Formal Methods at NASA: Past, Present, and Future

Paul Miner and Natasha Neogi, NASA Langley Research Center

June 13, 2025

Outline

- Past
- Present
- Future

PAST (PAUL MINER)

- Pre-history of formal methods at NASA Langley (late ‘70s – early ‘80s
- Visit by RSRE in Summer 1987 -- VIPER
- Langley Formal Methods Research 1988 to present

Fault Tolerant Systems (Mid ‘70s to Early ‘80s)

Two competing developments for proof of concept of ultra-reliable computer systems for aircraft control

- Software-Implemented Fault-Tolerance (SIFT)
  - Design – SRI; Implementation – Bendix
  - *SIFT - Design and analysis of a fault-tolerant computer for aircraft control*, Wensley, et al. 1978 https://ntrs.nasa.gov/citations/19790041705
  - Included some formal analysis
  - Original formulation of Byzantine Fault Tolerance https://lamport.azurewebsites.net/pubs/pubs.pdf
- Fault-Tolerant Multiprocessor (FTMP)
  - Design – Draper Labs; Implementation – Collins
  - *FTMP - A highly reliable Fault-Tolerant Multiprocessor for aircraft,* Hopkins, et al. 1978 https://ntrs.nasa.gov/citations/19790041704

VIPER

In the mid 1980s, the Royal Signals and Radar Establishment (RSRE) at Malvern developed the Verifiable Integrated Processor for Enhanced Reliability (VIPER)

- John Cullyer of the RSRE visited NASA Langley in the summer of 1987.
- This visit resulted in the establishment of the formal methods research team at NASA Langley.
- Initial activity was to task Computational Logic, Inc. to provide an independent assessment of the VIPER verification claims
  - *Report on the formal specification and partial verification of the VIPER microprocessor*, Bishop Brock and Warren Hunt, 1991 https://ntrs.nasa.gov/citations/19910018472

Early Formal Methods at LaRC

- In-house team started in 1988
  - Ricky Butler (team lead)
  - Mix of civil servants and on-site contractors
  - Initial focus on fault tolerant systems and digital hardware design
- MOU with RSRE Malvern
- External contracts beginning in 1989
  - SRI International
  - Computational Logic, Inc.
  - Odyssey Research Associates
- Early activities focused on pairing formal methods specialists with industry partners
  - Desire to ground research in relevant applications

- Langley's Formal Methods Program hosted one of the earliest websites describing formal methods
  - Residual pieces are here:
    - https://shemesh.larc.nasa.gov/fm/oldfm/
  - More recent, but still old:
    - https://shemesh.larc.nasa.gov/fm/fm-now-contract.html
  - Current site:
    - https://shemesh.larc.nasa.gov/fm/index.html

Next slide: Participants of the 1st NASA Langley Formal Methods Workshop from August 1990

How many can you identify?


NASA

Thanks to the College of William and Mary:

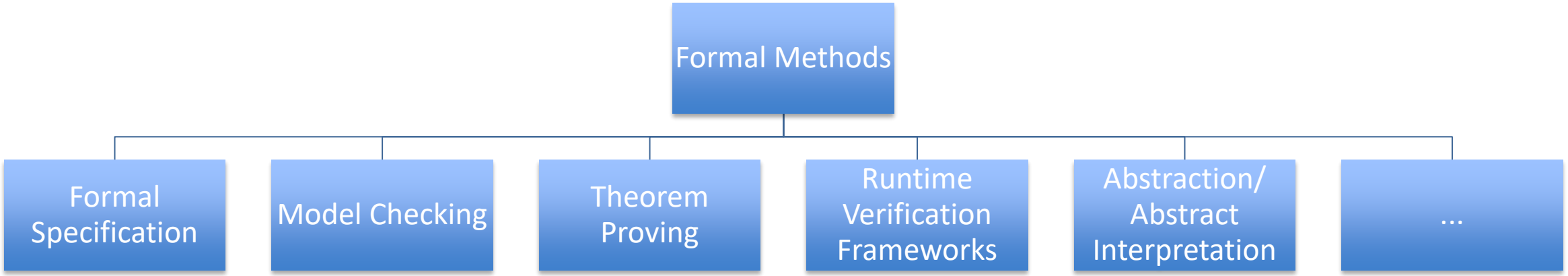
- Verification of Fault-Tolerant Clock Synchronization Systems, Paul S. Miner, 1992. Master's Thesis, College of William and Mary
  - https://scholarworks.wm.edu/etd/1539625738/
  - 1993 NASA Technical Paper version
    - https://ntrs.nasa.gov/citations/19940012976

I still use derivatives of this work decades later

PRESENT (PAUL MINER & NATASHA NEOGI)

The Different Types of Formal Methods at NASA Today

- Although the use of mathematical logic is a unifying theme across the discipline of formal methods, there is no single best "formal method".
- NASA's use of formal methods spans multiple centers (e.g., LaRC, JSC, ARC, JPL, etc.) and employs multiple different techniques (and tools)

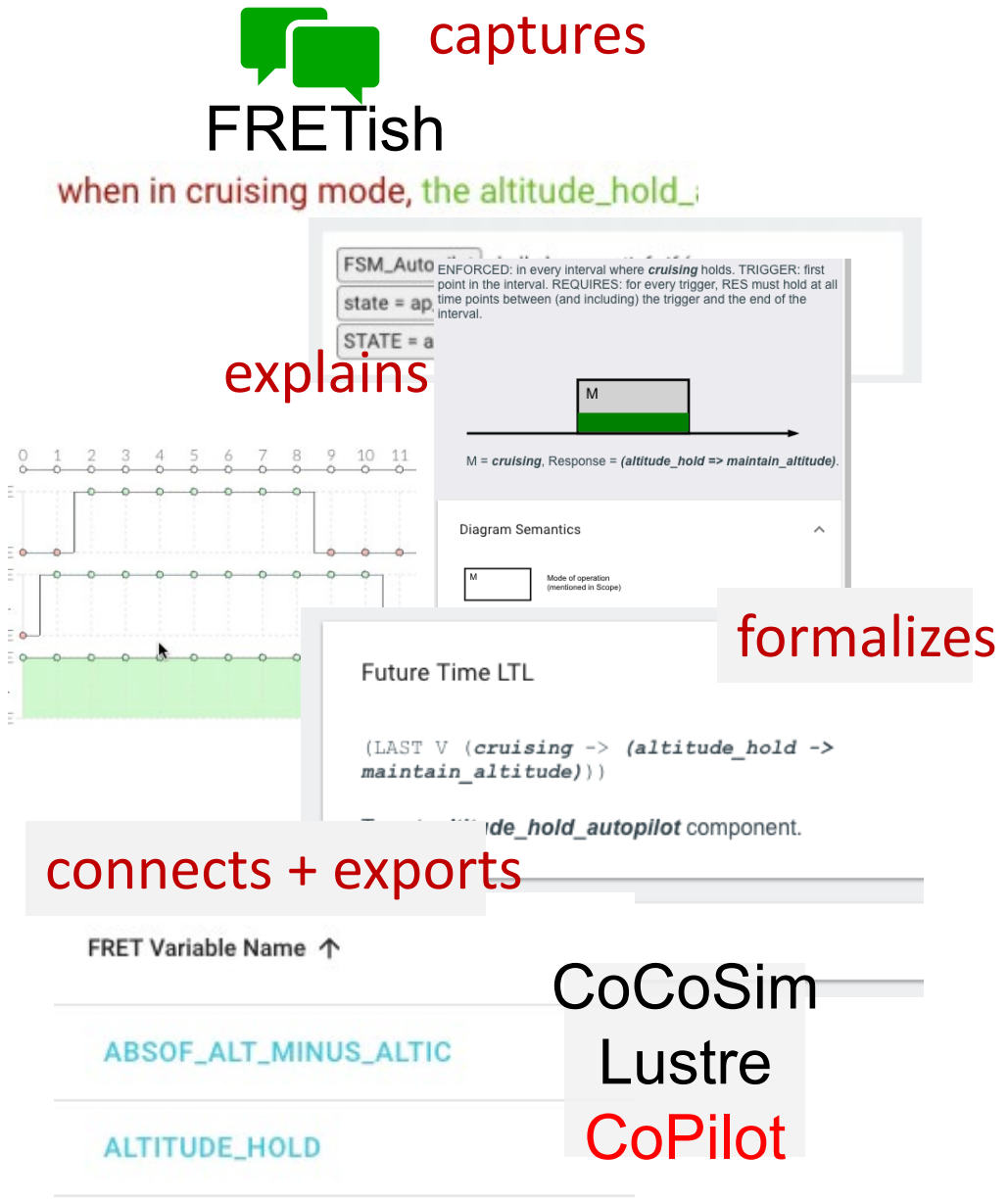


- Application domains (i.e., space, aeronautics, etc.) can require different modeling methods and different proof approaches.
- Different phases of the life-cycle may be best served by different tools and techniques.

Credit: https://shemesh.larc.nasa.gov/fm/

Formal Specification: Formal Requirements Elicitation Tool (FRET)

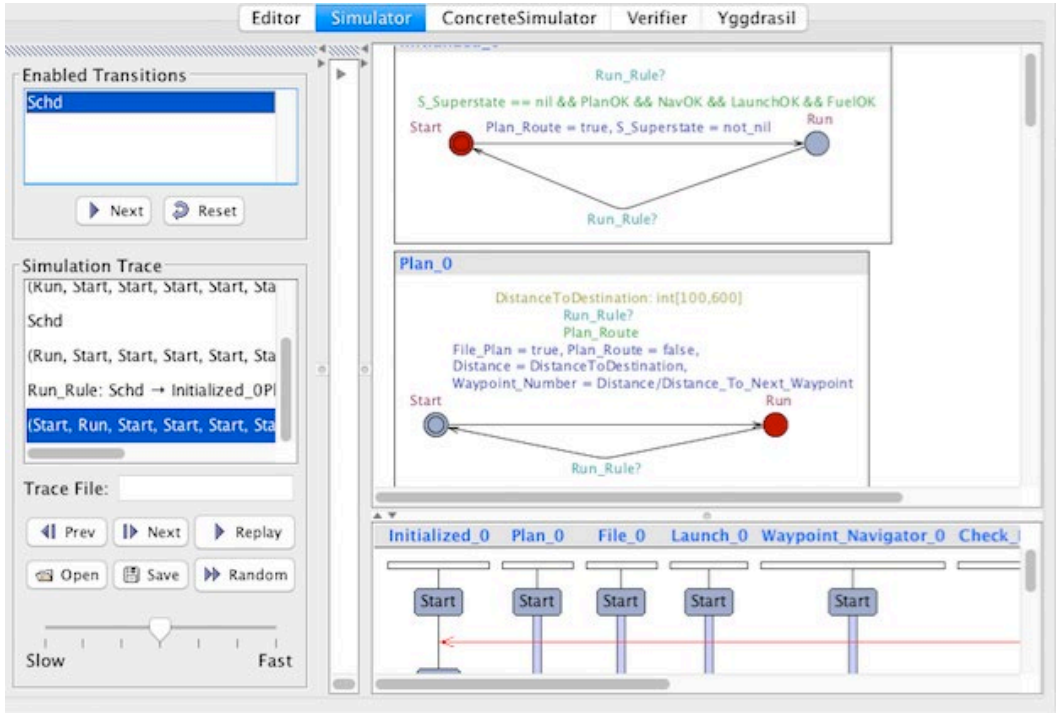
- Framework for the elicitation, formalization and understanding of requirements
- Allows its user to enter hierarchical system requirements in a structured natural language.
  - Requirements written in this language are assigned unambiguous semantics.
- Supports user in understanding semantics and reformulating requirements if applicable
  - natural language description,
  - formal mathematical logics,
  - diagrams, and
  - interactive simulation.
- Export requirements into forms that can be used by a variety of analysis tools, such as Cocosim, Simulink Design Verifier, and CoPilot.

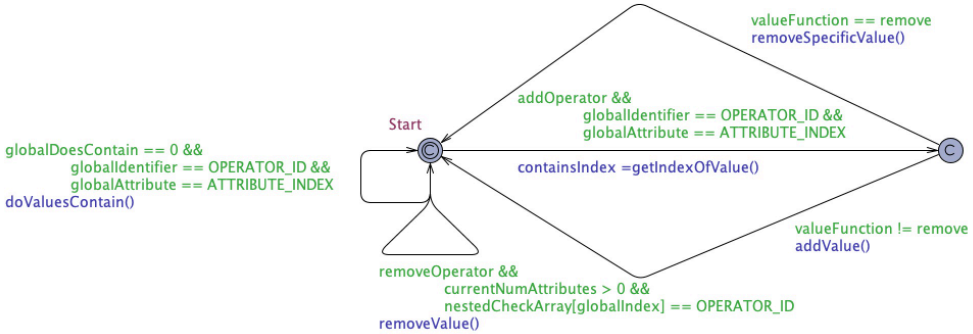


Technical POC: Anastasia Mavridou, (anastasia.mavridou@nasa.gov)

Model Checking: UPPAAL & Lost Link Case Study

- Modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)
- Supports modelling of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables
- Uppaal consists of three main parts:
  - a description language—system behavior described as networks of timed automata extended with clock and data variables.
  - a simulator—validation tool which enables examination of *possible* dynamic executions of a system during early design (or modeling) stages.
  - a model-checker—tool that checks invariant and reachability properties by exploring the state-space of a system





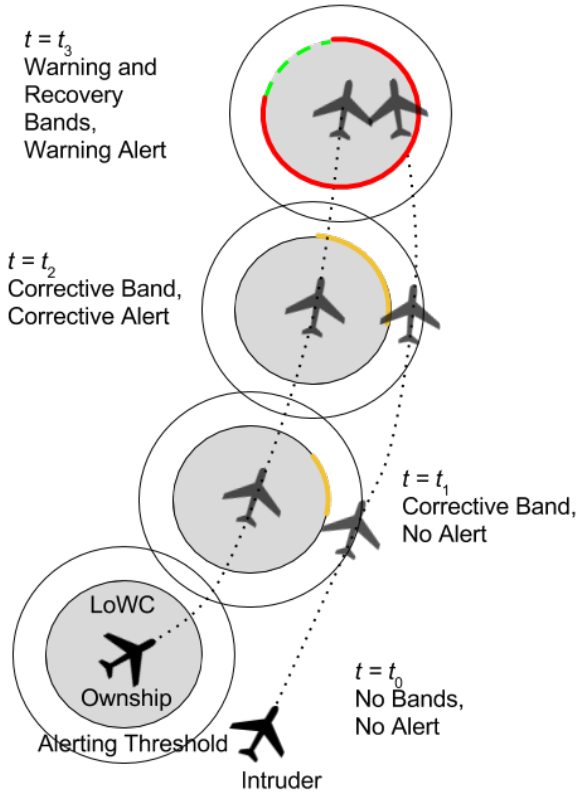
Technical POC: Natasha Neogi, natasha.a. neogi@nasa.gov

Theorem Proving: Prototype Verification System (PVS) & CD&R

- PVS Environment allows for the formal specification and verification of complex systems
- PVS consists of a specification language, an interactive theorem prover and a number of other features…
- The specification language of PVS is based on classical, typed higher-order logic.
- The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework.
- Serves as a productive environment for constructing and maintaining large formalizations and proofs

DAIDALUS

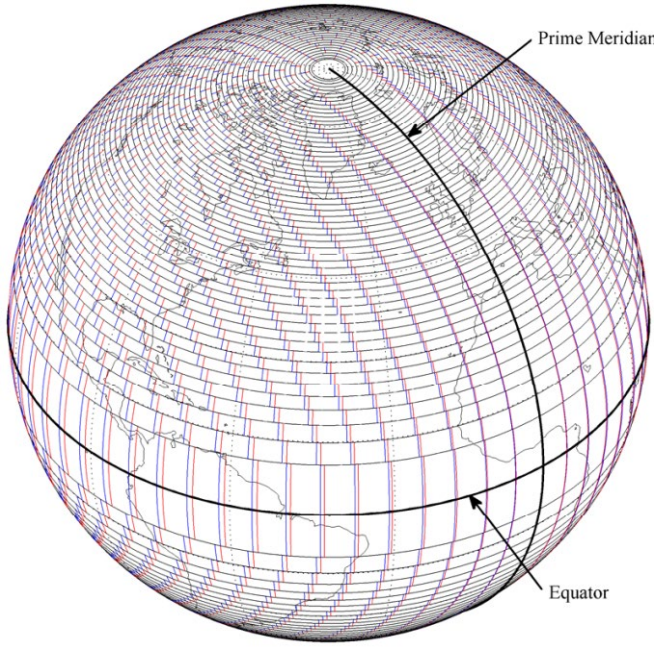
ICAROUS
Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems



Daidalus in use in the MACS simulation environment

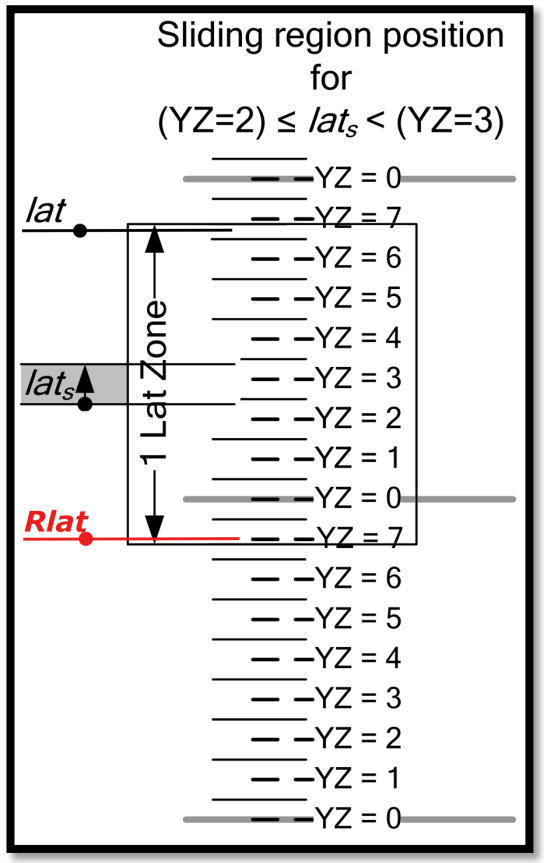
Technical POC: C. Munoz, cesar.a.munoz@nasa.gov https://shemesh.larc.nasa.gov/fm/ICAROUS/

Compact Position Reporting—ADS-B Positioning



CPR divides the globe into "zones," and transmits only the target's position within the zone. The receiver has to determine the correct zone for proper decoding.

- CPR is used to save message space in ADS-B position messages
- Formal analysis led to tightening of decoding requirements, and simplified calculations.
- Spurred development of a PRECiSA, a tool for formal analysis of floating point (IEEE-754 spec) programs
- Formally verified implementations in floating point (double) and fixed point (single).
- Changes from formal analysis and verified implementation to be in revision C of DO-260 (ABS-B MOPS)



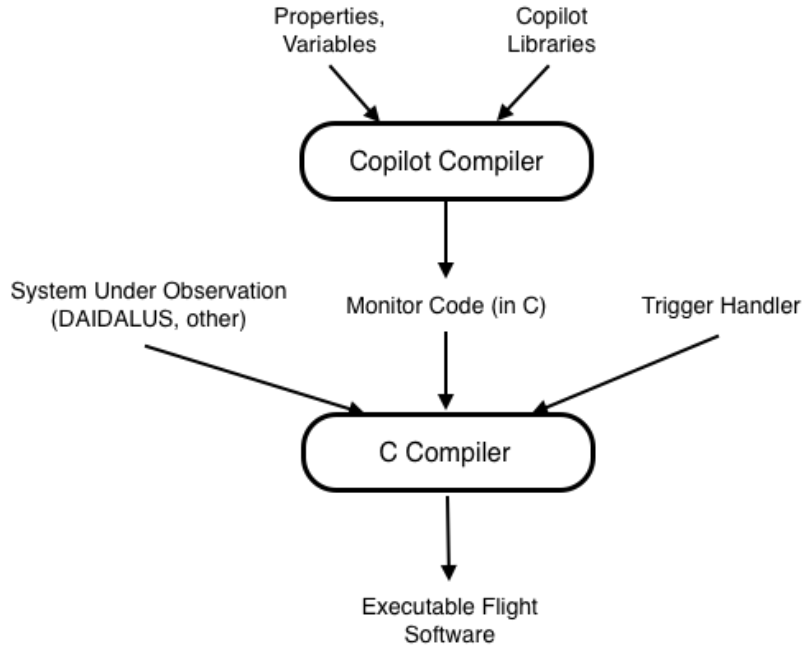
Visualization of loose requirement for decoding. Target is within stated distance threshold, but decodes incorrectly

POC: Aaron Dutle (aaron.m.dutle@nasa.gov)

https://shemesh.larc.nasa.gov/fm/CPR/; https://shemesh.larc.nasa.gov/fm/PRECiSA/

Runtime Verification: CoPilot Runtime Verification Framework

- High-assurance RV can be assured like any conventional safety-critical system though the system under observation cannot
  - Far simpler than the System Under Observation (SOU)
- Challenges
  - If RV is to ensure safety, the specifications being enforced must be derived from safety analysis
  - RV frameworks should generate documentation to support traceability from specification to code
  - Assure the correctness of the monitor specifications (e.g., specification written in Regular Languages, LTL, MTL etc.)
  - Assure that the monitors correctly implement the specification
  - Assured RV must safely compose with the SUO
- **Copilot RV Framework**
  - Haskell based Embedded Domain Specific Language (DSL)
  - Synthesize monitors for real-time embedded systems
  - Generates Misra-like C monitors and Verilog
    - Constant time, constant memory
  - Minimum instrumentation of SUO source code
  - Samples the system under observation
  - Integrated with tool support via Ogma
    - FRET, F', CSF, MBSE tools etc.



Technical POC: Alwyn Goodloe, a.goodloe@nasa.gov

Runtime Verification: PyContract and Offline Monitoring

- Objective is to analyze a log file produced by a previous run of the SUO
  - Less constraints concerning memory use
  - Enables use of more expressive monitoring languages
- Processing log files often requires performing computation on data in log (e.g., substring extraction, averaging, etc.)
  - Internal DSL (i.e., library in a general-purpose programming language) is often needed
- PyContract is a Python library offering a way of writing temporal properties using a combination of state machines and rule-based programming
  - States can be parameterized with data, effectively representing facts, which are stored in the monitor memory, as is typical for rule-based systems
- Monitor is instantiated as a class, with any statistics desired being a part of the monitor (statistics class)
  - Monitor can be fed a trace to verify for desired properties or can be fed events sequentially (potentially enabling online monitoring)
  - Enables various forms of visualization and trace mining
  - Potential to exploit links between runtime verification and data analysis

All Credit and Technical POC: Klaus Havelund (klaus.havelund-1@nasa.gov)

Other Techniques for V&V

- Abstract Interpretation—PRECISA, etc.
- SAT/SMT Solving—Integrated into multiple frameworks/tools
- Static analysis of code—AdaStress, ReFlow, etc.
  - Does not require execution of code
  - Lexical analysis of the program syntax
  - Investigates and checks the structure and usage of individual statements; often automated
- Dynamic analysis of code (RACE, etc.)
  - Involves running the system (testing)
  - Program run formally under controlled conditions with specific results expected
  - Path and Branch Testing
- Note that Simulation & Testing explore some of the possible behaviors and scenarios of a system (CocoSim & Lustre, etc.):
  - They leave open the problem of possible edge cases in the unexplored scenarios.

FUTURE (NATASHA NEOGI)

Safety Oriented Challenges for Cyber Physical Systems

- Verification and Validation of Increasingly Automated (IA) Systems
  - Properties of Concern: Safety, Liveness, Security, Fairness…
- Human Machine Teaming Interactions
  - Role Allocation: Authority and Responsibility
- Bounding Behavior of IA Functions in Uncertain Environments
  - Contingency Management
  - Fault Containment
  - Heterogeneous Vehicles
  - Mixed ConOps
- Trusted Decision Making
  - Adaptive/Non-Deterministic
  - Shifting control paradigm
- Certification & Operational Approval
- Public Acceptance/Trust

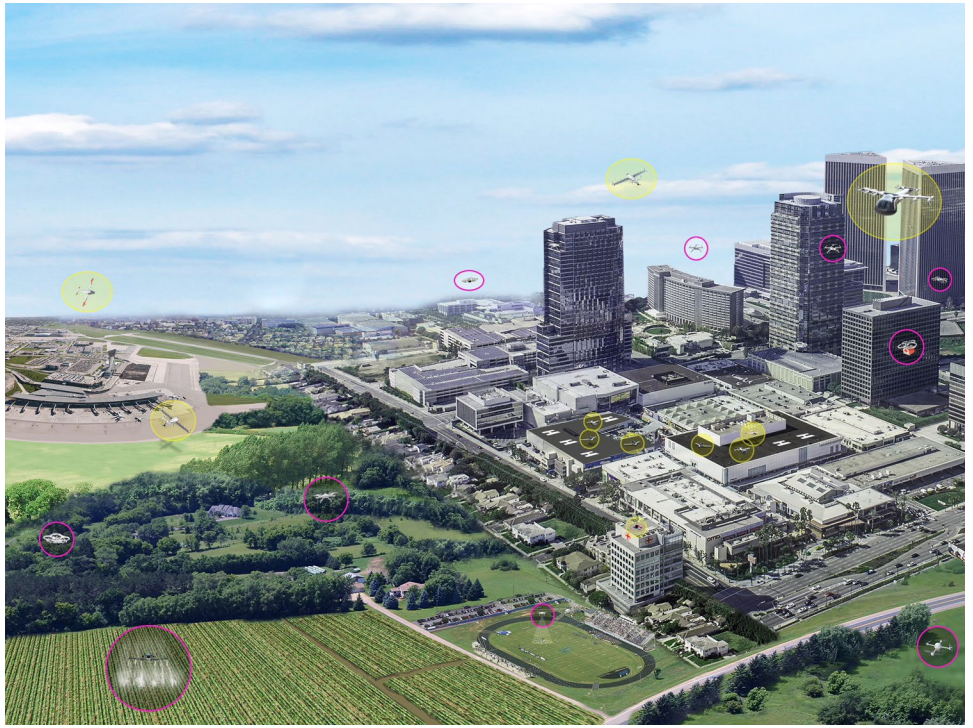


Photo Credits: NASA Ames Research Center https://www.nasa.gov/directorates/armd/aosp/amp/

FM for CPS: Distributed Hierarchical Framework for Spacecraft Autonomy

- Distributed and hierarchical approach to system monitoring and control is a key idea in the Modular Autonomous Systems Technology (MAST) framework
  - Allows for variable autonomy
- Framework enables a component-based architecture that provides interfaces and structure to developing autonomous technologies
- Supports communication and transparent interfaces between components
- Enforces a strict command and telemetry flow as a systems engineering tool
- Framework has a path to formal analysis and will create assume-guarantee contracts as long as the autonomous technology components can be verified individually
  - Inclusion of contract-based design concepts encourages design for verification methodologies
  - Supports component-level verification
  - Plays important role in overall system verification
- Facilitates data logging via its integration with the Lightweight Accumulator Gathering Efficiently in Real-time (LAGER) logging software

All Credit and Technical POC: Julia Badger (julia.m.badger@nasa.gov)

AI/ML Verification Challenges

	Traditional (Physical) Components	ML (Software) Components
Criteria	Criteria are simple (e.g., failure/breakage rate etc.) for the component as a whole.	Complexity of ML and its interdependence on its domain and environment make it difficult to have explicit and precise articulation of meaningful criteria that can be measured.
Feasibility of Testing	For physical artifacts, limited testing provides compelling evidence of quality, with the continuity of physical phenomena allowing widespread inferences to be drawn from only a few sample points.	Limited testing of ML cannot provide compelling evidence of behavior under all conditions.
Process & Product Correlation	Underlying principle of statistical quality control is that sampling the product coming out of a process gives a measure of the quality of the process itself, which in turn will determine the quality of items that are not sampled.	More rigorous ML design processes will likely lead to better quality ML components. However, this correlation is not sufficient as the sole provider of evidence, as correlation does not imply causation.

Adapted from: National Research Council. 2007. *Software for Dependable Systems: Sufficient Evidence?*. Washington, DC: The National Academies Press. https://doi.org/10.17226/11923.

Using Formal Methods for AI/ML Verification

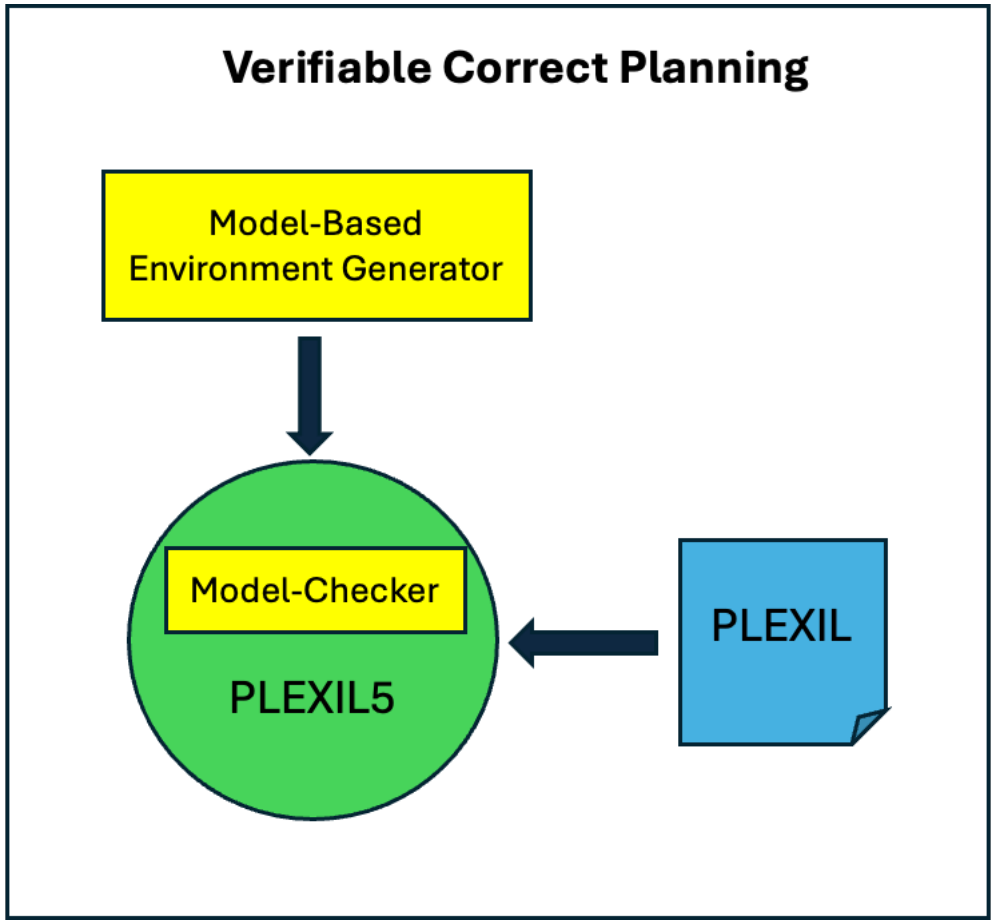
- Testing for ML components is indispensable.
  - However, testing alone is insufficient, as it is unclear what coverage means in terms of ML components.
- Simulation and analysis can provide needed checks for ML components.

Validation of environmental assumptions, interface assumptions, and constraints	Feasibility or satisfiability analysis of temporal behaviors	Verification of code implementation against component specifications	Checking that components in aggregate achieve appropriate system-level effects

  - However, simulation and analysis is insufficient due to model inaccuracy, incorrect assumptions (e.g., environmental, operator response, execution platform), etc.
- Formal methods can provide guarantees for ML components.
  - Formal methods can provide formal proofs of correctness.
  - Formal methods techniques often lack scalability.

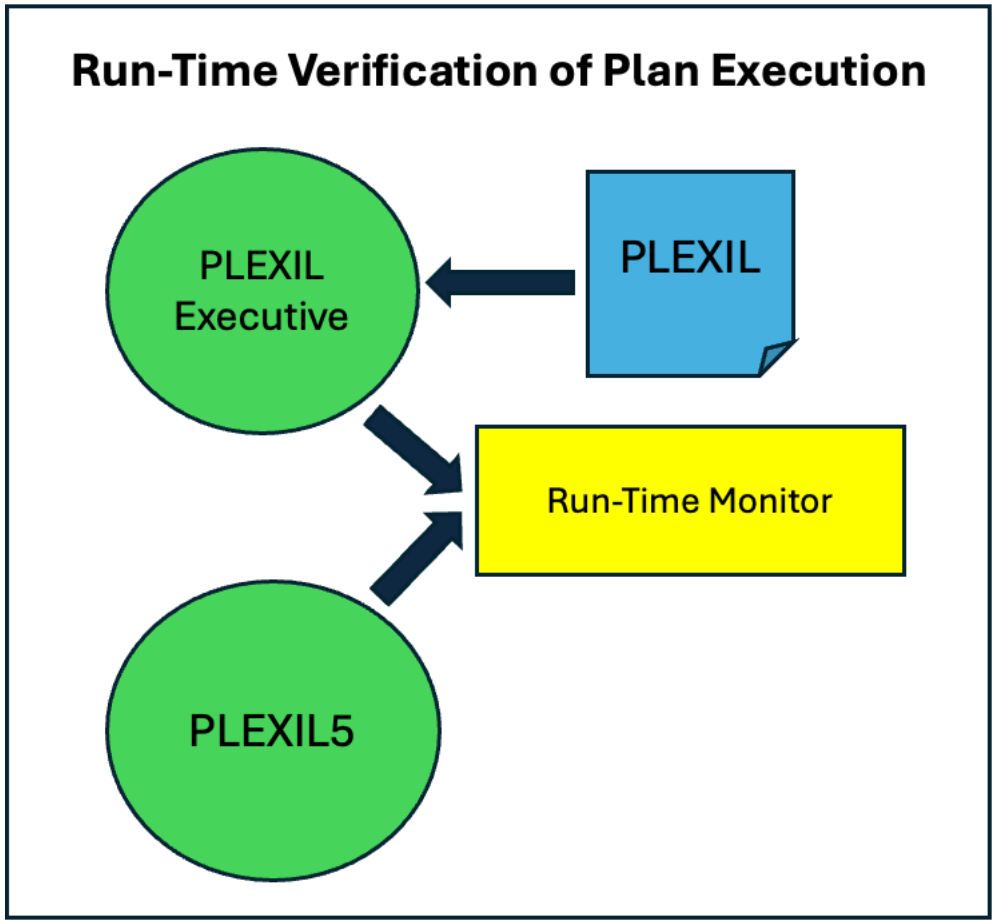
FM for AI: Plan Execution Interchange Language (PLEXIL) (I)

- **PLEXIL** is a *NASA-developed* plan execution language for representing plans for **automation**, as well a technology for executing these plans on real or simulated systems.
- PLEXIL5 is a NASA-LaRC developed formal operational semantics of PLEXIL
  - Provides a reference implementation of the PLEXIL executive
  - Uses theorem-proving and model-checking for the formal verification of plans and plan executions.



FM for AI: Plan Execution Interchange Language (PLEXIL) (II)

- **PLEXIL** is a *NASA-developed* plan execution language for representing plans for **automation**, as well a technology for executing these plans on real or simulated systems.
- PLEXIL5 is a NASA-LaRC developed formal operational semantics of PLEXIL
  - Provides a reference implementation of the PLEXIL executive
  - Uses theorem-proving and model-checking for the formal verification of plans and plan executions.



Using AI/ML for Formal Verification (I)

- Aiding specifier in writing a specification
  - Suggesting or accessing templates for common specification patterns and data structures in the formal tool (caveat: must be checked by specifier)
  - Suggesting a code implementation based on specification (caveat: must be checked by specifier)
- Searching for information, models, techniques, and methods related to a given framework or tool
  - Searching for a lemma in a proof library
  - Suggesting a next step in a proof
- Proof repair and change management
  - Checking whether alterations to a proof (e.g., adding a new case, changing assumptions, etc.) break it
  - Suggesting structures by which proof can be repaired (e.g., using same inductive method, providing schema for cases, etc.) thereby leveraging how proof was previously done

(Selected) Remaining Challenges for FM

- Scalability and Complexity
  - Language, technique, and tool scalability
  - Verification at scale (i.e., parallelization)
  - Modelling and abstraction
  - Time and cost constraints
- Education and Training
  - Of the FM users
  - Of the FM developers
- Ease of Use and Interpretability
  - Specialized expertise
  - Ambiguous results

Summary Thoughts

- FM can detect defects earlier in life cycle
- FM can be applied at various levels of resource investment
- FM can be integrated within existing project process models
- FM can improve quality assurance when applied judiciously to appropriate projects
- FM can enable the assurance of systems not currently amenable to traditional practices

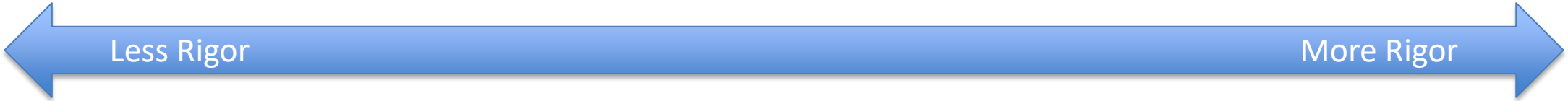
However:

- FM are not a panacea, but can increase confidence in a product's reliability if applied with care and skill
- FM are very useful for consistency checks, but FM cannot assure completeness of a specification
- Judicious application of FM to suitable project environments is critical if benefits are to exceed costs
  - Scalability
  - Reusability
- FM and problem domain expertise must be fully integrated to achieve positive results

BACKUPS

What is Formal Methods?

- "Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.
- The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e., each step follows from a rule of inference and hence can be checked by a mechanical process.)



Occasional mathematical notation embedded in English specifications

Fully formal specification languages with a precise semantics

- Formal methods can:
  - Be a foundation for describing complex systems
  - Be a foundation for reasoning about systems
  - Provide support for program development

Why use Formal Methods?

- The value of formal methods is that they provide a means to symbolically examine the entire state space of a system design and establish a correctness or safety property that is true for all possible inputs.
- Several approaches are used to address the state explosion problem associated with real systems:
  - Apply formal methods to requirements and high-level designs where most of the details are abstracted away
  - Apply formal methods to only the most critical components
  - Analyze models of software and hardware where variables are discretized and ranges drastically reduced.
  - Analyze system models in a hierarchical manner that enables "divide and conquer"
  - Automate as much of the verification as possible

Formal Specifications

- Formal Specifications:
  - Have their syntax and semantics rigorously defined
  - Have a precise form, perhaps mathematical
  - Eliminate imprecision and ambiguity
  - Provide a basis for mathematically verifying equivalence between specification and implementation
  - May be hard to read without training
- A Formal Specification has two parts:
  - Logical Theory: Means by which one reasons about specifications, properties and programs
    - First order predicate calculus (quantification over variables); Second order predicate calculus (quantification over relations); Temporal logic etc.
  - Structuring Theory: Defines elements being reasoned about
- Property Oriented Formal Specification: State desired properties in a purely declarative way
  - Algebraic: Data type viewed as an algebra, axioms state properties of data type's operations
  - Axiomatic: Uses first order predicate logic, pre and post conditions
  - Operational Specification: Describe desired behaviour by providing model of system
- Model Oriented Formal Specification: Provide direct way of describing system behaviour (sets, sequences, tuples, maps) :
  - Abstract Model (in terms previously defined mathematical objects eg., sets, sequences, functions, etc.)
  - State machines

Model Checking

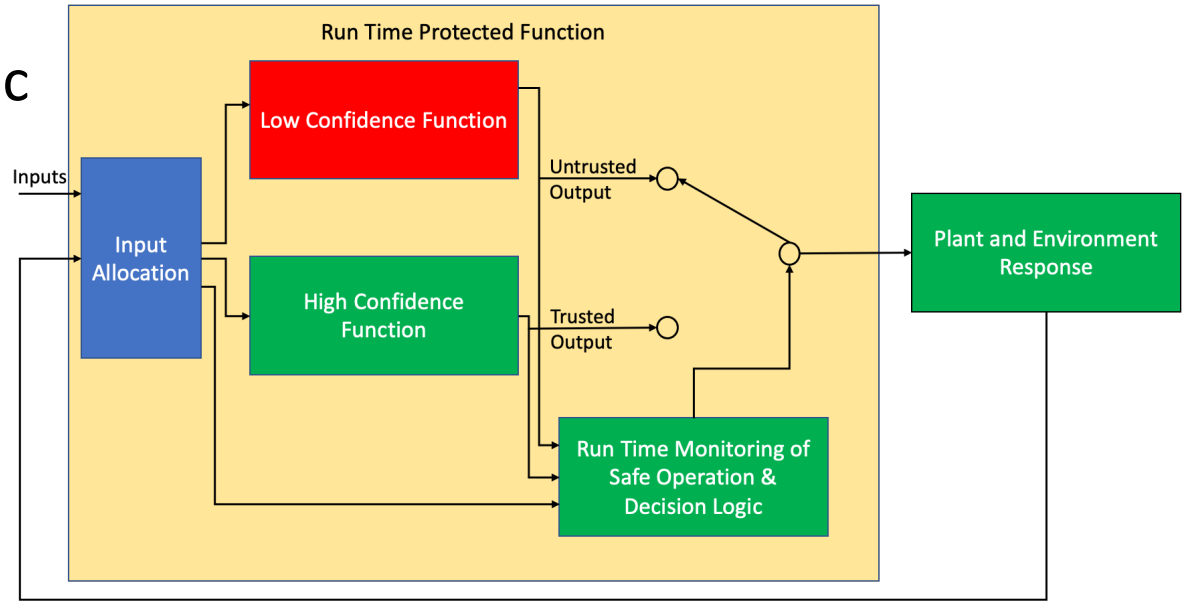
- Model checking describes a system and its states by means of logical notions called a Formal Specification. It then applies an automatic logic checker to verify correctness.
  - Formal Specification: Model $\mathcal{M}$ of a particular logical theory.
  - Property to Verify: Formula ϕ of the chosen logical language.
  - Verification Method: Check whether the model $\mathcal{M}$ satisfies the property ϕ: $\mathcal{M} \vDash \phi$.
- Note that in a time varying setting, the truth of a formula becomes a dynamic notion: the formula can be true in one state of the system and false in another state.
  - Formal Specification: The Model $\mathcal{M}$ is a *Transition System* (e.g., state machine, hybrid automaton etc.).
  - Property to Verify: Formula ϕ is expressed in a *Temporal Logic* (e.g., Linear Temporal logic etc.).
  - Verification Method: Check whether the transition system $\mathcal{M}$ satisfies the TL property ϕ: $\mathcal{M} \vDash \phi$.
- Model Checking has the following attributes:
  - It is fully automatic, and it does not require user supervision;
  - When the design fails to satisfy a property, the method produces a counterexample that shows a behavior which falsifies that property;
  - The advent of Symbolic Model Checking allows one to describe implicitly an astronomic number of states.

Theorem Proving

- Formal verification involves the use of logical and computational methods to establish claims over a system via a proof, when both are expressed in precise mathematical terms
  - Most conventional proof methods can be reduced to a small set of axioms and rules in any number of foundational systems.
- Theorem provers can help establish a claim by:
  - finding the proof, and/or
  - verifying that a purported proof is correct.
- Theorem provers can be automated or interactive:
  - *Automated theorem proving* focuses on finding the proof
    - Resolution theorem provers, tableau theorem provers, fast satisfiability solvers, etc., provide a means of establishing the validity of formulas in propositional and first-order logic
    - Hard to guaranteed soundness and it is difficult to ensure that the results they deliver are correct.
  - *Interactive theorem proving* focuses on verifying the proof is correct.
  - Every claim is supported by a proof in a suitable axiomatic foundation → every rule of inference is justified by appealing to prior definitions and theorems, all the way down to basic axioms and rules.
  - Requires deep understanding and interaction from users

Runtime Verification

- Runtime verification (RV) refers to the use of monitors to observe the behavior of a system and detect if it is inconsistent with a given specification
  - Monitored system - system under observation (SUO)
- RV specifications are written in a decidable logic or appropriate abstract notation
  - Regular languages
  - Linear temporal logic (LTL)
  - Metric temporal logic (MTL)
- Typically, when a monitor detects that a property is violated, it raises an alarm or steers the system to a safe mode of operation
- RV frameworks automatically synthesize monitors from specifications



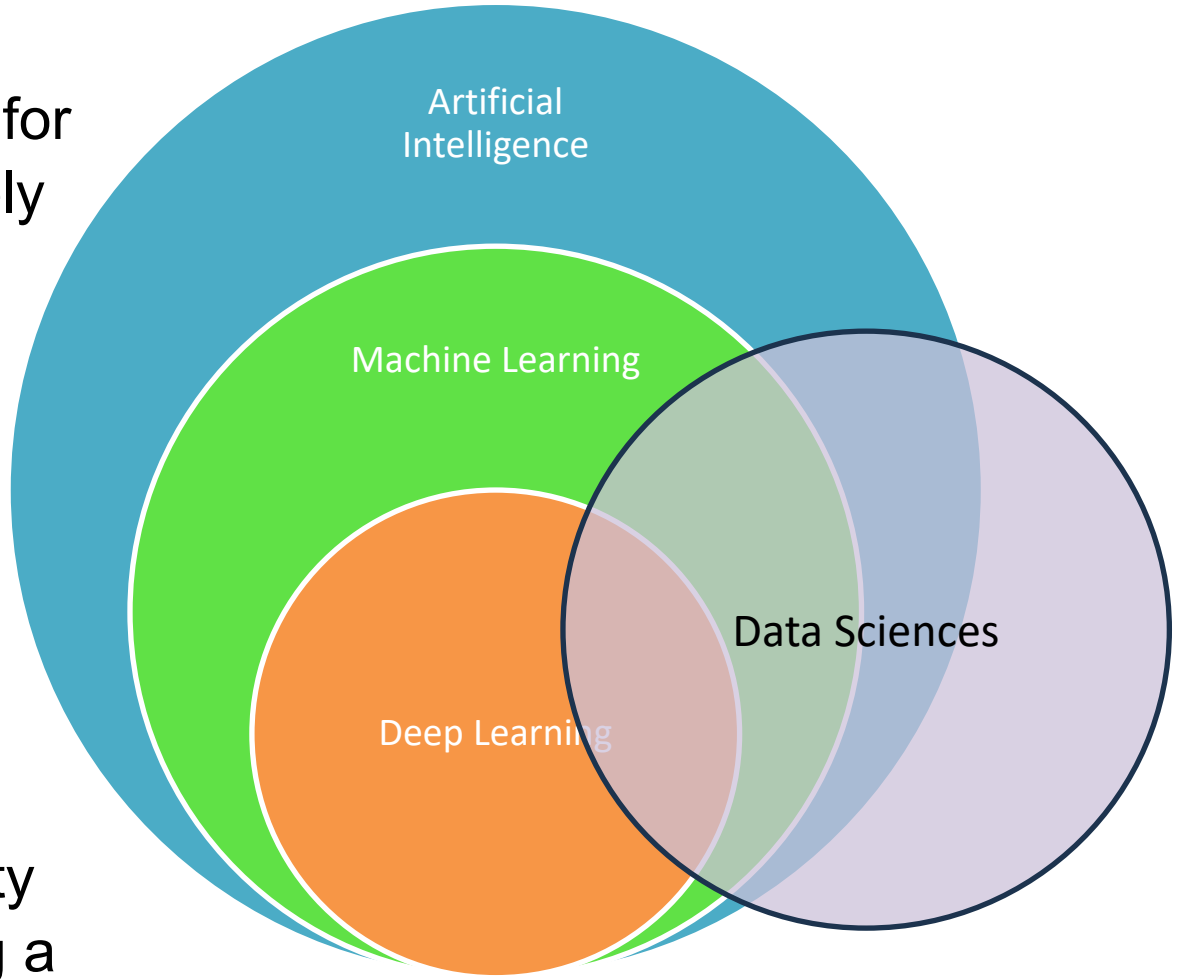
Problem and Goal

Problem

- The inability to establish appropriate assurance for AI/ML components leaves us unable to effectively manage their risks and benefits.
  - Drives cost of development uneconomically high
  - Delays adoption of AI/ML at scale in safety critical systems
  - Results in unknown and unmanageable risks

Goal

- Discover and define what constitutes sufficient scientific-based evidence to substantiate a safety claim related to an AI/ML component performing a safety-critical function.



AI/ML will not see widespread adoption in safety-critical aviation systems until it is properly assured.