

Analyzing and Assigning Meaning to Autocoded Control Software

Eric Feron
Aerospace Engineering
Georgia Tech

November 3, 2009

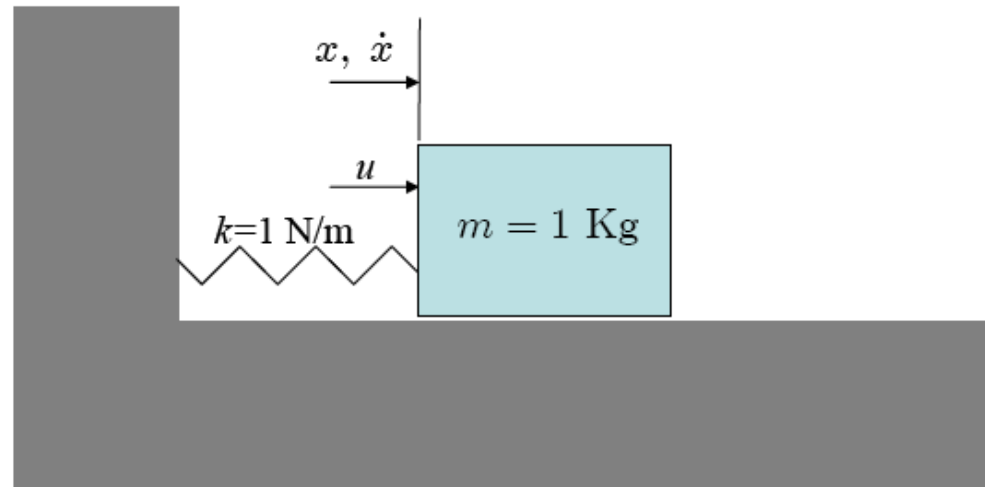
Take-Home Message

We've got proofs for our control systems,
so let's use them beyond cool research
papers, inside software

Outline

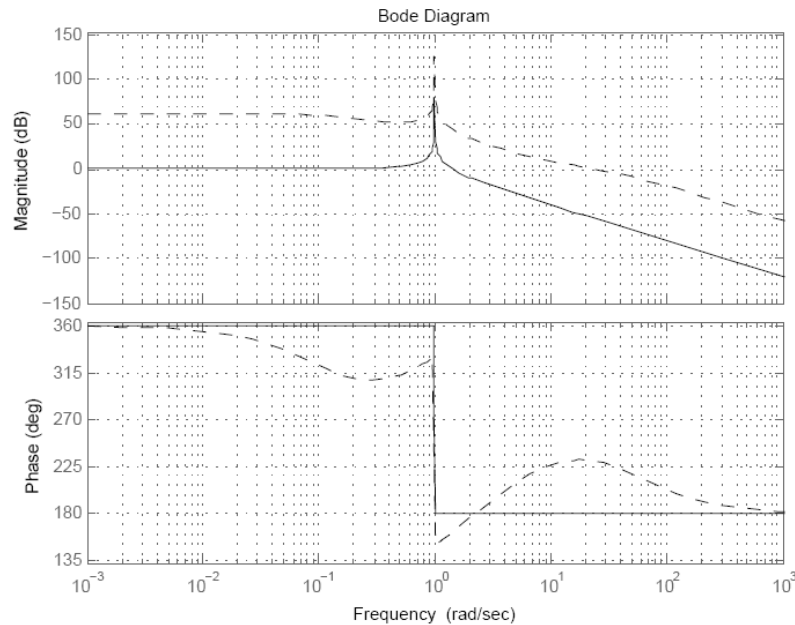
- A simple control example
- Stability and performance analyses
- Why code-level analyses?
- Hoare logic and partial correctness
- Analysis of controller implementation
- Closed-loop system analysis

A simple control example



$$\frac{d}{dt} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u, \quad x(0) = x_0, \dot{x}(0) = \dot{x}_0$$
$$y = [0 \ 1] \begin{bmatrix} x \\ \dot{x} \end{bmatrix}.$$

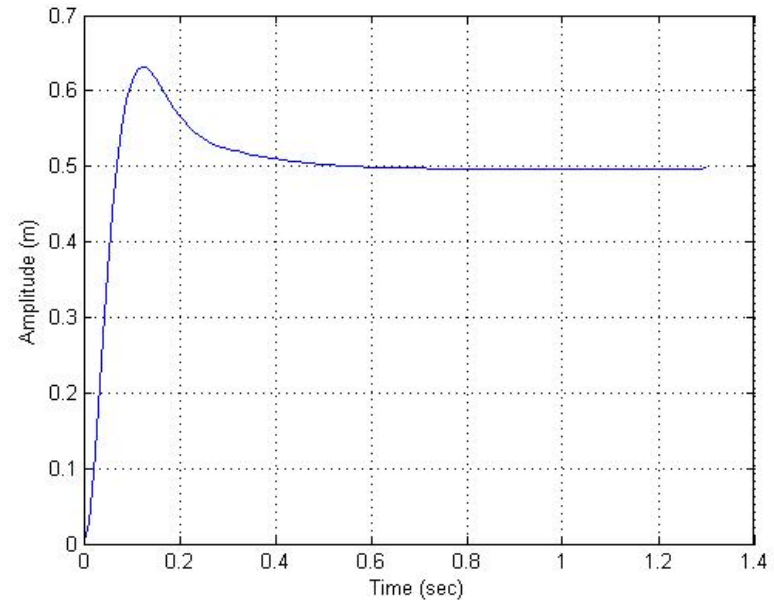
A simple control example (Ct'd)



$$\tilde{y}(t) = \mathbf{SAT}(y(t)),$$

$$u(s) = 128 \frac{s+1}{s+0.1} \frac{s/5+1}{s/50+1} \tilde{y}(s),$$

Step response



Controller implementation

$$\tilde{y}(t) = \mathbf{SAT}(y(t)),$$
$$u(s) = 128 \frac{s+1}{s+0.1} \frac{s/5+1}{s/50+1} \tilde{y}(s),$$

$$\frac{d}{dt} x_c = \begin{bmatrix} -50.1 & -5.0 \\ 1.0 & 0.0 \end{bmatrix} x_c + \begin{bmatrix} 100 \\ 0 \end{bmatrix} \mathbf{SAT}(y) \quad \text{State-space realization}$$
$$u = -[564.48 \ 0] x_c + 1280 \mathbf{SAT}(y).$$

$$x_{c,k+1} = \begin{bmatrix} 0.499 & -0.050 \\ 0.010 & 1.000 \end{bmatrix} x_{c,k} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{SAT}(y_k) \quad \begin{array}{l} \text{Discrete time} \\ \text{Implementation} \\ 100\text{Hz} \end{array}$$
$$u_k = -[564.48 \ 0] x_{c,k} + 1280 \mathbf{SAT}(y_k)$$

Controller Program

```
1:  A = [0.4990, -0.0500;  
        0.0100, 1.0000];  
2:  C = [-564.48, 0];  
3:  B = [1;0];D = 1280;  
4:  x = zeros(2,1);  
5:  while 1  
6:    y = fscanf(stdin,"%f");  
7:    y = max(min(y,1),-1);  
8:    u = C*x + D*y;  
9:    fprintf(stdout,"%f\n",u);  
10:  x = A*x + B*y;  
11:  end
```

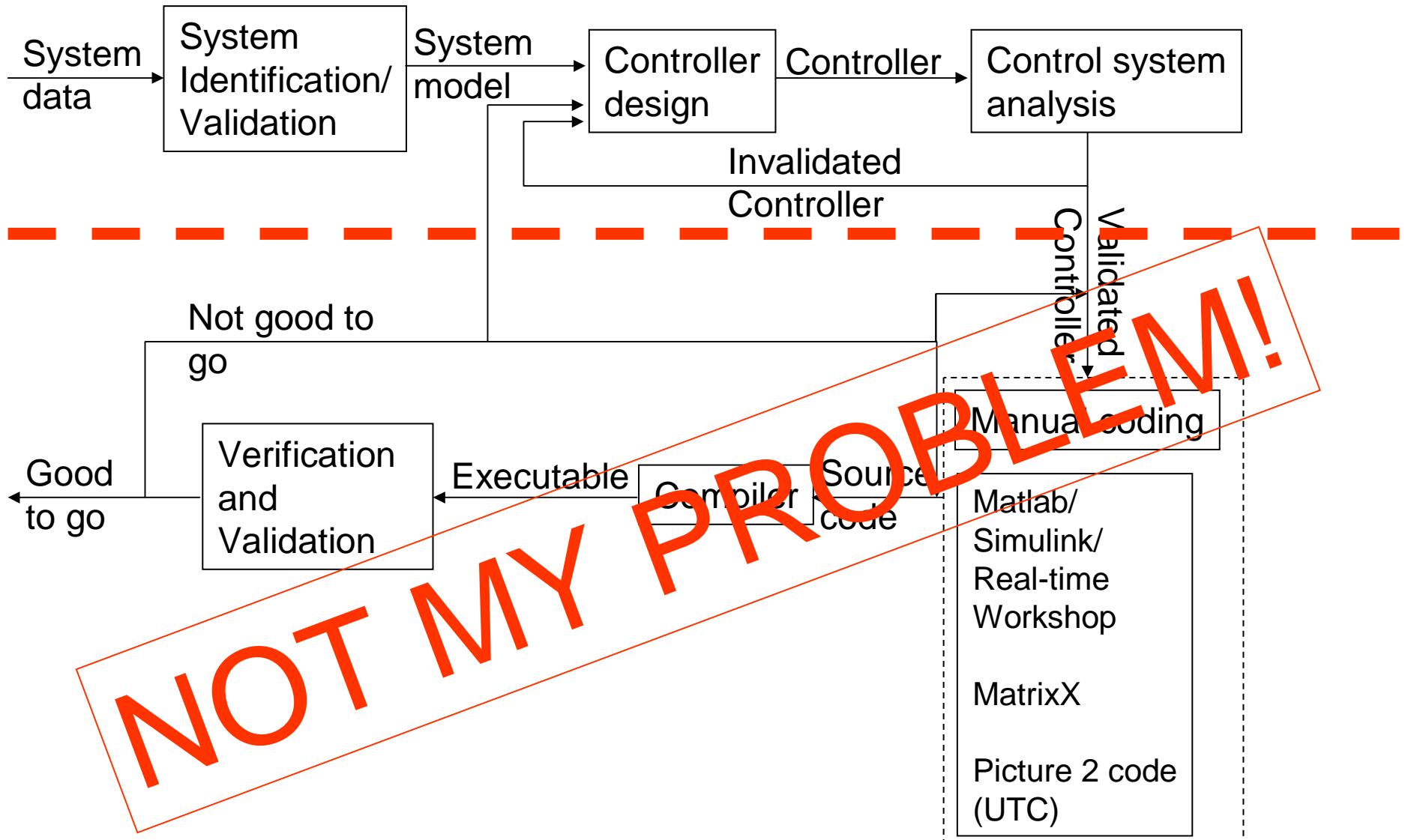
MATLAB

Controller Program

```
1: int main(int argc, char *argv[])
2: {
3:     double *x[2], *xb[2], y, u;
4:     x[0] = 0;
5:     x[1] = 0;
6:     while(1){
7:         fscanf(stdin,"%f",&y);
8:         if (y >1){
9:             y=1
10:        }
11:        if (y<-1){
12:            y=-1
13:        }
14:        u = -564.48*x[0]+1280*y;
15:        fprintf(stdout,"%f\n",u);
16:        xb[0]=x[0];
17:        xb[1]=x[1];
18:        x[0] := 0.4990*xb[0]-0.0500*xb[1]+y;
19:        x[1] := 0.01*xb[0]+xb[1];
20:    }
21:}
```

C

Control system as seen by control engineers



Code-level analyses of control software

- Most significant contribution is from Patrick Cousot's group at Ecole Normale Supérieure, Paris.
- Abstract interpretation aims at capturing semantics of programs
- Most important application is ASTREE analyzer for Airbus A380 control code.
- From Feret, "Static Analysis of Digital Filters", 2004 (also with ASTREE).

A simplified second order filter relates an input stream E_n to an output stream defined by:

$$S_{n+2} = aS_{n+1} + bS_n + E_{n+2}.$$

Thus we experimentally observe, in Fig. 4, that starting with $S_0 = S_1 = 0$ and provided that the input stream is bounded, the pair (S_{n+2}, S_{n+1}) lies in an ellipsoid. Moreover, this ellipsoid is attractive, which means that an orbit starting out of this ellipsoid, will get closer of it. This behavior is explained by Thm. 5.

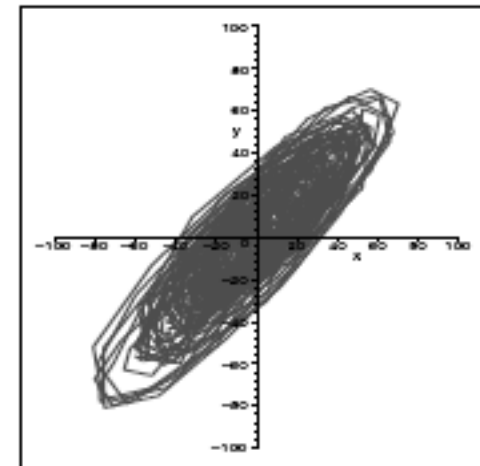


Fig. 4. Orbit.

Questions about control system and its implementation

- Closed-Loop Stability
- Closed-Loop Performance
- Run-time errors
- Timing performance and scheduling correctness

Making sense out of computer programs

- Consists of asking: “Where can the program go? Does it do what it’s supposed to do?”
- Well-known limits to what can be done.
- Several options are available: “Formal Methods”
 - Model checking
 - Hoare logic
 - Abstract interpretation

Desirable attributes of “program proofs”

- Must be expressive enough to tell nontrivial statements about program
- Must be “elementary enough” so that each “proof element” can be verified using only local program elements. “Line-by-line verification”.

Hoare logic: While programs

- Assignments $y:=t$
- Composition $S1; S2$
- If-then-else if e then $S1$ else $S2$ fi
- While e do S od
- Hoare logic consist of instrumenting each line with “what it does to the state-space”

assignment \Rightarrow $\begin{array}{l} \{\text{state-space statement 1}\} \leftarrow \textit{precondition} \\ \text{assignment} \\ \{\text{state-space statement 2}\} \leftarrow \textit{postcondition} \end{array}$

Compositionality:

$\{pre_1\}loc\ 1\{post_1\}; \{pre_2\}loc\ 2\{post_2\} \wedge (\{post_1\} \rightarrow \{pre_2\})$
 $\Rightarrow \{pre_1\}loc\ 1;loc\ 2\ \{post_2\}$

See Peled, 2001, Monin, 2000

Example: Capturing controller behavior

The control-systemic way

$$\begin{aligned}x_{c,k+1} &= \begin{bmatrix} 0.499 & -0.050 \\ 0.010 & 1.000 \end{bmatrix} x_{c,k} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{SAT}(y_k) \\ u_k &= -[564.48 \ 0] x_{c,k} + 1280 \mathbf{SAT}(y_k)\end{aligned}$$

Assume the controller state is initialized at $x_{c,0} = 0$

What range of values could be reached by the state $x_{c,k}$ and the control variable u_k ?

There is a variety of options, including computation of ∞ norms.

A Lyapunov-like proof (from Boyd *et al.*, Poola):

The ellipsoid $\mathcal{E}_P = \{x \in \mathbf{R}^2 \mid x^T P x \leq 1\}$. $P = \begin{bmatrix} 0.0300 & 0.2000 \\ 0.2000 & 10.0000 \end{bmatrix}$

is invariant. None of the entries of x exceeds 7 in size.

How do you say the same thing about the *program* that implements the controller?

A Matlab program

```

1: A = [0.4990, -0.0500;
        0.0100, 1.0000];
2: C = [-564.48, 0];
3: B = [1;0];D = 1280;
4: x = zeros(2,1);
5: while 1
6:   y = fscanf(stdin,"%f");
7:   y = max(min(y,1),-1);
8:   u = C*x + D*y;
9:   fprintf(stdout,"%f\n",u);
10:  x = A*x + B*y;
11: end

```

```

{true}
4: x = zeros(2,1)

```

```
{x ∈ EP}.
```

```
5: while 1
```

```
{x ∈ EP}
```

```
6: y = fscanf(stdin,"%f")
```

```
{x ∈ EP}
```

```
7: y = max(min(y,1),-1);
```

```
{x ∈ EP, y2 ≤ 1}
```

```
8: u = C*x+D*y;
```

```
{x ∈ EP, u2 ≤ 2(CP-1CT + D2), y2 ≤ 1}
```

```
9: fprintf(stdout,"%f\n",u)
```

```
{x ∈ EP, y2 ≤ 1}
```

```
{Ax + By ∈ EP, y2 ≤ 1,
u2 ≤ 2(CP-1CT + D2)}
```

```
9: fprintf(stdout,"%f\n",u)
```

```
{Ax + By ∈ EP, y2 ≤ 1}
```

```
10: x = A*x + B*y;
```

```
{x ∈ EP}
```

```
11:end
```

```
{false}
```

?

Critical step

$$\{x \in \mathcal{E}_P, y^2 \leq 1\} \Rightarrow \{Ax + By \in \mathcal{E}_P, y^2 \leq 1\}.$$

is true for the particular instances of A, B, P . (ellipsoid invariance condition)/

-Either leave it as is to check via automated checker that can handle systems of polynomial inequalities (eg Zumkeller, They 2008)

-Or provide additional proof element:

$$\forall(x, y) (Ax + By)^T P(Ax + By) - 0.01x^T Px - 0.99y^2 \leq 0.$$

$$\{x \in \mathcal{E}_P, y^2 \leq 1, (Ax + By)^T P(Ax + By) - 0.01x^T Px - 0.99y^2 \leq 0\} \\ \Rightarrow \{Ax + By \in \mathcal{E}_P, y^2 \leq 1\},$$

Commented code

```
{true}
1:  A = [0.4990, -0.0500; 0.0100, 1.0000];
{true}
2:  C = [-564.48, 0];
{true}
3:  B = [1;0];D=1280
{true}
4:  x = zeros(2,1);
{x ∈ EP}
5:  while 1
{x ∈ EP}
6:  y = fscanf(stdin,"%f")
{x ∈ EP}
7:  y = max(min(y,1),-1);
{x ∈ EP, y2 ≤ 1}
8:  u = C*x+D*y;
{x ∈ EP, u2 ≤ 2(CP-1CT + D2), y2 ≤ 1}
9:  fprintf(stdout,"%f\n",u)
{x ∈ EP, y2 ≤ 1, (Ax + By)TP(Ax + By) - 0.01xTPx - 0.99y2 ≤ 0}
skip
{Ax + By ∈ EP, y2 ≤ 1}
10:  x = A*x + B*y;
{x ∈ EP}
11:  end
```

Forward constraint propagation

```

1:  A = [0.4990, -0.0500;
        0.0100, 1.0000];
2:  C = [-564.48, 0];
3:  B = [1;0];D = 1280;
4:  x = zeros(2,1);
5:  while 1
6:    y = fscanf(stdin,"%f");
7:    y = max(min(y,1),-1);
8:    u = C*x + D*y;
9:    fprintf(stdout,"%f\n",u);
10:   x = A*x + B*y;
11: end

```

$$Q = \begin{bmatrix} 0.01P & 0 \\ 0 & 0.99 \end{bmatrix}$$

```

{true}
1:  A = [0.4990,-0.0500;0.0100,1.0000]
{true}
2:  C = [-564.48, 0];
{true}
3:  B = [1;0];D = 1280;
{true}
4:  x = zeros(2,1);
{x ∈ EP}
5:  while 1
{x ∈ EP}
6:  y = fscanf(stdin,"%f");
{x ∈ EP}
7:  y = max(min(y,1),-1);
{x ∈ EP, y2 ≤ 1}
{( x y ) ∈ EQ}
8:  u = C*x + D*y;
{( x y ) ∈ EQ, u2 ≤ [C D] Q-1 [C D]T}
9:  fprintf(stdout,"%f\n",u);
{( x y ) ∈ EQ}
10: x = A*x + B*y;
{x ∈ E( $\frac{1}{0.01}AP^{-1}A^T + \frac{1}{0.99}BB^T$ )-1}
{x ∈ EP} ←
11: end
{false}

```

C code analysis

```

1: int main(int argc, char *argv[])
2: {
3:   double *x[2], *xb[2], y, u;
4:   x[0] = 0;
5:   x[1] = 0;
6:   while(1){
7:     fscanf(stdin,"%f",&y);
8:     if (y >1){
9:       y=1;
10:    }
11:    if (y<-1){
12:      y=-1
13:    }
14:    u = -564.48*x[0]+1280*y;
15:    fprintf(stdout,"%f\n",u);
16:    xb[0]=x[0];
17:    xb[1]=x[1];
18:    x[0] := 0.4990*xb[0]-0.0500*xb[1]+y;
19:    x[1] := 0.01*xb[0]+xb[1];
20:  }
21:}

```

$$\begin{array}{l}
 \left\{ \begin{array}{l} x \\ x \\ x \end{array} \right\} \in \mathcal{E}_P \\
 8: \quad \text{if } (y > 1) \\
 \left\{ \begin{array}{l} y \\ x \\ xb[1] \end{array} \right\} \in \mathcal{E}_P, y > 1 \\
 9: \quad y=1 \\
 \left\{ x \in \mathcal{E}_P, y \leq 1 \right\} \\
 \mathcal{E}_R = \left\{ x \mid x^T R^{-1} x \leq 1 \right\} = \\
 \mathcal{G}_R = \left\{ x \in \mathcal{E}_P, \begin{bmatrix} y \\ x \\ R \end{bmatrix} \geq 0 \right\}
 \end{array}
 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}
 \begin{bmatrix} x[1] \\ x[2] \\ y \end{bmatrix}$$

$$\left\{ \begin{bmatrix} x \\ y \end{bmatrix} \in \mathcal{E}_Q \right\} \quad Q = \begin{bmatrix} 0.01P & 0 \\ 0 & 0.99 \end{bmatrix} \\
 16: \quad xb[0]=x[0] \\
 \left\{ \begin{bmatrix} x \\ y \\ xb[1] \end{bmatrix} \in \mathcal{G}_R \right\} \\
 R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad Q^{-1} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}^T$$

Is C code the limit?

- Of course not!

```
1: int main(int argc, char *argv[])
2: {
3:     double *x[2], *xb[2], y, u;
4:     x[0] = 0;
5:     x[1] = 0;
6:     while(1){
7:         fscanf(stdin,"%f",&y);
8:         if (y >1){
9:             y=1
10:        }
11:        if (y<-1){
12:            y=-1
13:        }
14:        u = -564.48*x[0]+1280*y;
15:        fprintf(stdout,"%f\n",u);
16:        xb[0]=x[0];
17:        xb[1]=x[1];
18:        x[0] := 0.4990*xb[0]-0.0500*xb[1]+y;
19:        x[1] := 0.01*xb[0]+xb[1];
20:    }
21:}
```

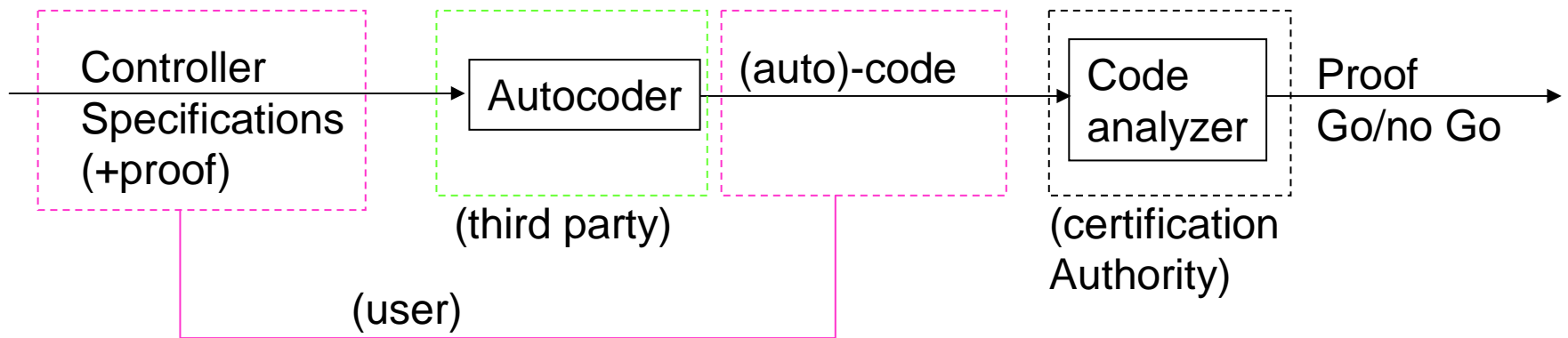
Ideally we want to go all the way down to binary!

Other important questions

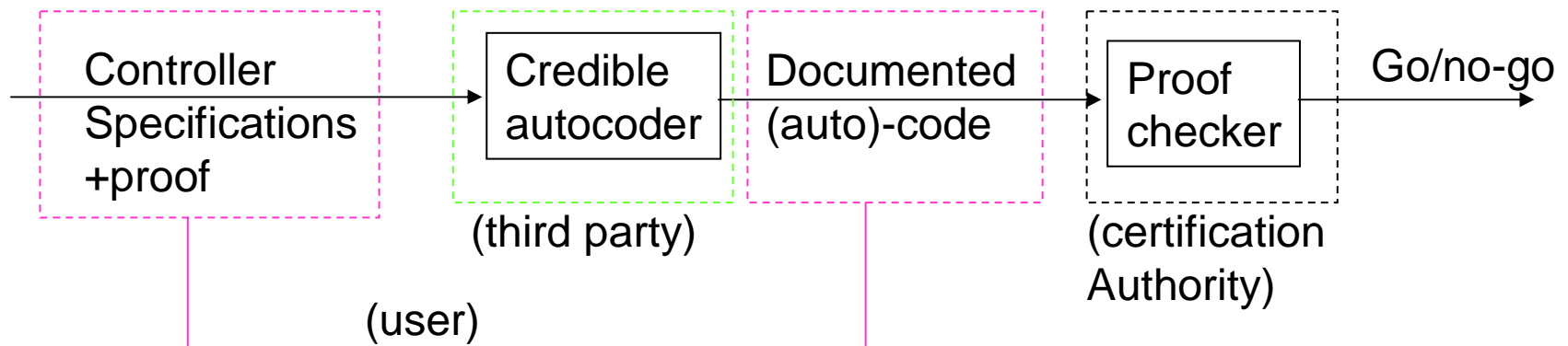
- Verifying closed-loop system
- So far made assumption that analyzer could handle real numbers like you and me.....
- Many controllers are more complex than a simple lead-lag controller – nonlinear? Adaptive? Receding horizon???

Applications

(auto) Code analyzer



Credible autocoder (a la Rinard)



Conclusion

- Stability and performance proofs of control systems fundamentally compatible with formal static analysis and verification methods