc
- Constraints cannot be monitored directly
  - (A monitor has no more access to real-time than the what's monitored.)
  - Techniques for probabilistic runtime checking in soft real-time systems are applicable

[Sammapun, Lee, Sokolsky, Regeher. Statistical runtime checking of probabilistic properties, RTV'07.]