

*A Framework for Stability Analysis of Control Systems
Software at the Source Code Level*

Fernando Alegre and Eric Feron
Georgia Institute of Technology

Overview

- Leverage synergy between some Aerospace and Computer Science disciplines to develop *certified* control software
- Mathematical foundations
 - Control theory, dynamical systems, optimization [**Model level**]
 - Formal logic, theorem proving, lattices, semantics [**Code level**]
- Controllers are
 - realizations of particular dynamical systems [**AE view**]
 - embedded systems of some particular restricted types [**CS view**]
- What we want
 - Connect model and implementation
 - Provide formal justification of their equivalence
 - Translate safety properties of models to safety guarantees of corresponding code
- Our focus
 - Use of autocoders to implement models
 - Fully automated verification at source code level
 - Analysis as independent of specific autocoder as possible

Motivation: Autocoders in Industry

- Autocoders are being increasingly used
- A prominent example: The X43 scramjet vehicle program
 - **Goal:**
 - To design and generate flight control software including flight control, propulsion, actuators and sensors
 - Minimize manual coding and debugging
 - **Approach:**
 - Use Simulink to model and validate control systems
 - Real-Time Workshop to automatically generate flight code
 - **Analysis:**
 - Reduced development time by months
 - Accurately predicted separation clearance
 - Aided in achieving SEI CMM (Software Engineering Institute Capability Maturity Model) Level 5 process rating

Problem

Correctness of model does not ensure correctness of code

- Different implementations of the same model are sometimes possible
 - Translation needs some semantic knowledge about the models
 - Autocoders need to make choices (eg. transfer function to state space, integration schemes)
 - Some implementations may be preferable due to better behavior
- Exact numerical equivalence between model and code is impossible
- Typical numerical approximations:
 - discretization of continuous states
 - conversion to finite precision (rounding, truncating)
 - use of unevenly distributed numeric types instead of reals or rationals
 - restriction of computation to a processor-dependent bounded interval
 - interval equality ($|x - y| < \epsilon$) instead of exact equality ($x == y$)

Model vs Code - Example 1

- Consider the following system:

$$x_{n+1} = 1 - a + ax_n$$

$$x_0 = 10, 0 < a < 1$$

- At the model level, this system converges to 1:

$$\forall \epsilon > 0, \exists n_0, |x_n - 1| < \epsilon \text{ whenever } n > n_0$$

- However, this implementation does not converge to 1.0:

```
int n; /*32-bit*/float e=0.1, x=10.0, d=8.046e-
float a=1.-d;
for (n=0; ;n++) {
    if(x-1.<e && 1.-x<e) break;
    x = d + a*x;
}
```

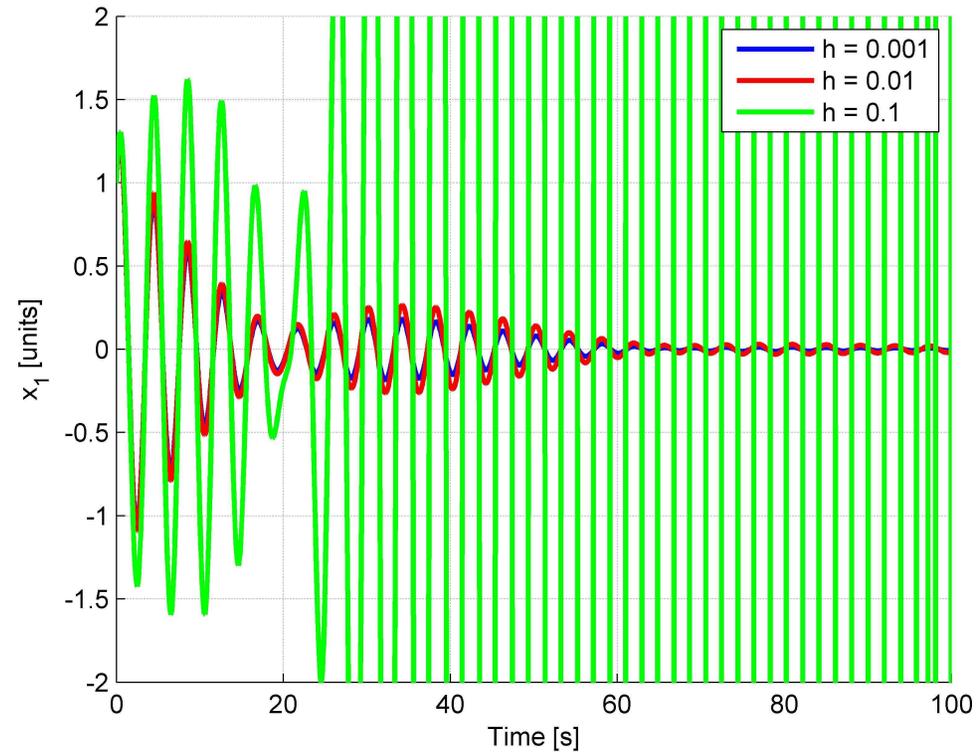
Model vs Code - Example 2

- Consider the system $\dot{x} = Ax$
- The corresponding Euler discretization is $x_{n+1} = (I + hA)x_n$
- Example: Let A be given by:

$$A = \begin{bmatrix} -0.05 & 0.75 & -0.7625 & 0.0125 & 0.025 & -0.775 & -0.7875 & 0.0375 \\ -0.75 & -0.05 & -0.0125 & 0.7625 & 0.775 & -0.025 & -0.0375 & 0.7875 \\ 0.7625 & 0.0125 & -0.05 & -0.775 & 0.7875 & -0.0125 & -0.025 & 0.8 \\ -0.0125 & -0.7625 & 0.775 & -0.05 & 0.0125 & -0.7875 & -0.8 & 0.025 \\ -0.025 & -0.775 & -0.7875 & -0.0125 & -0.05 & 0.8 & -0.8125 & 0.0125 \\ 0.775 & 0.025 & 0.0125 & 0.7875 & -0.8 & -0.05 & -0.0125 & 0.8125 \\ 0.7875 & 0.0375 & 0.025 & 0.8 & 0.8125 & 0.0125 & -0.05 & -0.825 \\ -0.0375 & -0.7875 & -0.8 & -0.025 & -0.0125 & -0.8125 & 0.825 & -0.05 \end{bmatrix}$$

- Behavior of the discretization depends on the choice of parameter h

Model vs Code - Example 2



Model vs Code - Example 3

- Autocoders may have bugs.

Some Real-Time Workshop bugs listed by Mathworks:

- **Incorrect code generation** Non-inlined S-Function block with port-based sample times in multitasking model that uses continuous time (Open)
- **Exported global variable name** Incorrect code if you use tid or controlPortIdx Description (Fixed in Last Version)
- **Incorrect code generation** Real-Time Workshop generates invalid code due to block output optimization settings (Fixed in Last Version)

Version	Open	Fixed from Previous Version
R2007a	18	40
R2006b	50	15
R2006a	51	43

Goal - Large view

- Given
 - A model
 - Some guarantees on its behavior (correctness, bounds, stability)
 - A trustworthy autocoder
- We want to transfer guarantees to source code level
 - Correctness of implementation
 - Termination analysis and stability
 - Proof of functional equivalence between model and implementation

A Simple Model

Consider the simple model

$$\dot{x} = Ax + bu$$

$$y = cx$$

$$x(0) = x_0$$

$$x \in R^{n \times 1}, u, y \in R$$

$$c \in R^{1 \times n}, A \in R^{n \times n}$$

Questions:

- How to translate model into code?
- How to prove properties about model?
- Does the translated code have similar properties?

System Properties -Stability

- Stability

- Asymptotic: $x \rightarrow 0$ as $t \rightarrow \infty$

- Regular: x remains bounded

- BIBO: A bounded input produces a bounded output

- Criteria for stability depends on eigenvalues of matrix A

- Continuous systems: eigenvalues must have negative real parts

- Discrete systems: eigenvalues must lie inside unit circle

System Properties -Stability

- Alternative characterization
- Generalizes to non-linear systems
- Similar to rank function in termination analysis
- A function measures *progress* towards goal
- For linear systems: $V(x) = x'Px$, with P positive definite.
- Existence criteria
 - Continuous case: $A'P + PA + Q = 0$
 - Discrete case: $A'PA + Q = P$

Code Translation - Manual

```
A = [0.999, 0; 0, 1];
c = [1, 0];
b = [2; 2];
x = [1000; 0];
while 1
    u = fscanf(stdin, "%f");
    x = A*x + b*u;
    y = c*x;
    fprintf(stdout, "%f\n", y);
end
```

```
#include <stdio.h>

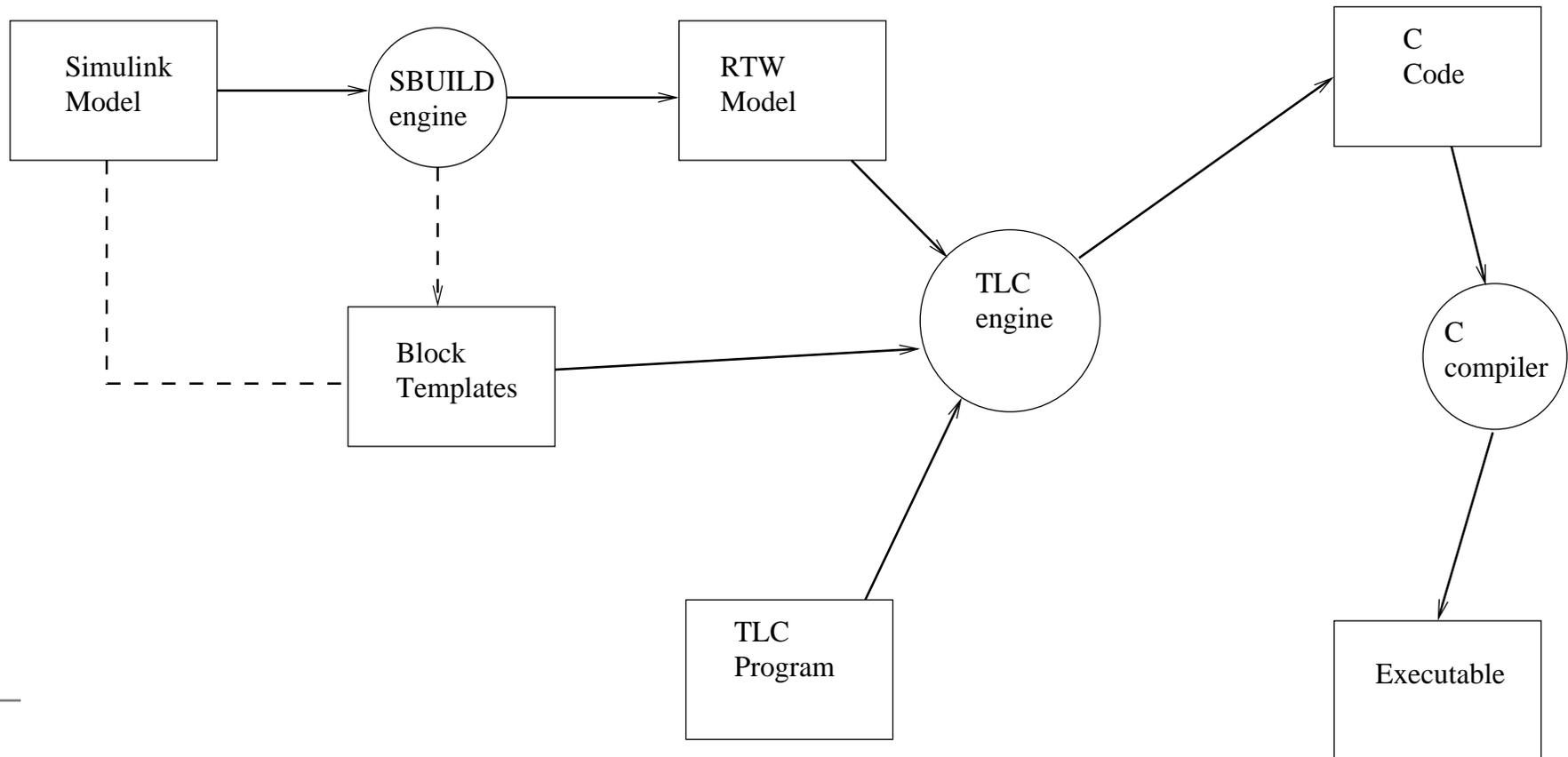
double A[2][2] = { {0.999, 0}, {0, 1} };
double c[2] = { 1, 0 };
double b[2] = { 2, 2 };
double x[2] = { 1000, 0 };
double u,y;

void main(void)
{
    int i,j;
    double x_new[2];

    while(1)
    {
        fscanf(stdin,"%f",&u);
        for(i=0;i<2;i++) {
            x_new[i] = 0;
            for(j=0;j<2;j++) x_new[i] += A[i][j]*x[j];
            x_new[i] += b[i]*u;
        }
        for(i=0;i<2;i++) x[i] = x_new[i];
        y = 0; for(i=0;i<2;i++) y += c[i]*x[i];
        fprintf(stdout,"%f\n",y);
    }
}
```

Code Translation - RTW

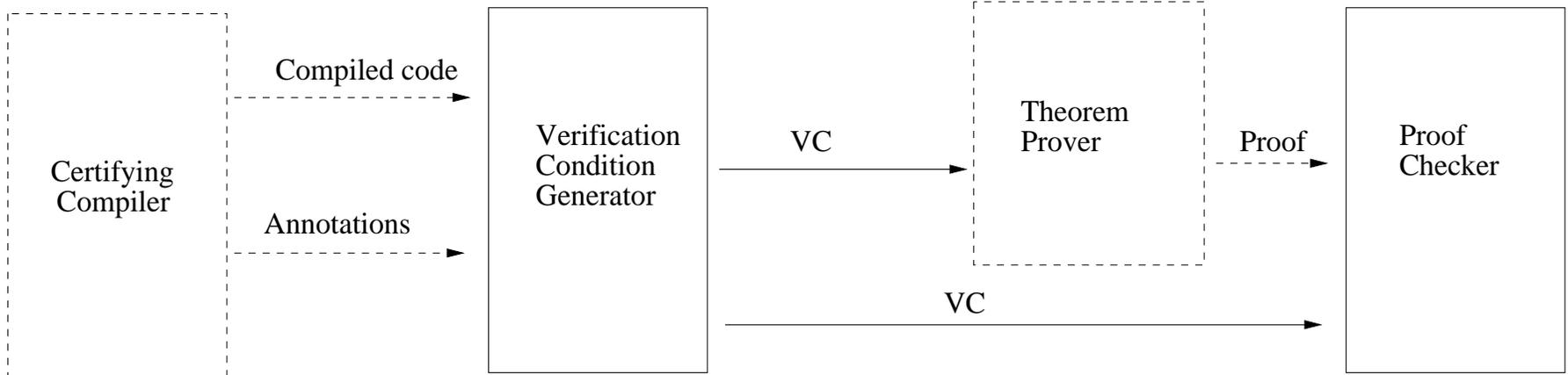
- Complicated output (pointers, casts, bitfields, ...)
- Ad-hoc implicit semantics
- Template based



Alternatives for Framework

- Given a proof at the spec level, how to extend it to the code level?
- Alternatives:
 1. Prove that translation preserves properties.
 - (a) Certified compiler
 - (b) Credible compiler
 - (c) Pattern matching
 2. Create proof at code level
 - (a) Template generator
 - (b) Decompiler
 - (c) Direct proof
 3. Mixed approaches

Proof Carrying Code



————— Trusted
- - - - - Untrusted

- Compiler is not essential, but annotations are
- VCG performs symbolic evaluation
- Prover in original PCC did type-checking only
- Checker used the Edimburg Logical Framework (very complicated)

Parametric Abstract Domains

Parametric abstract domains: Partitions of state space invariant wrt atomic operations in code. General form:

$$\mathcal{D}_f(p) = \{x \in S : f(p, x) \geq 0\}$$
$$\bigcup_p \mathcal{D}_f(p) = S$$

Hoare triples: $\{x \in \mathcal{D}_f(p)\} S \{x \in \mathcal{D}_f(q)\}$

Idea: Parametric abstract domains (PAD) make it possible to avoid exponential explosion when formally verifying numerically bound software, and in particular control systems software.

Ellipsoidal Abstract Domain

- Direct definition:

$$\mathcal{E}(P) = \{x \in \mathbf{R}^n : x^t P x \leq 1\}$$

- Reverse definition:

$$\mathcal{E}^\dagger(P) = \{x \in \mathbf{R}^n : \begin{pmatrix} 1 & x^t \\ x & P \end{pmatrix} \geq 0\}$$

- Both equivalent when P invertible

Ellipsoidal Invariants

```
initialize(x); /* x is the state */  
loop:  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
goto loop;
```

Figure 1: Pseudocode for the main loop of a controller

$$\begin{aligned}x_0 &\in \mathcal{E}(P_0) \\x_{n+1} &= f(x_n, y_n) \\u_{n+1} &= g(x_n, y_n)\end{aligned}$$

Figure 2: High-level specification of a typical controller

Invariant Propagation - 1

```
initialize(x); /* x is the state */  
[[x ∈  $\mathcal{E}(P_0)$ ]]  
loop:  
[[x ∈  $\mathcal{E}(P_0)$ ]]  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
[[x ∈  $\mathcal{E}(P_1)$ ]]  
goto loop;
```

Invariant Propagation - 2

```
initialize(x); /* x is the state */  
[[ $x \in \mathcal{E}(P_0)$ ]]  
loop:  
[[ $x \in \mathcal{E}(P_0) \vee x \in \mathcal{E}(P_1)$ ]]  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
[[ $x \in \mathcal{E}(P_1) \vee x \in \mathcal{E}(P_2)$ ]]  
goto loop;
```

Invariant Propagation - 3

```
initialize(x); /* x is the state */  
[[ $x \in \mathcal{E}(P_0)$ ]]  
loop:  
[[ $\forall_{n \geq 0} x \in \mathcal{E}(P_n)$ ]]  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
[[ $\forall_{n \geq 1} x \in \mathcal{E}(P_n)$ ]]  
goto loop;
```

Invariant Propagation - 4

```
initialize(x); /* x is the state */  
[[ $x \in \mathcal{E}(P)$ ]]  
loop:  
[[ $x \in \mathcal{E}(P)$ ]]  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
[[ $x \in \mathcal{E}(P)$ ]]  
goto loop;
```

Invariant Propagation - 5

```
initialize(x); /* x is the state */  
[[ $x \in \mathcal{E}(P)$ ]]  
loop:  
[[ $x \in \mathcal{E}(P)$ ]]  
[[ $x \in f^{-1}(\mathcal{E}(P))$ ]]  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
[[ $x \in \mathcal{E}(P)$ ]]  
goto loop;
```

Invariant Propagation - 6

```
initialize(x); /* x is the state */  
[[x ∈ E(P)]]  
loop:  
[[x ∈ E(P)]]  
Assume  $P \subset f^{-1}(E(P))$   
[[x ∈ f-1(E(P))]]  
read(y); /* y is the input from the plant */  
compute(u,x,y); /* u is the output to the plant */  
update(x,y);  
write(u);  
[[x ∈ E(P)]]  
goto loop;
```

Note: For linear cases, assumptions are Lyapunov inequality: $\exists P, P \geq A^t P A$.

Intersection Rule

Given quadratic constraints of the form $x \in \mathcal{C}(P)$, we have:

$$\frac{x \in \mathcal{C}(P_1) \quad \dots \quad x \in \mathcal{C}(P_n)}{\forall \vec{\lambda} \in \mathbf{X}^n : x \in \mathcal{C}(\sum_{i=1}^n \lambda_i P_i)}$$

Special case: disjoint variables

$$\frac{\{x_1 \in \mathcal{E}(P_1)\} \quad \dots \quad \{x_n \in \mathcal{E}(P_n)\}}{\forall \lambda \in \mathbf{X}^n, \{x_1 \oplus \dots \oplus x_n \in \mathcal{E} \left(\begin{array}{cccc} \lambda_1 P_1 & 0 & \dots & 0 \\ 0 & \lambda_2 P_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \lambda_n P_n \end{array} \right) \}}$$

Incremental introduction

Lemma 1 *Let $\lambda^1 = (0, 1)$, $\lambda^2 \in \mathbf{X}^2, \dots, \lambda^n \in \mathbf{X}^2$ be a family of 2-dimensional convex combinators and let $x \in \mathbf{R}^n$ be any arbitrary value such that*

$$\sum_{k=1}^n x_k \lambda_2^k \prod_{j=k+1}^n \lambda_1^j \leq 1$$

Then, there exists a convex combinator $\hat{\lambda} \in \mathbf{X}^n$ such that

$$\sum_{k=1}^n x_k \hat{\lambda}_k \leq 1$$

Based on assignment $\hat{\lambda}_k = \lambda_2^k \prod_{j=k+1}^n \lambda_1^j$

Projection Rule

Lemma 2 *Let $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ be two disjoint vectors that satisfy the joint constraint*

$$\{x \oplus y \in \mathcal{E} \begin{pmatrix} P & R^t \\ R & Q \end{pmatrix}\}$$

- 1. if Q invertible, then the vector x satisfies $\{x \in \mathcal{E}(P - R^t Q^{-1} R)\}$.*
- 2. if R is null, then the vector x satisfies $\{x \in \mathcal{E}(P)\}$.*

Assignment Rules

Lemma 3 *An assignment of a vector to a linear transformation of itself given by an invertible matrix A is associated to the Hoare triple*

$$\{x \in \mathcal{E}(P)\} \ x = A*x; \ \{x \in \mathcal{E}(A^{-t}PA^{-1})\} \quad (1)$$

Lemma 4 *Let A be an $n \times m$ matrix. Then, the following triple holds:*

$$\{x \in \mathcal{E}(P)\} \ y = A*x; \ \left\{ \begin{array}{l} \forall \lambda \in \mathbf{R}, \\ x \oplus y \in \mathcal{E} \left(\begin{pmatrix} P & 0 \\ 0 & 0 \end{pmatrix} + \lambda \begin{pmatrix} A^t \\ -I \end{pmatrix} \begin{pmatrix} A & -I \end{pmatrix} \right) \end{array} \right\}$$

If P is invertible, then the following triple also holds:

$$\{x \in \mathcal{E}^\dagger(\hat{P})\} \ y = A*x; \ \left\{ \begin{array}{l} \forall \epsilon \in \mathbf{R}, \\ x \oplus y \in \mathcal{E}^\dagger \left(\begin{pmatrix} \hat{P} & \hat{P}A^t \\ A\hat{P} & A\hat{P}A^t + \epsilon I \end{pmatrix} \right) \end{array} \right\}$$

where $\hat{P} = P^{-1}$.

Example of Rule Application

$$x[i] = x[i] + a[i][j]*x[j];$$

- i and j are integer indices
- x is the state
- a is a constant array, not part of the state

Assignment is

$$x \leftarrow (I + a_{ij}E_{ij})x$$

Therefore

$$\left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathcal{E}^\dagger(\mathbf{M}) \right\} x[i] = x[i] + a[i][j]*x[j]; \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathcal{E}^\dagger(\mathbf{M}^*) \right\}$$

$$\text{where } \mathbf{M}^* = \begin{pmatrix} TPT^t & TR^t \\ RT^t & Q \end{pmatrix} \text{ and } T = I + a_{ij}E_{ij}.$$

Rectangular vs Ellipsoidal Domains

Define rectangle as $\mathcal{R}(l_1, \dots, l_n) = \prod_{i=1}^n [-\frac{l_i}{2}, \frac{l_i}{2}]$

Lemma 5 *Let $\vec{\lambda} \in \mathbf{X}^n$ be a convex combinator, and let $P_{\vec{\lambda}}$ be the matrix $(\delta_{ij} \lambda_i \sqrt{2l_i^{-1}})_{i,j=1}^n$, where δ_{ij} is the Kronecker delta. Then we have*

$$\mathcal{R}(l_1, \dots, l_n) \subset \mathcal{E}(P_{\vec{\lambda}})$$

Furthermore, the right-hand side of (5) is a limit of non-degenerate ellipsoids in the following sense:

$$\mathcal{R}(l_1, \dots, l_n) = \bigcap_{\vec{\lambda} \in (0,1)^n} \mathcal{E}(P_{\vec{\lambda}})$$

Abstract compilation

- Look at Hoare propositions as a program in an abstract domain
- Initial condition is input
 - Assume initial state and control are uniformly bounded
 - Form state comprising all variables in program
 - Boundedness means state lives initially inside a circle or rectangle
- Execute Hoare propositions in abstract domain
- Propagate ellipsoidal constraints forward
- Proving stability basically means proving that output lies inside input
- The proof consists of solving a Lyapunov equation naturally arising from program execution.

Abstract compilation - Example

```
void main(void)
{
  int i,j;
  double x[2] = { 100.0, 200.0 }, x_new[2]
  double u = 10.0;

  while(1)
  {
    for(i=0;i<2;i++) x_new[i] = 0;
    for(i=0;i<2;i++)
      for(j=0;j<2;j++)
        x_new[i] += A[i][j]*x[j];
    for(i=0;i<2;i++) x_new[i] += b[i]*u;
    for(i=0;i<2;i++) x[i] = x_new[i];
    y = 0; for(i=0;i<2;i++) y += c[i]*x[i];
    fprintf(stdout,"%f\n",y);
  }
}
```

```
x = P; % P(2,2) from specs
u = Q; % Q from specs
while 1,
  for i=1:2, aux = II(x_new)-EE(x_new,i); x_new = aux*x_new*aux';
  end
  for i=1:2,
    for j=1:2,
      aux = [II(x_new), A(i,j)*EE(A,i,j); ZZ(x_new), II(x)];
      aux = aux*[x_new;x]*aux';
      x_new = aux(1:size(x_new),:); x = aux(1+size(x_new),:);
    end
  end
  for i=1:2,
    aux = [II(u), ZZ(x_new); b(i)*EE(b,i), II(x_new)];
    aux = aux*[u;x_new]*aux';
    u = aux(1:size(u),:); x_new = aux(1+size(u),:);
  end
  y = 0;
  for i=1:2,
    aux = [II(y),c(i)*EE(c,i);ZZ(y),II(x)]; aux=aux*[y;x]*aux';
    y = aux(1:size(y),:); x = aux(1+size(y),:);
  end
  if(y < R), % R comes from specs
    break
  end
end
```

More complicated rules

Given a function f such that $|f(x)| \leq |x|$, we have

$$\{x \in \mathcal{E}(P)\} \oplus u = f(x); \left\{ \begin{array}{l} \exists \mu_0, \forall \mu \in [0, \mu_0], \\ x \oplus u \in \mathcal{E} \left(\begin{array}{cc} P - \mu I & 0 \\ 0 & \mu I \end{array} \right) \end{array} \right\}$$

- Introduces non-deterministic behavior
- Breaks equivalency between direct and reverse ellipsoids
- Often appears when $x = x + bf(cx)$ is expanded.

More complicated rules

Given a function f such that $|f(x)| \leq |x|$, we have

$$\{x \in \mathcal{E}(P)\}_u = \mathbf{f}(\mathbf{x}) ; \left\{ \begin{array}{l} \exists \mu_0, \forall \mu \in [0, \mu_0], \\ x \oplus u \in \mathcal{E} \left(\begin{array}{cc} P - \mu I & 0 \\ 0 & \mu I \end{array} \right) \end{array} \right\}$$

Transformation rule:

$$\{x \in \mathcal{E}(P)\}$$

$$y = cx$$

$$u = \mathbf{f}(y)$$

$$\mathbf{x} = \mathbf{x} + bu$$

$$\{x \in \mathcal{E}(P) \wedge u^2 \leq y^2 \wedge (y - cx)^2 = 0\}$$

More complicated rules

$$\{x \in \mathcal{E}(P) \wedge u^2 \leq y^2 \wedge (y - cx)^2 = 0\}$$

equivalent to

$$\{\forall \lambda \in \mathbf{R}, \mu \geq 0, xP^t x + \lambda(y - cx)^2 + \mu(u^2 - y^2) \leq 1\}$$

equivalent to

$$\{\forall \lambda \in \mathbf{R}, \mu \geq 0, (x \ y \ u)^t \in \mathcal{E}^\dagger(M_{\lambda\mu})\}$$

$$M_{\lambda\mu} = \begin{pmatrix} S^{-1} & S^{-1}\eta c^t & 0 \\ \eta c S^{-1} & \frac{cP^{-1}c^t}{1+\eta\mu cP^{-1}c^t} & 0 \\ 0 & 0 & \mu^{-1} \end{pmatrix}$$

$$S = P - \eta\mu c^t c$$

$$\eta = \frac{\lambda}{\lambda - \mu}$$

Example

$$\{x \in \mathcal{E}^\dagger(P)\}$$

$$y = c * x;$$

$$\{x \oplus y \in \mathcal{E}^\dagger \left(\begin{array}{cc} P & Pc^t \\ cP & cPc^t + \epsilon_y I \end{array} \right) = \mathcal{E}^\dagger(P_1(\epsilon_y))\}$$

$$u = f(y);$$

$$\{x \oplus y \oplus u \in \mathcal{E}^\dagger \left(\begin{array}{cc} -\epsilon_u P_1(\epsilon_y) [P_1(\epsilon_y) - \epsilon_u I]^{-1} & 0 \\ 0 & \epsilon_u I \end{array} \right) =$$

$$\mathcal{E}^\dagger(P_2(\epsilon_y, \epsilon_u))\}$$

$$x = x + b * u;$$

$$\{x \oplus y \oplus u \in \mathcal{E}^\dagger(AP_2(\epsilon_y, \epsilon_u)A^t), A = \left(\begin{array}{ccc} I & 0 & b \\ 0 & I & 0 \\ 0 & 0 & I \end{array} \right) \}$$

Figure 3: Expansion of assignment $x = x + bf(cx)$

Code-related rules

- Constant folding
- Push arrays to bottom of structures
- Convert pointer arithmetic to index arithmetic
- Symbolic interpretation to reduce number of variables
- Dereference leftover pointers
- Work around casts

```
k = 3;
p = &(a[k]);
q = b;
for(i=0;i<n;i++)
    *p++ = *q++;
s[j].t = *(p-1);
```

```
p_i = 3;
q_i = 0;
==> for(i=0;i<n;i++)
        a[p_i++] = b[q_i++];
s.t[j] = a[p_i-1];
```

Code-related rules

- Constant folding
- Push arrays to bottom of structures
- Convert pointer arithmetic to index arithmetic
- Symbolic interpretation to reduce number of variables
- Dereference leftover pointers
- Work around casts

```
if(t > 0) p = func1;
else p = func2;
(*p)(s);
x = s->f1;
```

==>

```
struct s s_d;
s_d = *s;
if(t > 0)
    s_d = func1(s_d);
else
    s_d = func2(s_d);
x = s_d.f1;
```

Further generalizations

- Identify abstract domain: difficult
- Identify main invariant: difficult from scratch, medium from model
- Separate state from data: easy to medium
- Transform code into Flowchart C: easy to medium (pointers, casts)
- Propagate non-parametric constraints: easy
- Propagate parametric constraints: easy if parameters known
- Check proof: easy to medium (depending on how much assumed known)

Thank You