Formal Methods for Flight-Critical Systems

Alwyn E. Goodloe

NASA Langley Research Center

May 9, 2011
I work in the formal methods team of the Safety-Critical Avionics Systems Branch at NASA Langley Research Center.

At Langley historical focus on formal methods applied to fault-tolerant avionics and airspace management.

NASA Ames and JPL have formal methods groups that focus on model checking, static analysis, symbolic execution, etc.

Approximately 25 researchers in NASA working in formal methods.

Computers increasingly control and monitor all critical aircraft functions.

Fuel management, Navigation, Flight control.

Critical flight systems proscribed to have one in a billion ($10^{-9}$) probability of catastrophic failure per hour of operation.

Flight critical software subject to very very stringent FAA regulation in the form of DO-178B.

Yet incidents still occur.
Example: Space Shuttle

- 2008, a pre-launch failure of STS-124 was reported in the Space Shuttle data processing system
  - Four general purpose computers (GPC)
  - FA 2: the flight-aft mux/demux card
- Diode fails on FA2
- GPC 4 receives bad data from FA 2 in the data comparisons with GPC 1-3, it is voted out
- Then similarly for GPC 2
- GPC 3 also determined to be faulty
- Described as a “non-universal I/O error”
- See “Murphy was an Optimist”, K. Driscoll. SafeCOMP 2010
Example: Quantas Flight QF72 (A330)

- Autopilot disengaged and the *IR 1 failure* indication appeared on the ADIRS control panel.
- The flight display indicated that the aircraft was simultaneously approaching over-speed limit and the stall speed limit.
- The *ADR 1 failure* indication appeared on the ADIRS control panel.
- Two minutes into the incident, a high angle of attack was reported and the computers ordered a significant nose-down pitch and the plane descended 650ft.
- The crew switched the PRIM master from PRIM 1 to PRIM 2.
- PRIM 3 indicates a fault.
- The crew returned the aircraft to flight level 370.
- The captain switched his display to show data from ADIRU 3 instead of ADIRU 1.
Example: Quantas Flight QF72 (A330)

- The computers ordered a nose down pitch and the plane descends 400ft
- The captain applied back pressure to the sidestick
- The crew switched the PRIM master from PRIM 2 to PRIM 1
- The flight control law was manually changed from ‘normal law’ to ‘alternate law’ so that the computer was no longer enforcing predefined flight parameters
- The crew made an emergency landing at Learmouth
- People were hospitalized
- The safety checks were acting on data streams that were exhibiting Byzantine behavior
Systems Must be Designed to be Robust

- Formal methods help detect the *Physically Possible, but Logically unanticipated*
- Model and analyze the system design
- Formal methods help ensure the code meets the design
- Formal methods help verify safety properties at runtime
- My research has touched on all three levels (models, code, and runtime)
- Today’s survey touches on all three
Formal Models

- Analysis on models that abstract away some details, but model both physical and cyber components of a cyber-physical system
  - The most costly errors occur here
  - Assumptions such as Byzantine faults not possible
- Our applications require higher math and the physics of controls, fault tolerant systems, air space management, which drives us to interactive theorem provers (PVS, ACL2, Isabelle, Coq, ...)
  - Can take months/years of work to build application specific libraries in order to be truly effective
Verifying Controls

- Motivation comes from work of Prof Eric Feron on verifying control software using Hoare logics
- Collaborators: Eric Feron, Tim Wang, and Romain Jobredeaux (all Ga Tech), and Heber Herencia-Zapana (NIA)
- Currently control software *works like it always has*
  - V&V grandfathered
- Researchers in controls have made amazing discoveries that do not make it into commercial aircraft
- Computing scientists are part of the problem: we do not know the math or dynamical systems
- Control researchers are part of the problem: they know almost no computer science
Control system engineering as Seen by Control Researchers

System Data → System Identification → Design the Controller → Perform Control System Analysis

Invalid Controller → Valid Controller

No Good to Use!

Verification & Validation → Compiler → Control Code Generation

1. Manual Coding
2. Use Autocoder i.e. Matlab, Simulink/Realtime Workshop

Good to Use!
The Reality for Industrial Control Engineers

---

Control Code Generation
1. Manual Coding
2. Use Auto code e.g., Matlab, Simulink, Realtime Workshop

---

BOEING'S PROBLEM!
This project is aimed at applying formal techniques to verify control software.

Control researchers build models in Matlab and simulate.

Want to show they are robust (stable) via formal proof.

Hand translate Matlab algorithms to PVS.

Autocode verified C from Matlab.

Our focus is how to demonstrate to regulators EVIDENCE that control algorithms and their implementation are safe.
Basic Linear System

\[ \dot{x}(t) = A(t)x(t) + B(t)u(t) \]
\[ y(t) = C(t)x(t) + D(t)u(t) \]

State Vector \( x \)  A vector whose elements are the state variables
State Space  The \( n \)-dimensional vector space whose axes are the state variables
State Equations \( A \)  A set of \( n \)-simultaneous first-order linear differential equations
Input Control Matrix \( B \)
Output Vector \( y \)
Output Equations \( C \)  The algebraic equations that expresses the output variables as linear combinations of the state variables and the inputs
Input Control Vector \( u \)
Feed Forward Matrix \( D \)
Damped Spring

Figure: Dampened Spring
Damped Spring

\[ m\ddot{x} + c(x) + kx = 0 \]

Defining the state variables \( x_1 = x \) and \( x_2 = \dot{x} \) and \( \mathbf{x} = [x_1 \ x_2]^\top \), we get

\[
\begin{bmatrix}
\dot{x}_1 \\
\dot{x}_2 
\end{bmatrix}
= 
\begin{bmatrix}
x_2 \\
-\frac{c}{m}x_2 - \frac{k}{m}x_1 
\end{bmatrix}
= 
\begin{bmatrix}
0 & 1 \\
\frac{-k}{m} & \frac{-c}{m} 
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 
\end{bmatrix}
= \mathbf{A}\mathbf{x}
\]
Need Linear Algebra

- Linear algebra is fundamental to controls
- Proofs of robustness require us to have enough linear algebra to express theorems on linear matrix inequalities (LMI)
- Building a PVS linear algebra library focusing the math needed to prove control systems stable
  - Recently SRI joined the effort
A Taste of Linear Algebra

- Our first encounter with linear algebra is usually computational
  - Matrix operations
  - How to solve linear equations ($Ax = c$)
  - Computing Eigenvalues

- Control software is of this nature
- We do not want to turn a theorem prover into Matlab
- The following is a taste of the sort of linear algebra we have built in PVS to prove robustness properties
An $n$–dimensional vector space is a set closed under vector addition and scalar multiplication.

- We only consider $\mathbb{R}^n$.

A basis of a vector space $V$ is a subset of $V$ from which the vector space may be generated.

The standard basis $e$ of an $n$ dimensional vector space is defined as the set $e = \{e_1, e_2, \ldots, e_n\}$, where $e_1 = (1, 0, 0, \ldots, 0), \ldots, e_n = (0, 0, \ldots, 0, 1)$.

Every vector in $V$ can be represented as a finite linear combination of the vectors in $e$.

If a property holds for every element of the basis of a vector space, then the property holds for every vector of the vector space.
Quick Review of Matrix Algebra

- Transpose $A^\top(i, j) = A(j, i)$
- $(Ax)_{i,1} = \sum_{k=1}^{n} A_{i,k}x_{k,1}$
- $(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$
- $AA^{-1} = I = A^{-1}A$
- $(A + B)x = Ax + Bx$
- $(cA)x = c(Ax)$
A Linear map, \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \), is a linear function on vectors that satisfies the standard linearity property:

\[
\forall x, y \in \mathbb{R}^n. \forall a, b \in \mathbb{R}. f(ax + by) = af(x) + bf(y)
\]

The space of linear maps \( \mathcal{L}(V, W) \) from vector space \( V^n \) to vector space \( W^m \) is a set of linear maps from \( V \) to \( W \) closed under addition \((S + T)(v) = S(v) + T(v))\), multiplication by a scalar \((aT)v = a(Tv))\), and composition of linear maps.
Let $T \in \mathcal{L}(V, W)$, where $(e_1, \ldots, e_n)$ is the standard basis for $V$ and $(e'_1, \ldots, e'_m)$ is the standard basis for $W$. For each $e_k$, we can write $Te_k$ as a linear combination of $e'$

$$Te_k = a_{1,k}e'_1 + \cdots + a_{m,k}e'_m.$$ 

Hence the scalars $a_{j,k}$ completely determine the linear map. The matrix formed from these scalars is called the \textit{Matrix of $T$} and is denoted $\mathcal{M}(T)$.
Let $T \in \mathcal{L}(\mathbb{R}^2, \mathbb{R}^3)$ defined as follows:

$$T(x, y) = (x + 3y, 2x + 5y, 7x + 9y),$$

then with respect to the standard basis $T(1, 0) = (1, 2, 7)$ and $T(0, 1) = (3, 5, 9)$, so the matrix $T$ is the $3 \times 2$ matrix

$$M(T) = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 7 & 9 \end{bmatrix}$$
Matrix of an Operator

\[ \mathcal{M}(T + S) = \mathcal{M}(T) + \mathcal{M}(S) \]
\[ \mathcal{M}(cT) = c\mathcal{M}(T) \]
\[ \mathcal{M}(T \circ S) = \mathcal{M}(T) \ast \mathcal{M}(S) \]

The matrix of an \( n \)-dimensional vector is just a \( n \times 1 \) dimensional matrix.

Lemma

Suppose \( T \in \mathcal{L}(V, W) \) and \( e \) and \( e' \) are the standard basis of \( V \) and \( W \) respectively, then

\[ \mathcal{M}(Tv) = \mathcal{M}(T)\mathcal{M}(v) \]

for all \( v \in V \).
Operator of a Matrix

Define the operator $O_e$ such that $M(O(A)) = A$

$O_e$ is a bijective operator so there is only one such map and

$$M(O_e(A))(x) = A(x)$$

and

$$O_e(M(T))(x) = T(x)$$

The operations in the matrix space have a corresponding operation in $L(V,W)$
Applying the Matrix/Operator Correspondence

- Can define an operation in a matrix space using the associated operation in the linear map space.
- The existing PVS libraries have the definition and properties of inverse of a function.
  
  \[ T_A^{-1} \text { exists if } T_A \text { is a bijective function} \]
- Now the inverse of matrix is defined as follows,

  \[
  A^{-1} = \mathcal{M}(T_A^{-1})
  \]
PVS Linear Algebra Library

- We have seen the math
- Now let us look at our PVS Linear Algebra Library
- PVS is a higher-order logic
- Think of as higher-order typed functional programming language
  - OCAML, Haskell, . . .
Define a Matrix and Basic Operators

\[
\text{matrix}(n,m) : \text{type} = [[i : \text{below}[n], j : \text{below}[m]] \rightarrow \text{real}]
\]

The addition of two matrices is defined as follows

\[
+ : [\text{matrix}(n,m), \text{matrix}(n,m) \rightarrow \text{matrix}(n,m)]
= (\lambda (A : \text{matrix}(n,m), B : \text{matrix}(n,m))
  : (\lambda (i : \text{below}[n], j : \text{below}[m]) : A(i,j) + B(i,j)))
\]
Matrix Multiplication for Square Matrices

*: [matrix(n,n),matrix(n,n)→matrix(n,n)] =
    (λ (A: matrix(n,n), B: matrix(n,n)):
        (λ (i: below[n], j: below[n]):
            sigma[below[n]](0,n-1,λ(l:below[n]):A(i,l)*B(l,j))));

Multiplication of a matrix times a vector

*: [matrix(n,n),Vector[n]→Vector[n]] =
    (λ (A: matrix(n,n), x: Vector[n]):λ(j:below[n]):
        sigma[below[n]](0,n-1,λ(i:below[n]):A(i,j)*x(i)));
Basic Matrix Operation Lemmas

The \((i, j)\) element of the sum \(A + B\) is the sum of the \((i, j)\) elements.

**Lemma** \(\forall (A,B:\text{matrix}(n,m)):\ (A + B)(i,j) = A(i,j)+B(i,j)\)

The \((i, j)\) element of the difference \(A - B\) is the difference of the \((i, j)\) elements.

**Lemma** \(\forall (A,B:\text{matrix}(n,m)):\ (A - B)(i,j) = A(i,j) - B(i,j)\)

Matrix mult distributes over vector addition

**Lemma** \(m\models n \Rightarrow A*(x+y) = A*x + A*y\)
Bounded Input/Bounded Output Stability of Discrete Case

Let \( x(k + 1) = Ax(k) \)

\( x(k) \in \mathbb{R}^n, A \in \mathbb{R}^{n \times m} \)

Taking the real quadratic form \( V = x^T P x \), where \( P \) is a real symmetric matrix, as Lyapunov function

If \( \mathcal{E}_P = \{ x \in \mathbb{R}^n | x^T P x \leq 1 \} \)

then, the system is *bounded input/bounded output stable*
S-procedure theorem gives us an equivalence between inequalities

1. \( S_1 \): For all \( k = 1, 2, \ldots, N \), \( \sigma_k \geq 0 \implies \sigma_0 > 0 \)

2. \( S_2 \): There exists \( \tau_k \geq 0 \), \( k = 1, 2, \ldots, N \) such that

\[
\sigma_0(y) - \sum_{k=1}^{N} \tau_k \sigma_k(y) > 0, \ \forall y \in R^n
\]

The S-procedure theorem gives conditions under which \( S_1 \) is equivalent to \( S_2 \), \( S_1 \Leftrightarrow S_2 \)
Almost all control engineering is done in Matlab/Simulink.

- Basic data type is a matrix
- Scilab is an open source version of Matlab. The syntax is same modulo comments.

Implementation is usually done by either hand coding or an autocoder.

Existing autocoders are not verified nor produce code that can be verified.

In collaboration with Georgia Tech, we are producing an autocoder that produces annotated C code that is mechanically verified.

We only translate a subset

Written in Haskell
Examples of Scilab Code

A = [1, 2, -1; -2, -6, 4; -1, -2, -4];
b = [1; -2; 1];
x = A \ b;
L = A(2,3);
f = A(1:2,2:3);
Z = ones(2,1);
K = transpose(A);
Introduction Overview and Motivation Models Linear Algebra in PVS Taste of Linear Algebra PVS Implementation Motivation From Control Theory Matlab to C Translator Verification Using Frama-C/Jessie Runtime Verification: Copilot

Parser

AExpP. Exp5 ::= Exp5 "+" Exp6 ;
AexpM. Exp5 ::= Exp5 "-" Exp6 ;

MltExpM. Exp6 ::= Exp6 "*" Exp7 ;
MultExpEM. Exp6 ::= Exp6 ".*" Exp7 ;
MultExpD. Exp6 ::= Exp6 "/" Exp7 ;
MultExpED. Exp6 ::= Exp6 "./" Exp7 ;
MltExpLD. Exp6 ::= Exp6 "\\" Exp7 ;
MltExpCrt. Exp6 ::= Exp6 "^" Exp7 ;
MltExpECT. Exp6 ::= Exp6 ".^" Exp7 ;
Translator

- The translator maps Scilab operations on matrices to C functions
  - AllocM, AllocB, freemM, freeB, ones, submatrix, mult, mult_elem, div_elm, add, subtract, transpose, add_scalar, mult_scalar, div_scalar, copy, opposite, equals, and_Mat, or_Mat, equal_mat, ...

- The functions have ACSL annotations so as to verified using frama-C tool set
Building a Verified Library

- C library is annotated with contract information and loop invariants
- ANSI C Specification Language (ACSL)
- Frama-C’s Jessie tools used in verification
Frama-C

- Frama-C is a set of tools developed for analyzing C source code
  - Comprised of a number of independently developed tools
  - Value Analysis, Jessie, more components coming
- Developed at CEA in France (French Nuclear Agency)
- We use the Jessie tool for functional verification as well as verifying safety properties
ACSL

- First order logic
- Memory model
- Function contracts
- Hoare logics
  - requires P1; Requires P2;
  - assigns L1; assigns L2;
  - ensures E1; ensures E2;
- loop invariants
Contracts

- Specifying contracts for a library of matrix operations corresponding to the Scilab operators
- Treat those specifications as the axiomatic definition of the atomic matrix operation
- Translate the Scilab matrix operations to our function definitions
- The translation typically some matrix initialization and sequence of function calls
- Currently using the Hoare logics for procedure calls to reason about composition
Example Contract: Elementwise Matrix Multiplication

```c
/*@requires rowA > 0 && colA > 0 && rowB > 0 && colB > 0;
   @requires rowA == rowB && colA == colB;
   @requires (\valid(a+ (0..rowA-1)) && (\forall integer k;
       @ 0 <= k < rowA ==> \valid(a[k] +(0..colA-1) ) ));
   @requires (\valid(b+ (0..rowB-1)) && (\forall integer k;
       @ 0 <= k < rowA ==> \valid(b[k] +(0..colB-1) ) ));
   @requires (\valid(c+ (0..rowA-1)) && (\forall integer k;
       @ 0 <= k < rowA ==> \valid(c[k] +(0..colB-1) ) ));
   @ensures (0 <= i < rowA && 0 <=j < colB) ==> 
   @
   c[i][j] == a[i][j]*b[i][j];
@*/
```
float** mult_elm(float** a, int rowA, int colA,
    float** b, int rowB, int colB, float** c)
{
    int i=0, j=0;
    /*@ loop invariant (0 <= i <= rowA) && (\forall integer l, k; (0 <= l < colB && 0 <= k < i) ==>
        c[k][l] == a[k][l]*b[k][l]) ;
    @ loop variant rowA-i; */
    while (i < rowA)
    {
        j=0;
        ...
Annotated Code

```c
/*@ loop invariant (0 <= j <= colB) &&
   @ (\forall integer k; (0<=k<j) ==>
   @     c[i][k] == (a[i][k]*b[i][k]) ) ;
   @ loop variant  colB-j; */

while (j < colB)
{
    c[i][j]=a[i][j]*b[i][j];
    j = j+1;
}

/*@ assert \forall integer k; 0<=k<colB ==>
   @ (c[i][k] == a[i][k]*b[i][k]) ; */

i= i+1;

}

return c;
```
Jessie

- Interfaces to a number of SAT/SMT solvers and interactive provers
  - Alt-Ergo, Simplify, Yices, Z3, CVC
  - PVS, Coq, Isabelle
- Functional correctness
- Safety properties
- Float Models
  - math - unbounded reals
  - defensive - bounded reals
  - full - IEEE-754
  - multirounding - support non-IEEE-754 architectures
We have the linear algebra in PVS
We have the linear algebra libraries with ACSL contracts
We want to connect the two
Refinement calculi???
We currently do not deal with floating point and that’s critical
Lessons Learned

- Matrices are more complicated than arrays
- Row major works; column major does not
- Only works on a conservative subset of C
- Minimize memory allocation
- If you cannot figure out why the invariant doesn’t work for SMT solver, then move to an interactive prover. You will probably figure it out in that setting
- Not as well integrated with provers other than coq as we would like
Runtime Verification: The Basic Idea

- Joint work with Lee Pike (Galois), Sebastian Niller (NIA), and Robin Moriset (ENS Paris)
- System under observation (SUO) - system being monitored
- Correctness property \( \phi \)
- Monitors observe SUO and detect violations of \( \phi \)
  - Accept all traces admitting \( \phi \)
Our Focus

- Runtime verification (RV) is a very large field
- Focus on Java programs
  - Runtime Monitor and Checking (MaC) - Insup Lee, etc.
  - Monitor Oriented Programming (MoP) - Grigore Rosu, etc.
- Properties usually expressed in some variant of past-time temporal logic
- We focus on hard real-time embedded systems
- Periodic schedules
- Synchronized distributed systems
- C programs
Just the FaCTS

- Functionality: don’t change the functionality of the SUO
- Certifiability: don’t require recertification or make it easy
  - Don’t change source code
- Timing: don’t interfere with SUO’s timing
- SWaP: don’t exhaust size, weight, power reserves
Copilot is a Haskell eDSL targeted at runtime verification of hard real-time systems

Written as a Haskell library

Get parser, type-checker, etc. for free and more likely correct

A stream language
  - Streams have semantics of Haskell’s lazy streams (for free)
  - $\sigma(i)$ stream value at index $i$

A synchronous language defined by a set of stream equations
Closed Expressions

\[
\begin{align*}
\text{let } & \quad m_0 = \varB \, "m_0" \\
& \quad m_0 = [T, F] + + m_0
\end{align*}
\]

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_0)</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{let } & \quad m_1 = \VarI32 \, "m_1" \\
& \quad m_2 = \Var32 \, "m_2" \\
& \quad m_1 = \drop 1 m_2 \\
& \quad m_2 = [0, 1, 2] + + m_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m_1)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>\ldots</td>
</tr>
<tr>
<td>(m_2)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
Engine Monitor

If the majority of the three engine temperature probes has exceeded 250 degrees, then the cooler is engaged and remains engaged until the temperature of the majority of the probes drop to 250 degrees or less. Otherwise, trigger an immediate shutdown of the engine.
--external variable

\[ t_0 = \text{extW8} \ "temp\_probe\_0" \]
\[ t_1 = \text{extW8} \ "temp\_probe\_1" \]
\[ t_2 = \text{extW8} \ "temp\_probe\_2" \]
\[ \text{cooler} = \text{extB} \ "fan\_status" \]

-- Copilot variables

\[ \text{maj} = \text{varW8} \ "maj" \]
\[ \text{check} = \text{varB} \ "maj\_check" \]
\[ \text{overHeat} = \text{varB} \ "over\_heat" \]
\[ \text{monitor} = \text{varB} \ "monitor" \]
engineMonitor = do
  let temps = map (< 250) [t0, t1, t2]
  maj .= majority temps
  check .= aMajority temps maj
  overHeat \texttt {'ptltl'}
    \((\texttt {cooler} \texttt {|| maj} \texttt {&& check})
      \texttt {\texttt {'since'} not maj})
  monitor .= not overHeat
  trigger monitor "shutoff" void
Atom eDSL

- Atom is a Haskell eDSL that compiles to Embedded C
- Designed by Tom Hawking of Eaton to synthesize the embedded real-time control systems for off-road vehicles
- Automates aspects of scheduling and synchronization
- Used at Eaton in industrial vehicles
Copilot Compiler I

- Copilot is composed of approximately 3000 lines of Haskell code.
- Copilot specifications are translated into Atom.
- \[ \text{Copilot} \rightarrow \text{Atom} \rightarrow \text{C} \]
- Copilot type system is embedded in Haskell’s which provides a Hindley-Milner polymorphic type system, extended with type classes.
- Cuts development time substantially.
- Yes I am a big advocate of teaching and using typed functional languages.
Copilot Compiler II

- Synthesis algorithm compiles and schedules the monitor within the overall periodic schedule of the observed program
  - Makes it easy to use with microcontrollers
  - Can also use an RTOS
- Respects causality constraints - i.e., the data required to compute a value is available
- interferes with the program’s real-time constraints as little as possible
- These two criteria are handled at different levels of the compiler
Testbed

- A testbed to test algorithms and monitors
- Representative of fault-tolerant monitors
- $4 \times$ STM microcontrollers
- ARM Cortex M3 cores clocked at 72 Mhz
- $4 \times$ MPXV5004 differential pressure sensors
  - Senses dynamic and static pressure
  - Pitot tubes measure air speed
- Designed to fit in UAS (power, weight, etc.)
Conclusion

- Analyzing systems models and requirements is critical
- We cannot elide the engineering math
  - Big opportunity for decision procedures !!!!
- Typed functional programming is very useful
- Aerospace requires reasoning about functional correctness as well as safety properties of programs even though it’s harder
- RV can support heavier formal methods and testing
- Much work remains to connect the different levels of analysis
- NextGen air space management also has many verification challenges
  - A lot of physics and math remains to be formalized
  - New algorithms and protocols
  - Formal methods and simulation working as one team