External Runtime Monitoring for Critical Embedded Systems

Phil Koopman, CMU
Aaron Kane, CMU ECE
Mike Wagner, CMU NREC

(Research Funding from General Motors)

Overview

- Goal: Assure critical properties met
  - Especially cost-sensitive, adaptive systems
  - Large deployed fleets; everyday products (e.g., self-driving cars)
  - How can you demonstrate that they are met?
- Discussion of available building blocks
  - Testing, formal methods, runtime verification
- Our current area of work:
  - External runtime monitors – an attempt to go for cross-area low-hanging fruit

Problem Statement

- How do you make sure "robots" are safe?
  - Especially including autonomous vehicles
  - (Big ideas likely apply to all embedded systems)
- Need to take into account:
  - Significant cost, time-to-market constraints
  - Continual changes to software code base
  - Increasing complexity
  - Likely gaps in/lack of rigorous design artifacts
- Reality check: They're going to be built with us or without us. **How can our community be relevant?**

Approach: Testing

- Strategy: Test it into submission
  - Find the bugs; test some more; system-level testing
  - In industry, this is the default strategy
- Strengths:
  - There's nothing like the real thing
  - Historically works OK on non-software systems
- Weaknesses:
  - Need to test at least 3x MTBF – problem when MTBF is comparable to total fleet exposure
  - Need to recertify after even one line of code has been changed
  - Hard to test failure modes (e.g., need fault injection)
- Possible way to improve:
  - Use testing to validate quality rather than create quality

Peer Review

- Strategy: Inspection of design artifacts
- Strengths:
  - Expect to find 50% of the bugs for 10% of budget
- Weaknesses:
  - Management bias to create functionality, not do reviews
  - Usually better at unit level than system level
  - Informal; monitoring bug find rate helps assess effectiveness
  - Many designers are bad at imagining failure modes in a review
- Possible way to improve:
  - Can we say something stronger about review coverage?
  - Better techniques for system integration review

Static Analysis

- Strategy: Lint-like tools to analyze source code
- Strengths:
  - Helps find implementation problems
  - (Dynamic analysis may help too, e.g., bounds checking)
- Weaknesses:
  - Only good for narrow implementation problems
  - False positives unless adopt a lint-friendly coding style
- Possible way to improve:
  - Higher level static analysis (e.g., at architecture level) – some work in this area

Your List of Favorite Informal Analysis Techniques Goes Here

- Robustness testing & fault injection
- …

Formal Representations

- Strategy: Mathematically rigorous expression of specification and a model of system
- Strengths:
  - Mathematically rigorous; helps think about system
- Weaknesses:
  - Assumptions necessary in proofs may be significant
  - Need to ensure specs & system model are correct
  - Scalability problems to cars with 1M+ lines of code
  - Temporal aspects can be challenging
- Possible way to improve:
  - Improve accessibility to everyday engineers;
    "light weight" approaches to temporal properties

An Aside on Specifications

- Multiple representations of a system:
  - System specification: what it does
  - System model: how it is built
- But, it is usually unnecessary to prove the system is perfect
  - Really, what you care about is only the critical aspects of system behavior
  - ➔ Want a "safety specification"
    - (Or "critical property specification")
  - In practice, subset of system spec doesn't work – Need an entirely different safety spec

Model Checking

- Strategy: Prove properties about formal representations, e.g. via exhaustive search
- Strengths:
  - Mathematically rigorous; provides counter-examples
  - Impressive gains in scalability using SAT solvers
- Weaknesses:
  - Need to know what questions to ask ("safety spec")
  - Same general pro/con as formal representations
- Possible way to improve:
  - Biggest challenges (IMHO): adaptive systems and modeling faulty system behavior

Other Formal Analysis

Your list of favorite formal static techniques goes here…

- Design synthesis from formal specification
  - Model-based design
- …

Acceptance Test Style Techniques

- Strategy: Check for correctness at run time
- Strengths:
  - System being tested is the real system – warts and all
  - “Checking” can often be simpler than “doing”
- Weaknesses:
  - Need to know what properties to check (“safety spec”)
  - Doesn’t “prove” anything
    - (Not strictly true… proves that the test’s behavior trace is OK)
- Possible way to improve:
  - Formalize the acceptance tests
  - Architectural patterns to separate doing from checking

Safety Kernels

- Strategy: Safety kernel blocks unsafe actions (“safety gate” architecture pattern)
- Strengths:
  - Works on the real system, not just a model
  - Operating system provides some isolation
- Weaknesses:
  - Same as acceptance tests
  - Must predict effect of action on system to work
  - Need to recertify kernel? (How good is isolation?)
- Possible way to improve:
  - Stronger isolation to avoid recertification

Runtime Monitoring

- Strategy: Trigger a flag when system misbehaves at runtime (“safety monitor” architecture pattern)
- Strengths:
  - Similar to safety kernel
  - Doesn’t need to predict; just react
- Weaknesses:
  - Technically, system is momentarily unsafe when fault detector triggers
- Possible ways to improve:
  - Physically isolate from system to avoid recertification
  - Design systems to explicitly permit bounded-time failure detection

Other Runtime Verification

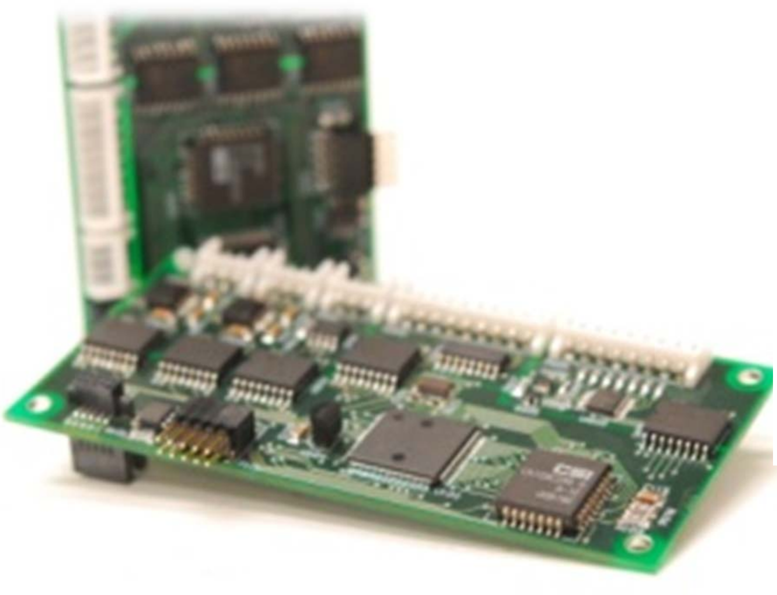
- Your list of favorite runtime techniques goes here…
- Mechanical interlocks and safety monitors
  - Historically useful, but often too simple to permit optimized control behaviors
- My favorite is: External runtime monitoring

External Safety Monitor

- Idea: External runtime monitor
  - Formal (or semi-formal) safety specification
  - System presents state information
  - Monitor checks sate against safety spec at run time
- We’re going to sweep recovery under the rug
  - For now, concentrating on a real time failure detector – e.g. to trigger emergency shutdown
  - For example, to provide fail-stop subsystem behaviors

Run-Time Safety Monitor

- How do you know this unmanned ground vehicle is safe?
  - Ensure speed limit not violated
  - Ensure it stays stopped when commanded to stop
  - But, autonomy software has been modified at 3 AM on demo day(!)
- Solution: independent safety monitor
  - This is the one thing you can count on



TARGET GVW: 8,500 kg
TARGET SPEED: 80 km/hr
Approved for Public Release. TACOM Case #20247 Date: 07 OCT 2009

Safety Monitor Approach

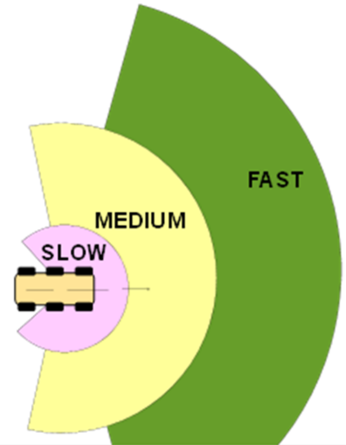
- Dedicated, trusted hardware to monitor behaviors
  - Invariants to describe “safe” behaviors
    - For example: vehicle speed < speed limit
  - State machines to account for system operating modes
    - Different invariants are active in different modes (e.g., “stop” vs. “run”)
  - Emergency shutdown sequencing if any invariant is false

APD Safety System

Objective: Enforce and control safe standoff distance between APD and nearby personnel.

Approach:

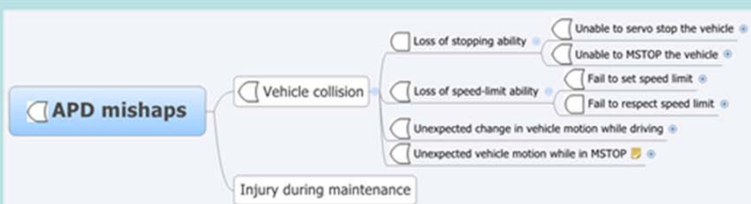
- Provide fail-safe braking mechanisms with well-modeled stopping distance.
- Incorporate Safety Monitor for redundant, high-reliability means of restraining vehicle speed.
- Identify and mitigate risks that could lead to failures of braking and speed-limiting.



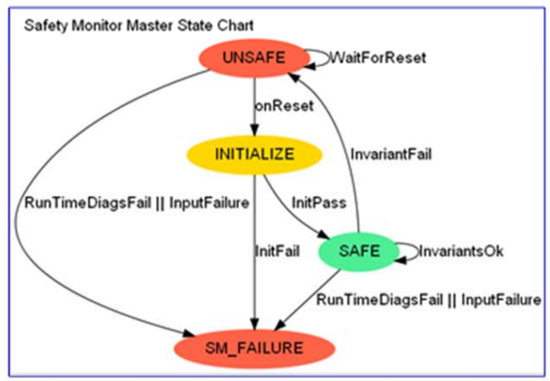
Reliable speed limiting allows safe standoff distances to be decreased

Techniques:

- Identifying hazards that lead to safety mishaps.
- Modeling of correlation between latent hazards with rich instrumentation.
- Firewalling safety-criticality to a subset of vehicle components.
- Developing & testing fault-resistant software for speed limiting.
- V&V testing traced to safety requirements.



Careful analysis of mishaps drives safety system design



Safety Monitor ensures that safety invariants are maintained

APPROVED FOR PUBLIC RELEASE, TACOM CASE #20094, DATE: 17 AUG 2009

Automotive Prototype

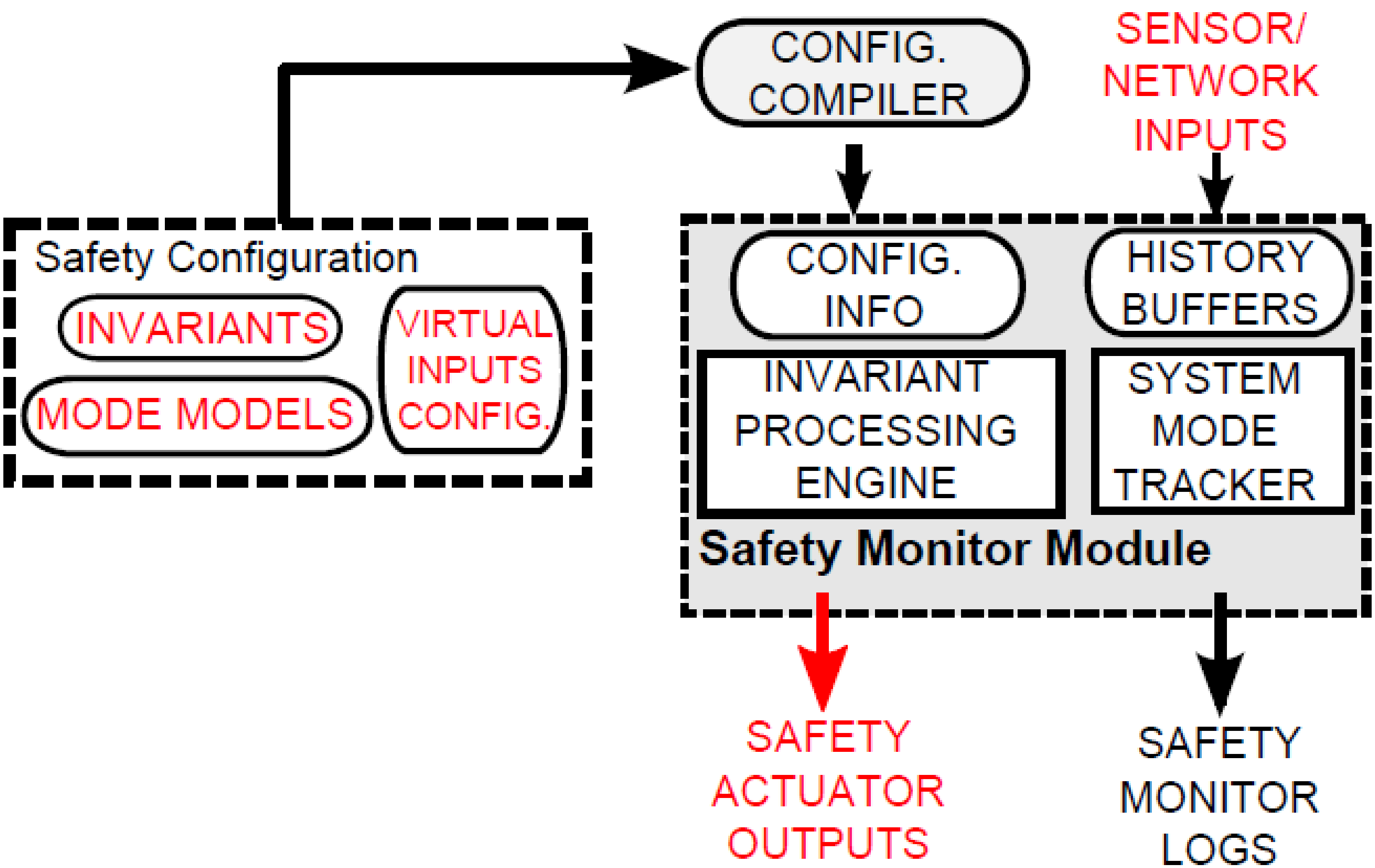
- Laptop based monitor
  - Log data for offline monitoring
  - Run-time monitor with alert
    - Can trigger commands
  - OBD-II and UDP networks

- Watching system level properties
  - Not monitoring individual subsystems

Prototype Safety Specification

- Invariants
  - Syntax based on bounded real-time linear temporal logic   ("Metric Temporal Logic")
- Modes
  - State machines
  - Hold contextual system state
- Virtual Inputs
  - Abstraction to simplify policy

Generic External Runtime Monitor



Simplified Invariant Language

```
rule ::= G -> P
G ::= expression
P ::= expression | temp_op expression
expression ::= extended_java
temp_op ::= [timestep, timestep]  | <timestep, timestep>
timestep ::= integer
```

Figure 2: Current Prototype Specification language

Pattern Name	Bounded A triggers B
Description	If A occurs then B must occur within t time of A
Logic	$A \rightarrow \Diamond_{0,t} B$
ASCII	`A -> <0,t> B`

Figure 3: Example Pattern for Building Safety Specifications

The Good Parts

- Can get (semi-)formal “proofs” of test runs
  - Even if a fault in the system is present
- Don’t need to build a system model
  - The vehicle itself is the “model”
  - “Free” modeling of implementation defects
- Minimally intrusive
  - Separate test box doesn’t affect system
  - If monitor provides safe shutdown, don’t need to recertify rest of system after a change
  - If monitor is test oracle, don’t need to change

The Challenging Parts

- Ensuring coverage
  - Still need to fault inject during testing
- How do you know the safety spec is right?
  - But you have to know that regardless…
- How do you know you can see sufficient internal state?
  - For now this has worked out well, but need more experience to understand this

What We've Learned About Time

- Need simple temporal approach
  - Simple MTL (represents a few cycles of time)
    - Need "always" over a bounded time
    - Need "eventually" over a bounded time
    - Everything else is linear; state machines help a lot
- Need to look at time a little differently
  - "Past time" instead of future time for monitoring
    - Safety kernel would require looking ahead a bit
  - What does "eventually" mean at run time?
  - Need to compress and bound history to avoid keeping all data since system was turned on

Other Things We've Learned

- Embedded systems are highly modal
  - Mode dramatically affects what "safe" means
  - Our approach: use state machines
    - Need to infer system modes based on outputs
    - Also use to compress system history
- Need to consider reliability of sensor info
  - Our approach: minimal redundant sensors that do sanity checks on primary sensors
- Designers are generally allergic to special symbols
  - But, when you find things in a real system, they pay attention!