Software Robustness Testing and Run–Time Monitoring of Autonomous Vehicles

Prof. Phil Koopman
koopman@cmu.edu

This material is based upon work supported by the Test Resource Management Center (TRMC) Test and Evaluation/Science & Technology (T&E/S&T) Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W900KK–11–C–0025, “Stress Testing for Autonomy Architectures (STAA)”.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Test Resource Management Center (TRMC) Test and Evaluation/Science & Technology (T&E/S&T) Program and/or the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI).
Overview

- Very brief CMU overview

- Autonomous vehicle & robotic software safety
  - Goes beyond current software safety standards

- Automated robustness testing
  - Finds significant software defects

- Run-time safety monitors
  - Used on large autonomous vehicle to ensure safety

- ASTAA project: automated stress testing of robots
  - ASTAA = Robustness stress testing + simple safety monitors

- Some future challenges
  - Getting from demos to full scale deployment will be hard!
ECE Department:
~100 Faculty
~150 undergrads/yr
~500 grad students
(Note: Computer Science is a whole school)
National Robotics Engineering Center

DARPA SC–ALV
NASA Lunar Rover
Auto Excavator
Auto Forklift
DARPA PerceptOR
DARPA LAGR
DARPA UPI
Army FCS
Laser Paint Removal

1985
1990
1995
2000
2005
2010

ARPA Demo II
NASA Dante II
Auto Harvesting

DARPA Grand Challenge
Mars Rovers
Urban Challenge
Auto Haulage
Auto Spraying

NREC:
~175 Faculty, staff, students
Off-campus Robotics Institute facility, SCS Engineering & Technology Transfer

© 2014 Carnegie Mellon University, all rights reserved.
How Well Tested Are Autonomy Features?

USDOT Proposes Updated Safety Standard to Prioritize Braking Control, Reduce Risk of High-Speed Unintended Acceleration for Nation's Cars

NHTSA 04-12
April 12, 2012
Contact: Karol Aldana, 202-366-9550

‘Brake-Throttle Override’ requirement will reduce the risk of high-speed unintended trapped accelerator pedal

Jaguar recalls 18,000 cars over cruise control software fault

Car system upgrade needed, but no hardware affected

By Leo King | Computerworld UK | Published: 10:23, 24 October 11

Jaguar has recalled nearly 18,000 X-type cars after it discovered a major software fault, which meant drivers might not be able to turn off cruise control.

The problem lies with engine management control software developed in-house by Jaguar. The problematic software is only installed on diesel engine X-Type's, which were all produced between 2000 and 2010.

Some 17,678 vehicles have been recalled, as a result of the potentially dangerous problem: if the fault occurs, cruise control can only be disabled by turning off the ignition while driving – which would mean a loss of some control and in many cars also disables power steering.

Braking or pressing the cancel button will not work.

"Jaguar has identified that an error with certain interfacing systems be detected the cruise control system will be disabled and an error message displayed to the driver on the instrument cluster," the company said in a statement.

Self-driving cars now legal in California

By Heather Kelly | CNNMoney | Updated: 12.30 AM EDT, Tue October 30, 2012 | Filed under: innovations

Drivless car now legal in California

[On Tuesday afternoon] along with Google co-founder Sergey Brin and State Sen. Alex Padilla, who authored the bill, at Google's headquarters in Mountain View, California. The bill, SB 1290, will set up procedures and requirements for determining when the cars are road-ready.

Brin hopes that self-driving cars will be able to drive on public streets in five years or less.
Testing Isn’t Enough To Ensure SW Safety

- In current systems, system-level testing is useful and important
  - It can find unexpected component interactions
- **But**, it is impracticable to test everything at the vehicle/system level
  - There are too many possible operating conditions
  - There are too many possible timing sequences of events
  - There are too many possible faults
  - All possible combinations of component failures and memory corruptions
  - Multiple software defects activated by a sequence of operations

[Koopman 2013]
Robot Testing Is Even More Difficult

Test coverage over high-dimensional inputs

Sensitivity to calibration

Adaptive systems

Non-linear motion planning

Validation of machine learning results

© 2014 Carnegie Mellon University, all rights reserved.
[Wagner/Koopman]
Fuzz testing [Miller98] uses a random input stream
• Finds interesting failures
• But can be inefficient

Ballista (1996..2008) uses “dictionaries” of values
• Combinations of exceptional and ordinary values
• More efficient, but still scalable, approach to robustness testing
Generates test cases based on parameter data types

- Ignoring functional ‘correctness’ provides scalability

API: \texttt{write(int filedes, const void *buffer, size_t nbytes)}

<table>
<thead>
<tr>
<th>TESTING OBJECTS</th>
<th>TESTING VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESCRIPTOR</td>
<td>BUF_SMALL_1, BUF_MED_PAGESIZE, BUF_LARGE_512MB, BUF_XLARGE_1GB, BUF_HUGE_2GB, BUF_MAXULONG_SIZE, BUF_64K, BUF_END_MED, BUF_FAR_PAST, BUF_ODD_ADDR, BUF_FREED, BUF_CODE, BUF_16, BUF_NULL, BUF_NEG_ONE</td>
</tr>
<tr>
<td>TEST OBJECT</td>
<td>SIZE_1, SIZE_16, SIZE_PAGE, SIZE_PAGEX16, SIZE_PAGEX16plus1, SIZE_MAXINT, SIZE_MININT, SIZE_ZERO, SIZE_NEG</td>
</tr>
</tbody>
</table>
Ballista Found Plenty of Robustness Issues!

Ballista Robustness Tests for 233 Posix Function Calls

 normalized failure rate

[Abort Failures]

[Restart Failure]

1 Catastrophic

2 Catastrophics

© 2014 Carnegie Mellon University, all rights reserved.
[Koopman / Ballista]
Abort Failures Might Predict Bad Software Quality

- "Abort" failures are a core dump
  - Individual process crash rather than system crash
  - Whether a process crash matters depends upon your system & philosophy

- Most failures found were highly repeatable, "one-liner" calls
  - Not race conditions (surprise!)
  - Not long complex sequences (surprise!)

- HP-UX gained a system-killer in upgrade from Version 9 to 10
  - In newly re-written memory management functions… … which had a 100% failure rate under Ballista testing!
RTI–HLA Simulation Backplane/Middleware

Robustness Failures of RTI 1.3.5 for Digital Unix 4.0

STATED GOAL OF HLA: 100% Robust
Stress Testing Finds Bugs On Robots Too…

Important vulnerabilities have been found in over twenty systems tested on our project so far

... more to come...
CRUSHER
But, safety standards might not apply:
(Example from IEC–61508)

<table>
<thead>
<tr>
<th>Technique/measure</th>
<th>Ref</th>
<th>SIL</th>
<th>Interpretation in this application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Fault detection and diagnosis</td>
<td>C.3.1</td>
<td>HR</td>
<td>Used as far as dealing with sensor, actuator and data transmission failures and which are not covered by the measures within the embedded system according to the requirements of IEC 61508-2</td>
</tr>
<tr>
<td>2 Error detecting and correcting codes</td>
<td>C.3.2</td>
<td>R</td>
<td>Only for external data transmissions</td>
</tr>
<tr>
<td>3a Failure assertion programming</td>
<td>C.3.3</td>
<td>R</td>
<td>Results of the application functions are checked for validity</td>
</tr>
<tr>
<td>3b Safety bag techniques</td>
<td>C.3.4</td>
<td>R</td>
<td>Used for some safety related functions where 3a and 3c are not used</td>
</tr>
<tr>
<td>3c Diverse programming</td>
<td>C.3.5</td>
<td>R</td>
<td>Used for some functions where source code is not available</td>
</tr>
<tr>
<td>3d Recovery block</td>
<td>C.3.6</td>
<td>R</td>
<td>Not used</td>
</tr>
<tr>
<td>3e Backward recovery</td>
<td>C.3.7</td>
<td>R</td>
<td>Not used</td>
</tr>
<tr>
<td>3f Forward recovery</td>
<td>C.3.8</td>
<td>R</td>
<td>Not used</td>
</tr>
<tr>
<td>3g Re-try fault recovery mechanisms</td>
<td>C.3.9</td>
<td>R</td>
<td>Not used</td>
</tr>
<tr>
<td>3h Memorizing executed cases</td>
<td>C.3.10</td>
<td>R</td>
<td>Not used (measures 3a, 3b and 3c are sufficient)</td>
</tr>
<tr>
<td>4 Graceful degradation</td>
<td>C.3.11</td>
<td>HR</td>
<td>Yes, because of the nature of the technical process</td>
</tr>
<tr>
<td>5 Artificial intelligence - fault correction</td>
<td>C.3.12</td>
<td>NR</td>
<td>Not used</td>
</tr>
<tr>
<td>6 Dynamic reconfiguration</td>
<td>C.3.13</td>
<td>NR</td>
<td>Not used</td>
</tr>
</tbody>
</table>
APD (Autonomous Platform Demonstrator)
How did we make this scenario safe?

TARGET
GVW: 8,500 kg
SPEED: 80 km/hr
The Autonomous Platform Demonstrator (APD) was the first UGV to use a Safety Monitor as part of its safety case.

As a result, the U.S. Army approved APD for demonstrations involving soldier participation.

U.S. Army cites high quality of APD safety case and turns to NREC to improve the safety of unmanned vehicles.

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.

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

© 2014 Carnegie Mellon University, all rights reserved.
Approved for Public Release – Distribution is Unlimited
(NREC case #: STAA-2012-10-17)
How Can We Combine These Ideas?

**Ballista Stress-Testing Tool**
Robustness testing of defined interfaces
• Most test cases are exceptional
• Test cases based on best-practice software testing methodology
• Detects software hanging or crashing

*Earlier work looked at stress-testing COTS operating systems*

*Uncovered system-killer crash vulnerabilities in top-of-the-line commercial operating systems*

**NREC Safety Monitor**
Monitors safety invariants at run-time
• Designed as run-time safety shutdown box for UAS applications

Independently senses system state to determine whether invariants are violated

*Firewalls safety-criticality into a small, manageable subset of a complex UAS; prototype deployed on Autonomous Platform Demonstrator (APD), a 9-ton UGV capable of reaching 80 km/hr*
The ASTAA Project

- Automated Stress Testing of Autonomy Architectures
  - Three-year project sponsored by the Test Resource Management Center within the Office of the Secretary of Defense
  - The project continues through September 2014

- Project goals:
  - Use automatic software stress-testing to uncover safety problems in unmanned systems that wouldn’t otherwise be found during system testing
  - Implement testing tools that interface with software components in an unobtrusive way
Mature (6 years old) “RECBot” vehicle tested with initial tool set
- No access to source code or design details; just interface specification
- ASTAA elicited a speed-limit violation

Distribution Statement A - Approved for public release; distribution is unlimited. NAVAIR Public Affairs Office tracking number 2013–74, NREC internal case number STAA–2012–10–23
ASTAA Workflow

Existing Documentation
- Safety Requirements
- System Design (IDD)
- Message Dictionary (ICD)

ASTAA Test Specification
- Invariants & Interfaces (XML)

Test Generator

Test Runner

System Under Test

Test Cases (XML)

Test Results (XML)
## Methods of test execution

<table>
<thead>
<tr>
<th>Injection with log replay or running component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleaving during injection</td>
</tr>
<tr>
<td>Interception</td>
</tr>
</tbody>
</table>

### Injection

<table>
<thead>
<tr>
<th>Injection</th>
<th>ASTAA</th>
<th>SUT</th>
</tr>
</thead>
</table>

### Injection with log replay or running component

<table>
<thead>
<tr>
<th>Injection with log replay or running component</th>
<th>ASTAA PM</th>
<th>Component B</th>
<th>SUT</th>
</tr>
</thead>
</table>

### Interleaving during injection

<table>
<thead>
<tr>
<th>Interleaving during injection</th>
<th>ASTAA</th>
<th>Component A</th>
<th>Component B</th>
<th>SUT</th>
</tr>
</thead>
</table>

### Interception

<table>
<thead>
<tr>
<th>Interception</th>
<th>Component A</th>
<th>Component B</th>
<th>SUT</th>
</tr>
</thead>
</table>
Example: CAN / J1939 interception

In this example:

- **CAN Interceptor**
  - Isolates actuators from ECU by splitting the CAN bus
  - Modifies J1939 status messages from by-wire controllers before forwarding to ECU
  - Reads messages for invariant evaluation

- **ASTAA Test Runner**
  - Instructs CAN interceptor about how to modify incoming CAN messages
  - Monitors invariants

---

© 2014 Carnegie Mellon University, all rights reserved.

DISTRIBUTION A – NREC case number STAA-2013-10-02
An invariant is an expression involving SUT state that takes the form of a guard and predicate ("FAIL" or "WARN")

State machines track the system’s state
  • Transition guards are inputs from the SUT
  • Each state activates potentially different invariants
Automated Stress-Testing for Autonomy Architectures

Test Specification and Execution Overview

- Define ASTAA Test Specification (guided manual process)
- Execute Test Generator (automated process)
- Execute Test Cases with Test Runner (automated process)

ASTAA Test Spec (XML)
- Interface Definition
  - Ports & Protocols
  - Message Dictionary
  - Parameter 1 (int)
  - Parameter 2 (bounded int)
- Message Frame 1
- Message Frame 2
- Parameter 1 (float)
- Parameter 2 (speed)

Exceptions Database
- Invariants
- Types
- Exception List:
  - Inv1: Param1 <= PLimit1
  - Inv2: <Condition>

Test Generator
- Message Types
- Type Exceptions

Test Case (XML)
- Invariants Definition
- Constructors
- Test Command Sequence
- Destructors
- Interface Definition

Test Command Sequences

Test Cases (XML)

ASTAA Test Runner
- Invariant Monitors
- Test Injectors
- Orchestrator
- GUI

Module Manager
- Protocol Module

System Under Test
- Safety Monitor (optional)
Types of components tested so far

- Communications: Message serialization and routing
- Control: motion control, I/O
- Perception: terrain perception, terrain classification, obstacle detection, map building
- Planning: path tracking, motion planning, obstacle avoidance

Stress testing finds bugs in autonomy software
  - Over 50 vulnerabilities have been found in over twenty systems tested on our project so far
Root causes of robustness vulnerabilities include...

- **Improper handling of floating-point numbers**
  - Failure to handle exceptional values (e.g., NaN, Inf)
  - Normalization of floating-point angles

- **Array indexing and allocation**
  - E.g., images, point clouds, evidence grids
  - Segmentation faults due to arrays that are too small
  - Many forms of buffer overflow, especially dealing with complex data types
  - Large arrays and memory exhaustion

- **Time**
  - Time flowing backwards, jumps
  - Not rejecting stale data

- **Problems handling dynamic state**
  - E.g., lists of perceived objects or command trajectories
  - Race conditions permit improper insertion or removal of items
  - Vulnerabilities in garbage collection allow memory to be exhausted or execution to be slowed down

- **Assertions that have not been disabled**
The Ballista/ASTAA Team

System Safety for Autonomous Robots (2008 – )
Automated Stress Testing of Autonomy Architectures (2011 – )

A Ballista is an ancient siege weapon for hurling large projectiles at fortified defenses.
Making “Easier” Systems Safe

- Elevators
  - Building codes describe required mechanisms
  - Electromechanical safeties (avoid trusting SW)

- Rail systems
  - Dual redundant hardware protection systems
  - Rigorously developed software EN-50126/8/9
    - Customers typically require these standards
    - “Safety net” architecture minimizes critical SW
  - Fail-stop approach – shut down if unsafe
Why HW Safety Is Difficult

- “Safe” might be 1e-9/hr catastrophic failures
  - (It is easy to argue cars must be safer than that)
  - Single fatalities at perhaps 1e-7/hr (probably less)
- Simplex hardware tends to fail at 1e-5 to 1e-6/hr
  - Cosmic rays result in bit flips (yes, really!)
  - Other things go wrong at about this rate
- Thus, need **redundancy** to be safe
  - No single point failure end-to-end in the system
  - Takes some effort to get redundant components to properly synch.

- **Infeasible to test** to 1e-9/hr
  - Need testing time 3x-10x longer than failure rate

© 2014 Carnegie Mellon University, all rights reserved.
[Koopman 2014, Transportation CPS Workshop]
Making “Harder” Systems Safe

Aviation
- Do-178 and other FAA standards
- Federal certifying agency (FAA)
  - Testing + examination of how system is designed
- Fail operational; significant redundancy

Automotive
- NHTSA does not proactively certify safety
  - FMVSS don’t really address SW safety
- Some redundancy; tough cost constraints
  - Steering & brakes must fail (partially) operational
- MISRA Guidelines ➔ ISO 26262 safety standard
  - But neither is really intended to cover autonomous vehicles

[Koopman 2014, Transportation CPS Workshop]
Why SW Safety Is Difficult

- Testing does not make software safe!
  - You can’t test all SW corner cases
  - Proving correctness is not enough for safety either
    - How do you know your requirements are correct?
    - Have you proven correctness under all fault conditions?

- Software safety requires process in addition to testing
  - Follow standards (e.g., ISO 26262)
    - List of practices based on SW criticality
    - Ensure development process quality
  - Testing checks you really did it right
    - Testing is not “debugging” – test for absence of bugs
  - Adaptive/robot software can go beyond existing SW safety
The World Is Full Of Unexpected Situations...

- Extreme contrast
- No lane infrastructure
- Poor visibility
- Unusual obstacles
- Construction
- Water (note that it appears flat!)

So just getting all the obvious cases covered is challenging

© 2014 Carnegie Mellon University, all rights reserved.

[Wagner 2014]
NOBODY Has Seen It ALL!
Autonomy Validation Challenges

- Specifying safety
  - Artfully select subset of functionality to equal safety
  - Need a realistic role for human operator

- Unconstrained environments
  - Uncontrolled, unpredictable urban roadways
  - Can inductive-based algorithms cover enough corner cases?

- Trusting validation
  - How do you know you are really safe?
  - How do you know someone else’s system is really safe when you cooperating with it?
Questions?