Formal Methods for Aviation Software Design and Certification

PART II: Formal Methods

Gordon Stewart
COUNT Short Course Series
May 24-26
What Are Formal Methods?

DO-333:

“Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems.”

Or more simply: good engineering practice applied to the design and implementation of software systems.
What Are Formal Methods?

DO-333:

“Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems.”

Or more simply: **good engineering practice** applied to the design and implementation of software systems.

Formal Methods = Formal Models + Formal Analysis

Plus the tools that make such methods practical
Why Use Formal Methods?

DO-333:

• Unambiguously describe requirements of software systems
• Enable precise communication between engineers
• Provide verification evidence such as consistency and accuracy of a formally specified representation of software
• Provide verification evidence of the compliance of one formally specified representation with another
Concretely:

• Prove the absence of {buffer overflows, double frees, null pointer dereference, etc.} in a C program
• Prove worst-case execution bounds on a program in x86 assembly (e.g., in number of cycles)
• Prove information-flow guarantees (e.g., noninterference, or “no leakage” conditions) of a program that accesses both high- and low-sensitivity data
• Prove the absence of timing side channels (e.g., constant-time cryptography)
• Prove that a compiler never introduces bugs
• Prove that a C implementation of SHA256-HMAC is unbreakable up to cryptographic hardness assumptions
• ... and many more ...
“Formal”? 

Formal = mathematical or logical specification + mathematical or logical proof

• A specification is formal if its syntax and semantics are interpretable by a computer.

• A proof is formal if it’s checkable by a computer (against a computer-interpreted specification)
1. Identify the software system “under proof”

2. Write down a formal specification (in logic or mathematics)

3. Prove that the system satisfies the specification
Formal Methods From 10,000 Feet

1. **Identify the software system “under proof”**
2. Construct a formal model of (1)
3. Validate the model (e.g., by analysis or real-world experiment)
4. **Write down a formal specification (in logic or mathematics)**
5. Validate the specification (e.g., wrt. desired end-to-end properties)
6. **Prove that the (model of the) system satisfies the specification**
7. Identify the *Trusted Computing Base (TCB)* – what must a human inspect to be convinced of assurance case
Formal Methods Are Sound

• A *sound* method is one that never says “your system satisfies its spec” when it actually doesn’t

• For automated methods:
  – **NO** is acceptable (completeness/usability)
  – **I GIVE UP** is acceptable (lack of resources)
Trusted Computing Base (TCB)

Which components must be trusted (inspected) to have confidence in assurance case?

System Under Proof → Modeled By → Formal System Model → Validated Wrt. → Specification

Proof

Real World
A (Very Partial) Formal Methods History*

Early “Formal Methods”:
• 1800s BCE: Babylonian astronomy
• 360s BCE: Plato’s *Sophist* (“model” vs. “kind”)
• c. 300 BCE: Euclid’s *Elements* (axiomatic reasoning)

Early Formal Methods in Logic & Computer Science:
• 1900s: Hilbert (*Entscheidungsproblem*), Russell, Frege, Goedel, Curry, et al.
• 1936: Turing Machines
• 1936: Alonzo Church’s *lambda calculus*

*Cf. “A Brief History of ‘Formal Methods’”, B. Cohen*
“Traditional” Formal Methods

1950s: Floyd-Hoare Logic for reasoning about, and proving properties of, imperative programs (axiomatic semantics)

... 

1975: Dijkstra’s guarded commands language (GCL), predicate transformers and weakest preconditions

... 

1970s-today: operational and denotational semantics, process calculi, automated and interactive theorem provers, SAT and SMT solvers, separation logic, logics for concurrency and weak memory, verified compilers and operating systems, ...
Why Now?

• Computing power:

• plus the tooling to exploit it:

- Z3Prover / z3
- zChaff
- Coq
- alloy: a language & tool for relational models
Formal Methods Tools

Model Checking
Explore all paths through a model of the system, ensuring the system never enters a “bad” state (safety), or eventually enters a “good” state (liveness)

Type Systems & Static Analysis
Inspect the syntax of a program in order to prove the absence of certain (classes of) bugs

Abstract Interpretation
Evaluate a program wrt. an abstract domain over the program’s values, e.g., value ranges $[lo,hi]$

Theorem Proving
Use deductive reasoning to build models of (and prove properties of) systems
Expressivity vs. Interaction

Expressivity

Human Interaction

Interactive Theorem Proving

PVS

Agda

Full Functional Correctness

Model Checkers / Abstract Interpreters

Astrée

Type Systems

Astrée

Java

CBMC
Expressivity vs. Interaction

Interactive Theorem Proving

PVS

Agda

Full Functional Correctness

Model Checkers / Abstract Interpreters

Astrée

Type Systems

We’ll explore both CBMC and Coq in today’s case study
Plan for Part II

Intro. to Formal Methods

Formal Methods Tools
  • Type Systems, Static Analyses
  • Abstract Interpretation, Model Checking
  • Interactive Theorem Proving

Case Study: Forward Error Correction
  • C Bounded Model Checking
  • Interactive Theorem Proving
  • https://github.com/gstew5/count17

Looking Ahead
  • Mechanized End-to-End Systems
CASE STUDY:
FORWARD ERROR CORRECTION
WITH REPLICATION CODES
The Setting

Flip some bits in codes transmitted on channel

Syndrome (e.g., cosmic ray)

Encode

Channel

Decode

bit

bit

code

length N
Replication Codes

- Introduce redundancy by replicating bits: A simple coding scheme suitable for demonstration
- But note: highly inefficient (N [=3, below] times reduction in throughput)
- Other better algorithms:
  - Hamming codes (e.g., [7,4])
  - Reed-Muller, Reed-Solomon, etc.

code words = \{000, 111\}
code(0) = \text{000}
code(1) = \text{111}
decode(xyz) = \text{majority vote}
E.g., decode(001) = 0
decode(101) = 1
Replication Codes

• Introduce redundancy by replicating bits: A simple coding scheme suitable for demonstration

• But note: highly inefficient ($N \leq 3$, below] times reduction in throughput)

• Other better algorithms:
  • Hamming codes (e.g., [7,4])
  • Reed-Muller, Reed-Solomon, etc.

\[
\text{code words} = \{000, 111\} \\
\text{code}(0) = 000 \\
\text{code}(1) = 111 \\
\text{decode}(xyz) = \text{majority vote} \\
\text{E.g., decode}(001) = 0 \\
\text{decode}(101) = 1
\]
typedef bool CODE[N];
CODE code0 = {0,0,...,0};
CODE code1 = {1,1,...,1};
...
CODE* encode(bool b) {...}
bool decode(CODE c) {...}

Formal Model: encode, decode, syndrome

Validate Model: C Bounded Model Checking

Analyze TCB

Prove that spec holds of model

Formal Specification

The Coq Proof Assistant
Formal Methods From 10,000 Feet

1. **Identify the software system “under proof”**
2. Construct a formal model of (1)
3. Validate the model (e.g., by analysis or real-world experiment)
4. **Write down a formal specification (in logic or mathematics)**
5. Validate the specification (e.g., wrt. desired end-to-end properties)
6. **Prove that the (model of the) system satisfies the specification**
7. Identify the *Trusted Computing Base (TCB)* – what must a human inspect to be convinced of assurance case
Example: Error-Correcting Codes

1. **Identify the software system “under proof”**

   C implementation of a simple error-correcting code (encode by replication, decode by majority vote)

```c
#define N 3
typedef bool CODE[N];

//initialized below in main
CODE code0; // {0,0,0}
CODE code1; // {1,1,1}

CODE* encode(bool b) {
    if (!b) return &code0;
    return &code1;
}
```
1. **Identify the software system “under proof”**

C implementation of a simple error-correcting code (encode by replication, decode by majority vote)

```c
bool decode(CODE c) {
    int d0 = hamming(code0, c);
    int d1 = hamming(code1, c);

    if (d0 < d1) { return 0; }
    else if (d0 > d1) { return 1; }
    else assert(0); // decoding failed
}
```
1. Identify the software system “under proof”

C implementation of a simple error-correcting code
(encode by replication, decode by majority vote)

```c
// Hamming distance between c and d
int hamming(CODE c, CODE d) {
    int sum = 0;
    for (int i=0; i < N; i++) {
        sum += c[i] ^ d[i];
    }
    return sum;
}
```
2. Construct a formal model of (1)

(1) $\text{code} = \text{bitvector}[N], c_0 = 00 \ldots 0, c_1 = 11 \ldots 1$

(2) $\text{encode} : \text{bit} \rightarrow \text{code}$
   \[ \forall b. \text{encode}(b) = bb \ldots b \]

(3) $\text{decode} : \text{code} \rightarrow \text{bit}$
   \[ \forall c, (\text{Ham}(c, c_0) < \text{Ham}(c, c_1) \rightarrow \text{decode}(c) = 0) \text{ and } 
   (\text{Ham}(c, c_1) < \text{Ham}(c, c_0) \rightarrow \text{decode}(c) = 1) \]
2. Construct a formal model of (1)

(1) \( code = bitvector[N], c_0 = 00 \ldots 0, c_1 = 11 \ldots 1 \)

(2) \( encode : bit \rightarrow code \)
\[ \forall b. encode(b) = bb \ldots b \]

(3) \( decode : code \rightarrow bit \)
\[ \forall c, (\text{Ham}(c, c_0) < \text{Ham}(c, c_1)) \]
\[ (\text{Ham}(c, c_1) < \text{Ham}(c, c_0)) \]

(4) \( syndrome : code \rightarrow code \)
\[ \forall c. \text{Ham}(c, syndrome(c)) \leq \text{floor} \left( \frac{N}{2} \right) \]

To scale up, one might use a more realistic probabilistic model like a Binary Symmetric Channel.
3. Validate the model (e.g., by analysis or experiment)

Use a *bounded model checker* (we’ll use CBMC) to validate the models of *encode* and *decode* (analysis)

Use data to validate that the environment may never flips more than \(\text{floor} \left( \frac{N}{2} \right)\) bits (experiment)

Or in a more realistic model, that the probability of a cosmic-ray bit flip is at most \(10^{-9}\) ...
C Bounded Model Checking using CBMC:

1. Encode the model as a C program enhanced with:
   - `nondet`: nondeterministically generate a value

   ```c
   int x = nondet_int(); //int in [min_signed,max_signed]
   ```

   - `assume`: impose restrictions on (generated) values

   ```c
   assume(0 <= x && x < 100); //x in [0, 100)
   ```

   - `assert`: communicate properties to be checked

   ```c
   assert(x < 150); //true given assume stmt. above
   ```

2. `cbmc <file.c>`
Modeling Encode

\[ \forall b. encode(b) = bb \ldots b \]

```
//encode
bool b1 = nondet_bit();
int i1 = nondet_int();
assume(0 <= i1 && i1 < N);
assert((*encode(b1))[i1] == b1);
```

“For all b1, i1 s.t. 0 <= i1 and i1 < N, *encode(b1)[i1] equals b1.”
Modeling Decode

\forall c, (\text{Ham}(c, c_0) < \text{Ham}(c, c_1) \rightarrow \text{decode}(c) = 0) \text{ and } \text{(Ham}(c, c_1) < \text{Ham}(c, c_0) \rightarrow \text{decode}(c) = 1)

//decode
CODE c2;
for (int i = 0; i < N; i++) c2[i] = nondet_bit();

Case 1: if (hamming(c2, code0) < hamming(c2, code1))
assert(decode(c2) == 0);

Case 2: if (hamming(c2, code1) < hamming(c2, code0))
assert(decode(c2) == 1);

//an additional property true only if N odd:
assert(hamming(c2, code1) != hamming(c2, code0));
Modeling the Syndrome

∀c. Ham(c, syndrome(c)) ≤ floor\(\left(\frac{N}{2}\right)\)

CODE channel;
CODE* syndrome(CODE c) {
  //copy code c into channel
  for (int i = 0; i < N; i++) {
    channel[i] = c[i];
  }
  //apply syndrome
  for (int n = 1; n <= N/2; n++) {
    int i = nondet_int();
    assume(0 <= i && i < N);
    channel[i] = nondet_bit();
  }
  return &channel;
}

Worst case: flip N/2 distinct bits
Validating: C Code Meets Its Specification (Formal Model)

> cbmc ecc.c

Generated 9 VCC(s), 9 remaining after simplification
Passing problem to propositional reduction
converting SSA
Solving with MiniSAT 2.2.1 with simplifier
4201 variables, 5053 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 0.017s

** Results:
[main.assertion.1] assertion (*encode(b1))[i1] == b1: SUCCESS
[main.assertion.2] assertion hamming(c1, *syndrome(c1)) <= N/2: SUCCESS
[main.assertion.3] assertion decode(c2) == 0: SUCCESS
[main.assertion.4] assertion decode(c2) == 1: SUCCESS
[main.assertion.5] assertion hamming(c2, code1) != hamming(c2, code0): SUCCESS

** 0 of 5 failed (1 iteration)
VERIFICATION SUCCESSFUL
Identify the software system “under proof”

C implementation of a simple error-correcting code (encode by replication, decode by majority vote)

```c
bool decode(CODE c) {
    int d0 = hamming(code0, c);
    int d1 = hamming(code1, c);
    if (d0 < d1) { return 1; }
    else if (d0 > d1) { return 0; }
    else assert(0); // decoding failed
}
```

Return bit \( b \) that maximizes \( \text{Hamming}(xyz, \text{code}(b)) \).
Validating: C Code Meets Doesn’t Meet Its Spec. (!)

> cbmc ecc.c

Generated 9 VCC(s), 9 remaining after simplification
Passing problem to propositional reduction
converting SSA
Solving with MiniSAT 2.2.1 with simplifier
4201 variables, 5053 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 0.017s

** Results:
[main.assertion.1] assertion (*encode(b1))[i1] == b1: SUCCESS
[main.assertion.2] assertion hamming(c1, *syndrome(c1)) <= N/2: SUCCESS
[main.assertion.3] assertion decode(c2) == 0: FAILURE
[main.assertion.4] assertion decode(c2) == 1: FAILURE
[main.assertion.5] assertion hamming(c2, code1) != hamming(c2, code0): SUCCESS

** 2 of 5 failed (3 iterations)
VERIFICATION FAILED
Bounded Model Checking Under the Hood

• Reduce the verification problem to an equivalent SAT instance, pass to SAT solver (e.g., MiniSAT)
  • Can deal only with finite control flow

• **Loops?** Unfold up to some *bound*:

  while e { c } = if e then (c; while e { c }) else skip

• **Consequence:** bounded model checking no longer sound for programs with unbounded loops

• **Performance:** scales with size of program
Example: Error-Correcting Codes

1. **Identify the software system “under proof”**

   C implementation of a simple error-correcting code (encode by replication, decode by majority vote).

   ```c
   #define N 3
   typedef bool CODE[N];
   //initialized below in main
   CODE code0; // {0,0,0}
   CODE code1; // {1,1,1}

   CODE* encode(bool b) {
      if (!b) return &code0;
      return &code1; }
   ```

   We proved the spec. only wrt. a particular N (=3)
Bounded Model Checking Under the Hood

Size of generated SAT instance for $N=\{3, 6, 12, 24, 48\}$

<table>
<thead>
<tr>
<th>$N$</th>
<th>Variables</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2883</td>
<td>3447</td>
</tr>
<tr>
<td>6</td>
<td>3499</td>
<td>5538</td>
</tr>
<tr>
<td>12</td>
<td>5116</td>
<td>11994</td>
</tr>
<tr>
<td>24</td>
<td>10518</td>
<td>35730</td>
</tr>
<tr>
<td>48</td>
<td>27603</td>
<td>117762</td>
</tr>
</tbody>
</table>

~0.1 million clauses
typedef bool CODE[N];
CODE code0 = {0,0,...,0};
CODE code1 = {1,1,...,1};
...
CODE* encode(bool b) {...}
bool decode(CODE c) {...}
Interactive Theorem Proving

ProofIDE

Error Message

Error?

Proof Script

OK?

Kernel

Done
The Coq Proof Assistant

CoqIDE
Iteratively step through programs / proofs

Green means “processed”
Interactive Theorem Proving

1. Re-encode the model in Coq

(* decode *)

Variable decode : code -> bool.
Hypothesis decode_ok : 
 forall c : code,
    (hamming c code0 < hamming c code1 -> decode c = false) /
    (hamming c code1 < hamming c code0 -> decode c = true).

2. Use CoqIDE to interactively prove the following specification of the model:

\( \text{decode} \circ \text{syndrome} \circ \text{encode} = \text{id} \)

equivalent to

\( \forall b. \text{decode} \left( \text{syndrome} \left( \text{encode} \left( b \right) \right) \right) = b \)
Interactive Theorem Proving

1. Re-encode the model in Coq

(* decode *)

Variable decode : code -> bool.

Hypothesis decode_ok :
    forall c : code,
    (hamming c code0 < hamming c code1 -> decode c = false) /
    (hamming c code1 < hamming c code0 -> decode c = true).

Wrt. our model of the environment (syndrome), the encoder/decoder present the abstraction of a perfect communication channel

2. Use CoqIDE to interactively prove the specification of the decoder:

\[ \text{decode} \circ \text{syndrome} \circ \text{encode} = \text{id} \]

\[ \forall b. \text{decode} \left( \text{syndrome} \left( \text{encode} (b) \right) \right) = b \]
Section ecc.

Variable \( N : \text{nat.} \)

Hypothesis \( N\_\text{odd} : \text{odd } N \).

Assume an arbitrary odd natural \( N \)

Definition \( \text{code} := \{ \text{ffun } '{I}_N \rightarrow \text{bool} \} \).

Codes are functions from \([0, N)\) to bool

Definition \( \text{code0} : \text{code} := \text{finfun} \ (\text{fun } i : '{I}_N \Rightarrow \text{false}) \).

\( \text{code0} \) is the constant function always returning \text{false}

Definition \( \text{code1} : \text{code} := \text{finfun} \ (\text{fun } i : '{I}_N \Rightarrow \text{true}) \).

\( \text{code1} \) is the constant function always returning \text{true}
Variable \texttt{encode} : bool \rightarrow \texttt{code}.

Uninterpreted function mapping \texttt{bools} to \texttt{codes}

Hypothesis \texttt{encode\_ok} :
   \texttt{forall} (b:bool) (i:'I_N), (\texttt{encode} b) i = b.

For all \texttt{bools} b, naturals i in \([0, N)\).

\texttt{encode}(b)[i] = b

Returns higher-order function (of type \texttt{code}) mapping \([0, N)\) to \texttt{bool}
(* hamming distance *)

Definition hamming (c1 c2 : code) : nat :=
\sum_(i : 'I_N | c1 i != c2 i) 1.

//Hamming distance between c and d
int hamming(CODE c, CODE d) {
    int sum = 0;
    for (int i=0; i < N; i++) {
        sum += c[i]^d[i];
    }
    return sum;
}
Decode

(* hamming distance *)

Definition hamming (c1 c2 : code) : nat :=
\sum_(i : 'I_N | c1 i != c2 i) 1.

(* decode *)

Variable decode : code -> bool.
Hypothesis decode_ok : forall c : code,
(hamming c code0 < hamming c code1 -> decode c = false) /
(hamming c code1 < hamming c code0 -> decode c = true).

∀c, (Ham(c,c₀) < Ham(c,c₁) -> decode(c) = 0) and
(Ham(c,c₁) < Ham(c,c₀) -> decode(c) = 1)
Syndrome

(* syndrome *)

Variable syndrome : code → code.

Hypothesis syndrome_ok :
  forall c : code, 
    hamming c (syndrome c) ≤ Nat.div2 N.

∀c. Ham(c, syndrome(c)) ≤ floor\left(\frac{N}{2}\right)
Theorem decode_encode_id :
  forall b : bool, decode (syndrome (encode b)) = b.

Proof.

... ?

Qed.
Key Lemmas

**Lemma hamming_code0_code1_N**: For all code, $c$, $\text{hamming}(c, \text{code0}) + \text{hamming}(c, \text{code1}) = N$.

For any code $c$, Hamming distance to $\{0,0,...,0\}$ plus Hamming distance to $\{1,1,...,1\} = N$.
Key Lemmas

Lemma \text{hamming\_code0\_code1\_N}:
\begin{align*}
\text{forall } c : \text{code}, \\
\text{hamming } c \text{ code0} + \text{hamming } c \text{ code1} = N.
\end{align*}

For any code \(c\), Hamming distance to \(\{0,0,\ldots,0\}\) plus Hamming distance to \(\{1,1,\ldots,1\}\) = \(N\)

Lemma \text{encode\_range}:
\begin{align*}
\text{forall } b, \\
(b=\text{false} \land \text{encode } b = \text{code0}) \lor \\
(b=\text{true} \land \text{encode } b = \text{code1}).
\end{align*}

Convenience lemma for structuring case analysis over input bit \(b\)
**Case 1:** \( b = false, encode(b) = code0 \)

**N.T.S.** \( decode(syndrome(code0)) = false \)

1. \( \exists N'. N = 2N' + 1 \)  \( \text{Nodd} \)

2. \( hamming(syndrome(code0), code0) \leq \) \( Nat. div2 (2N' + 1) = N' \)  \( \text{syndrome_ok, 1} \)

3. \( hamming(syndrome(code0), code0) + \) \( hamming(syndrome(code0), code1) = 2N' + 1 \)  \( \text{lemma, 1} \)

4. \( hamming(syndrome(code0), code0) < \) \( hamming(syndrome(code0), code1) \)  \( \text{2, 3} \)

**QED.** from 4, decode_ok
Theorem decode_encode_id :
  forall b : bool, decode (syndrome (encode b)) = b.
Proof.
  move => b; move: (syndrome_ok (encode b)).
  case: (decode_ok (syndrome (encode b))).
  case: (encode_range b) => [] [][] -> ->.
  { move => H _ => H2; rewrite H => //; clear H.
    move: (hamming_code0_code1_N (syndrome code0)) => H3.
    case: N_odd => N' H4; rewrite H4 in H2, H3; clear H4.
    rewrite Nat.add_1_r in H2, H3;
    rewrite Nat.div2_succ_double in H2.
    move: (leP H2) => H2'; apply/ltP; rewrite -plusE in H3.
    rewrite hamming_comm in H2'; omega. }
  (*symmetric case elided*)
Qed.
typedef bool CODE[N];
CODE code0 = {0,0,...,0};
CODE code1 = {1,1,...,1};
...
CODE* encode(bool b) {...}
bool decode(CODE c) {...}

Case Study Breakdown

Formal Model: encode, decode, syndrome

Analyze TCB

Validate Model: C Bounded Model Checking

Formal Specification

Prove that spec holds of model

The Coq Proof Assistant
Trusted Computing Base

TCB: those components that must be inspected/validated by a human to have confidence in assurance case (end-to-end spec holds of deployed system)

1. Models
   1. Specification (is this the right spec.?)
   2. Syndrome model (is this the right model of environment?)
   3. CBMC model ↔ Coq model (do the models correspond?)

2. Deployment
   1. Compiler, gcc/clang (to compile application)
   2. Assembler (to generate machine code)
   3. OS on which code is run (program loader, etc.)
   4. Hardware on which OS is run, wrt. which code is compiled
   5. ...?
3. Tools [tool qualification]
   1. CBMC (including all decision procedures, etc.)
   2. Coq lexer, parser, proof checker (kernel)

4. Tools used to build/use tools
   1. gcc/clang (to build CBMC)
   2. OCaml compiler and runtime (to compile and run Coq)
   3. OS on which tools are run (program loader, etc.)
   4. hardware on which OS is deployed

5. Security Questions
   1. Is the code I’m running the same as the code I verified? (software attestation)
   2. Is the hardware on which I’ve deployed my code the same as the hardware I validated? (hardware attestation, trojans)
   3. Side channels: Can an attacker break my system by “sliding underneath” the model?
LOOKING AHEAD: END-TO-END MECHANIZED SYSTEMS
Systems of the Future

**Thesis:**
The next generation of high-assurance systems will be

- **built from mechanically verified components:**
  - application programs,
  - compilers,
  - assemblers,
  - operating systems,
  - CPUs, etc.

- **aligned end-to-end at interface boundaries,** thus
- **drastically reducing the size of such systems’ TCBs.**
Building Blocks

CompCert Certified C Compiler
http://compcert.inria.fr/

CertiKOS
Certified Kit Operating System
http://flint.cs.yale.edu/certikos/

Projects in which I have been involved

Verifiable C
language & program logic

VST retargetable Separation Logic

COMPCERT verified C compiler (from INRIA)

verified machine language program

Verified Software Toolchain (VST)
http://vst.cs.princeton.edu

L4.verified (NICTA)
End-to-End Forward Error Correction

*about behavior of ecc.o running as CertiKOS process, wrt. model of CPU hardware (no need to trust intermediate models)
Intermediate Specs Need Not Be Trusted

- **ecc.c**
  - VST Program Logic
  - Property $P$
  - Proof
  - CompCert C Compiler
  - Proofs:
    - $P$ holds of ecc.c, wrt. (a model of) the C89 semantics
    - $P'$ holds of ecc.s, wrt. (a model of) the x86 semantics

- **ecc.s**
  - Untrusted intermediate spec.

- Trusted
  - (model of) C Semantics
  - (model of) x86 semantics
Course Summary

Intro. to Formal Methods

Formal Methods Tools

- Type Systems, Static Analyses
- Abstract Interpretation, Model Checking
- Interactive Theorem Proving

Case Study: Forward Error Correction

- C Bounded Model Checking
- Interactive Theorem Proving

Looking Ahead

- Mechanized End-to-End Systems
What Are Formal Methods?

DO-333:

“Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems.”

Or more simply: **good engineering practice** applied to the design and implementation of software systems.
Resources

Case Study:
https://github.com/gstew5/count17

Thanks!
Gordon Stewart (gstewart@ohio.edu)
http://ace.cs.ohio.edu/~gstewart