Test and Evaluation of Autonomous Systems in a Model Based Engineering Context

Raytheon
Michael Nolan

USAF AFRL
Aaron Fifarek
Jonathan Hoffman
3 March 2016
Agenda

• Motivation
• Trust and Certification Process
• Background
• Formal Analysis
• Requirements Analysis
• Architecture
• Model Traceability
• SysML Representation of Autonomous System and Autonomous System Development
• Basic example of Autonomous Systems T&E in MBE context
• Summary
Motivation

Introduction, Discovery, and Cost of Software Faults

- Identified Need
- Requirement Development
- Architecture Development
- Detailed Design
- Implementation
- Integration
- Verification
- Validation
- Transition

70% of faults are introduced
3.5% faults are found
1x estimated nominal cost for fault removal

20% of faults are introduced
16% faults are found
5x estimated nominal cost for fault removal

10% of faults are introduced
59.5% faults are found
20-80x estimated nominal cost for fault removal

20.5% faults are found
300-1000x estimated nominal cost for fault removal

Opportunity to find faults as they are introduced when costs are low


Rework and certification is 70% of SW cost. 

Trust and Certification
Products / Process

Design
• Requirements
• Architecture
• Models

Validation
Simulation
Testing

New Autonomy Need

System Design and Safety Requirements
(ARP 4761, ARP 4754/A, MIL-HDBK-882E)

Certified Assurance Case

Testable Requirements & Verification Plans
(DO-178C/254, MIL-HDBK-516)

Multiple V&V Technology Paths

Compositionally Verified Systems of Systems

Formalized Safety Assessment
Hazard Mitigation
Requirements

Design & Safety Certification
System of Systems Certification
**Formal Analysis**

**Formal Methods** refers to *mathematically rigorous* techniques and tools for the specification, design and verification of software and hardware systems.


• What is Formal Analysis?
  – Analysis performed on mathematically precise models utilizing elegant Computer Science algorithms and tools
    • Model-Checking
    • Theorem Proving

• Why do we want to do it?
  – We can exhaustively search the behavior of models to prove or disprove desired properties
  – Removal of ambiguity due to required mathematical rigor
  – Can identify unintended and unspecified behaviors
Analysis
Advantage of Model Checking

Testing Checks Only the Values We Select

Even Small Systems Have Trillions (of Trillions) of Possible Tests!

Model Checker Tries Every Possible Value!

Finds every exception to the property being checked!
Requirements Development & Analysis

Precise, structured standards to automate requirement evaluation for testability, traceability, and de-confliction

- Requirements
  Understanding the problem
- Architecture
  Outlining the solution
- Model
  Demonstrating the implementation of the solution
Formal Requirements Analysis

• Natural language requirements are difficult to process logically and mathematically especially if they are not written with a formal basis
  
  “The flight control function that performs the automatic avoidance maneuver shall be of a level of redundancy equivalent to the primary flight control system”

  • What is the formal definition of this constraint on the system?
  • Not a trivial definition on the system

**Formal Methods** refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.


Temporal logic definitions are not obvious to write for most individuals and takes years of practice to master effectively

<table>
<thead>
<tr>
<th>(p → a)</th>
<th>What does that mean?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p → □ a)</td>
<td>There may be logical basis but it’s not accessible to others.</td>
</tr>
<tr>
<td>(p → (¬b U ((a ∨ ¬p) ∨ □¬b)))</td>
<td></td>
</tr>
<tr>
<td>(p → (((b → (p U (a ∧ p))) U (¬p ∨ □((b → (p U (a ∧ p))))))))</td>
<td></td>
</tr>
</tbody>
</table>
Formal Requirements Analysis

- Our Approach – *Pattern Implementation*
  - Constrain natural language to patterns which contain a scope and a predicate
  - Enforces the formal basis necessary to ensure mathematical rigor

- Can requirements be defined and verified compositionally?

<table>
<thead>
<tr>
<th>Occurrence</th>
<th>Absence</th>
<th>Universality</th>
<th>Existence</th>
<th>Bounded Existence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>Precedence</td>
<td>Response</td>
<td>Chain Precedence</td>
<td>Chain Response</td>
</tr>
</tbody>
</table>

While the pump is ON the pump outflow shall be the maximum flow rate.

**Scope** | **Predicate**
--- | ---
*while (pump_state == ON) :: always (pump_flow == MAX_FLOW)*
Architecture

Guarantee appropriate decisions with traceable evidence during the system architectural design
Architecture: AADL and AGREE

- The Architecture Analysis & Design Language (AADL)
  - Developed by SAE
  - Architecture modeling notation with well-defined semantics

- Assume Guarantee REasoning Environment (AGREE) plugins
  - Developed by University of Minnesota and Rockwell Collins
  - Part of the DARPA High-Assurance Cyber Military Systems (HACMS) program

Assumptions: something a system assumes about its environment (inputs)
Guarantees: what you can assume about the system and the performance of the system (outputs)

• **Assume-Guarantee Contract** - Verifiable set of Assumptions and Guarantees that abstracts the behavior of a system component implementation

• **Assumptions**
  
  Constraints over what a component expects to see from its environment

• ** Guarantees**
  
  Constraints over how a component behaves in response to its environment
Compositional Verification

- A series of techniques to allow for systems to be decomposed into less complex modules to be enforce a hierarchical structure that can be leveraged for compositional techniques

- Systems can be hierarchically organized\(^1\)
  - Requirements vs. architectural design must be a matter of perspective
  - Need better support for \(N\)-level decompositions for **requirements** and **architectural design**

---

Model Development

Cumulative Evidence Through Research, Developmental, and Operational Test

- Requirements
  - Understanding the problem

- Architecture
  - Outlining the solution

- Model
  - Demonstrating the implementation of the solution

CUMULATIVE EVIDENCE THROUGH RDT&E, DT & OT
Progressive sequential modeling, simulation, test and evaluation
• Uses formal methods to find **violations of design properties and assumptions**

• Formal Analysis techniques from:
  – Prover Plug-In
  – Polyspace formal analysis engine from MathWorks
SLDV Analysis

Property Model

%% Guarantees (Proof Objectives)
% GO1: guarantee "The pump is initially off"
g1 = pump_initially_off(pre_Pump_State, time);
function result = pump_initially_off(Pump_State, time)
UnderThisCondition = (time == 0.0);
ResultShouldBe = (Pump_State == 0);
result = implies(UnderThisCondition, ResultShouldBe);

% GO2: guarantee "The valve is initially closed"
g2 = valve_initially_closed(pre_Valve_State, time);
function result = valve_initially_closed(Valve_State, time)
UnderThisCondition = (time == 0.0);
ResultShouldBe = (Valve_State == 0);
result = implies(UnderThisCondition, ResultShouldBe);

% GO3: guarantee "After the initial time step, When SL_Input is False, the Pump shall be on and Valve shall be Closed"
g3 = sl_input_false_cond(SL_Input, Pump_State, Valve_State, time);
function result = sl_input_false_cond(SL_Input, Pump_State, Valve_State, time)
UnderThisCondition = (SL_Input == 0) && (time > 0);
ResultShouldBe = (Valve_State == 0) && (Pump_State == 1);
result = implies(UnderThisCondition, ResultShouldBe);

% GO4: guarantee "When the SH_Input is true, the Pump shall be off and Valve shall be open"
g4 = sh_input_true_cond(SH_Input, Pump_State, Valve_State);
function result = sh_input_true_cond(SH_Input, Pump_State, Valve_State)
UnderThisCondition = (SH_Input == 1);
ResultShouldBe = (Valve_State == 1) && (Pump_State == 0);
result = implies(UnderThisCondition, ResultShouldBe);

% GO5: guarantee "When the SL_Input is True and the SH_Input is False, the Pump and Valve stay in their previous state"
g5 = sl_input_true_sh_input_false_cond(SL_Input, SH_Input, Pump_State, Valve_State, pre_Pump_State, pre_Valve_State);
function result = sl_input_true_sh_input_false_cond(SL_Input, SH_Input, Pump_State, Valve_State, pre_Pump_State, pre_Valve_State)
UnderThisCondition = (SL_Input == 1) && (SH_Input == 0);
ResultShouldBe = (Pump_State == pre_Pump_State) && (Valve_State == pre_Valve_State);
result = implies(UnderThisCondition, ResultShouldBe);
Requirements Traceability

Requirement - SpeAR Property

\[ g04 = \text{while} \ sensor\_high \]
\[ :: \text{always not pump\_state and valve\_state}; \]

Architecture - AGREE Guarantee

\[ \text{guarantee "G04: After the initial time step,}
\[ \text{When the SH\_Input is true,}
\[ \text{the Pump shall be off and Valve shall be open" :}
\[ \text{true -> ((tank1\_SH\_value = 1.0) =>}
\[ \text{((Valve\_State = 1.0) and Pump\_State = 0.0))}; \]

Modeling - Simulink Design Verifier Property

\[ g4 = \text{sh\_input\_true\_cond(SH\_Input, Pump\_State, Valve\_State);} \]
\[ \text{function result = sh\_input\_true\_cond(SH\_Input, Pump\_State, Valve\_State)} \]
\[ \text{UnderThisCondition} = (\text{SH\_Input} == 1); \]
\[ \text{ResultShouldBe} = (\text{Valve\_State} == 1) \&\& (\text{Pump\_State} == 0); \]
\[ \text{result} = \text{implies(UnderThisCondition, ResultShouldBe);} \]
Model Lifecycle Management Perspective

MLM autonomy perspective starts with MLM framework

- **Analysis Model**
  - «analysis results» rev j
  - «analysis model» rev1
  - «analysis model» rev1a
  - «analysis model» rev2
  - «analysis model» rev1b
  - «analysis model» rev3

- **Architecture Model**
  - «metadata» Change Data
  - «metadata» View-1
  - «arch model» rev1
  - «arch model» rev1a
  - «arch model» rev2
  - «arch model» rev1b
  - «arch model» rev3
  - «arch model» View-1

- **Whole System Model**
  - Includes many other kinds of models and tools

- **CAD Model**
  - «cad model» rev1
  - «cod model» rev2
  - «cod model» rev3
  - Mary

- Produce part based on rev2
- Retrofit/mod based on rev3

Raytheon
Integrated Defense Systems
SysML Representation of Autonomous System and Autonomous System Development

- Building on the MLM framework
- Nominal autonomous system modeled in SysML (Rhapsody example)

- UML Test Protocol or similar utility is used
- Enables efficient pairing of requirements, test straps, procedures, reports, and other artifacts with each member of a product family
- Models are executable within modeling environment at chosen level of fidelity
Basic example of Autonomous Systems T&E in MBE context

- Basic Machine Learning algorithm hosted in Simulink
- Data sets for nominal autonomous system developed
- Simulink components integrated within Rhapsody (SysML)
- Model executed in the SysML environment
- SysML test utilities placed around test and test results
  - IBM Test Conductor or potentially RQM wrapper
- Systems trained with different data sets behaved differently

**MBE considerations**
- Configuration management, Data management
- Flexibility, product family architecture support
- Training Data is paired with the autonomous system
  - Ability to trace system development back to the training data set used

**Autonomous systems development requires additional MBSE considerations**
Summary

- Discovery of critical flaws early in the design process can save time and money
- Formal requirement traceability throughout design process
- Composability for reuse and modular verification
- Autonomous systems development requires additional MBSE considerations
Future Directions of Work

• Continued research in the Development Process
  – Requirements
    • Realizability arguments could identify early conflicts
    • Natural language masking of formal representations
  – Architecture
    • Abstraction of different compositional levels across different teams
  – Modeling
    • Bounding nonlinear behavior within discrete defined systems

• Assurance Case Construction
  – Utilize the artifacts from the Development Process to provide evidence of behavior
    • Move the formulation forward with these artifacts

• Implementing the Development Process on more complex systems
  – Testing the scalability of the techniques
  – Designing challenges that approach the complexity of Air Force domain systems
  – Potentially build on MBSE – autonomy structure

• Run-time Assurance for nonlinear autonomy
  – If we can’t formally prove or test can we bound?
  – How can we safely bound a system?