

Symbolic Robustness Analysis

(Static Analysis for Embedded Control)

Rupak Majumdar

Computer Science Department
UC Los Angeles

Joint Work with Indranil Saha

Background

- A lot of recent progress in program analysis
 - Verify temporal properties of C programs [SLAM, Blast]
 - Test generation [DART, Cute, Splat]
- So far, tools have focused on low level systems programs (drivers, OS components)

Can we apply these techniques to embedded control programs?

Why Embedded Control Programs?

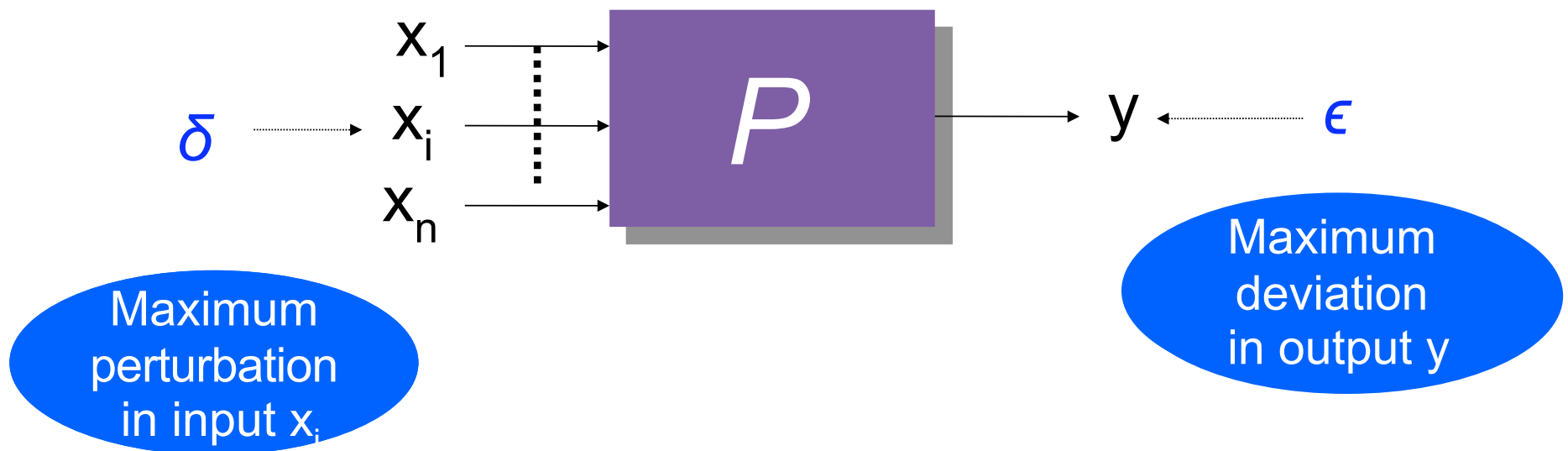
- Safety critical (avionics, automotive, ...)
- Challenging:
 - Physical world and software implementations may not match up
 - Uncertainties in measurements/actuators
- What properties?
 - Language level properties (arithmetic overflows) [Astree]
 - Generic specifications: stability, **robustness**

Robustness

- Slight perturbation in the inputs cause slight changes in the output
- Input perturbations: Measurement uncertainties
- Outputs: Actuators
- Question: Is a software implementation robust?

Robustness

Slight perturbation in the inputs cause slight change in the output



f is (δ, ϵ) -robust in the i -th input

Robustness Analysis

Question: Is a software implementation robust?

Input: Software implementation, tolerance δ

Output: **Test cases** demonstrating (δ, ϵ) -robustness for each input x_i

Robustness Analysis

- **Why is this interesting?**
 - There can be a semantic gap between control theory and the software implementation
 - Focus on **tests** (easier to convince practitioners)
- **Why is this hard?**
 - Input space too large
 - Complex control flow with many correlated paths and table lookups
 - Two “close” inputs can take different code paths based on Boolean tests

Example of Robustness

```
int calc_torque (int angle, int speed)
{
    int val;
    int gear, pressure1, pressure2;
    if (angle <= 45)
        val = 60;
    else
        val = 70;
    if ( 3 * speed <= val)
        gear = 3;
    else
        gear = 4;
    pressure1 = lookup(&(table1[0][0]), gear);
    pressure2 = lookup(&(table2[0][0]), gear);
}
```

table1

gear	pressure
1	0
2	0
3	1000
4	1000

table2

gear	pressure
1	0
2	0
3	0
4	1000

Example of Robustness

```
int calc_torque (int angle, int speed)
{
  int val;
  int gear, pressure1, pressure2;
  if (angle <= 45)
    val = 60;
  else
    val = 70;
  if ( 3 * speed <= val)
    gear = 3;
  else
    gear = 4;
  pressure1 = lookup(&(table1[0][0]), gear);
  pressure2 = lookup(&(table2[0][0]), gear);
}
```

angle = 30,
speed = 20

angle = 30,
speed = 21

val = 60

val = 60

gear = 3

gear = 4

pressure1 = 1000

Pressure1 = 1000

table1

gear	pressure
1	0
2	0
3	1000
4	1000

table2

gear	pressure
1	0
2	0
3	0
4	1000

Robust

Example of Robustness

```
int calc_torque (int angle, int speed)
{
  int val;
  int gear, pressure1, pressure2;
  if (angle <= 45)
    val = 60;
  else
    val = 70;
  if ( 3 * speed <= val)
    gear = 3;
  else
    gear = 4;
  pressure1 = lookup(&(table1[0][0]), gear);
  pressure2 = lookup(&(table2[0][0]), gear);
}
```

angle = 30,
speed = 20

angle = 30,
speed = 21

val = 60

val = 60

gear = 3

gear = 4

pressure2 = 0

pressure2 = 1000

table1

gear	pressure
1	0
2	0
3	1000
4	1000

table2

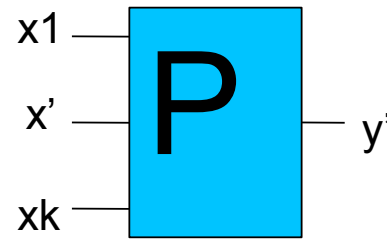
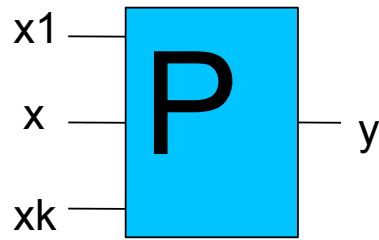
gear	pressure
1	0
2	0
3	0
4	1000

Not
Robust

Problem Definition

- *Given: Program P , input x , maximum uncertainty δ_x in measuring x*
- Find:
 - **maximum difference δ_{yx}** in the output y over all pairs of executions in which x differs by at most δ_x (and all other inputs are the same)
 - a test that exhibits the maximum difference

Problem Definition



Maximize $|y - y'|$

over all pairs $P(x_1, \dots, x, \dots, x_k), P(x_1, \dots, x', \dots, x_k)$

s.t. $|x - x'| \leq \delta_x$

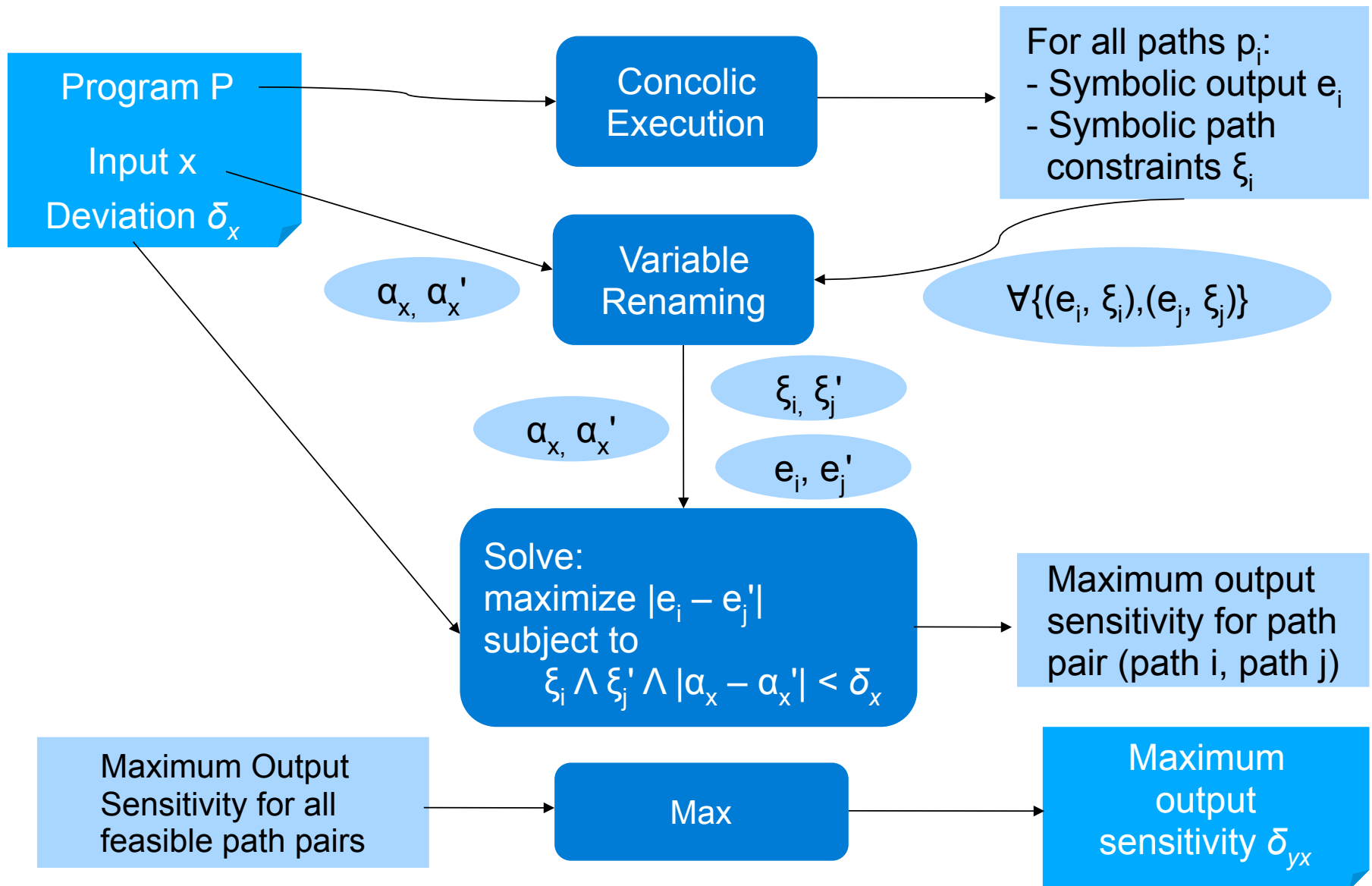
Note: The paths executed can be different

Question: How do we enumerate path pairs?

Symbolic Execution

- Run the program with *symbolic inputs*
- Each execution maintains
 - A **symbolic store**: map program variables to symbolic expressions
 - A **path constraint** that specifies constraints on inputs for the current path to be executed
- A satisfying assignment to the path constraint provides an input that guarantees execution along the path

Algorithm



Example

```
float func(float a, float b)
{
    float x, y;
    x = b + 10;
    if ( x > 0)
        y = x * a;
    else
        y = x * a2;
    return y;
}
```

Symbolic inputs a_0 and b_0

Example

```
float func(float a, float b)
{
    float x, y;
    x = b + 10;
    if ( x > 0)
        y = x * a;
    else
        y = x * a2;
    return y;
}
```

Symbolic inputs a_0 and b_0

Store: x: $b_0 + 10$

Example

```
float func(float a, float b)
{
    float x, y;
    x = b + 10;
    if ( x > 0)
        y = x * a;
    else
        y = x * a2;
    return y;
}
```

Symbolic inputs a_0 and b_0

Store: $x: b_0 + 10$

Constraint: $b_0 + 10 > 0$

Example

```
float func(float a, float b)
{
    float x, y;
    x = b + 10;
    if ( x > 0)
        y = x * a;
    else
        y = x * a2;
    return y;
}
```

Symbolic inputs a_0 and b_0

Store: $x: b_0 + 10$
 $y: (b_0 + 10) * a_0$

Constraint: $b_0 + 10 > 0$

Example

```
float func(float a, float b)
{
    float x, y;
    x = b + 10;
    if ( x > 0)
        y = x * a;
    else
        y = x * a2;
    return y;
}
```

Symbolic inputs a_0 and b_0

Store: $x: b_0 + 10$
 $y: (b_0 + 10) * a_0$

Constraint: $b_0 + 10 > 0$

Symbolic output: $(b_0 + 10) * a_0$

Overall Algorithm

- Perform symbolic execution
 - In the implementation: concolic execution
[GodefroidKlarlundSen,SenMarinovAgha,XuMGodefroid]
- For each pair of path constraints, set up an optimization problem to find the sensitivity of the output to input perturbations

Example

```
float func(float a, float b)
{
    float x, y;
    x = b + 10;
    if ( x > 0)
        y = x * a;
    else
        y = x * a2;
    return y;
}
```

Symbolic inputs a_0 and b_0

Path 1:

Symbolic Output:

$$y = (b_0 + 10) * a_0$$

Constraint: $b_0 + 10 > 0$

Path 2:

Symbolic Output:

$$y = (b_0 + 10) * a_0^2$$

Constraint: $b_0 + 10 \leq 0$

3 path pairs need to be considered for each input

Example: Optimization Problem

For input b and path pair (path 1, path 2)

The optimization problem is

$$\max |(b_1 + 10) * a_1 - (b_2 + 10) * a_2^2|$$

Subject to

$$b_1 + 10 > 0$$

$$b_2 + 10 \leq 0$$

$$a_1 = a_2$$

$$|b_1 - b_2| \leq \delta_b$$

Implementation

- **Symbolic execution engine: Splat**
- **Optimization: Lindo**
 - Optimization problems are non-linear

Case Studies

Two case studies on C programs from Ford's
“Smart Vehicle Baseline Report”

1. Fuel-air ratio control
2. Transmission shift control

Fuel-Air Ratio Control

- Calculates desired fuel based on two inputs: *speed* and *absolute pressure*
- We analyze *fixed point code* generated by *TargetLink*
- 3 paths found by concolic execution
- One path-pair found for which the output is sensitive to both the inputs

Transmission-Shift Control

- Controls for a 4-speed automatic transmission system
- Two inputs: *throttle angle* and *vehicle speed*
- Five outputs: *clutch pressures*
- Analyze *floating point code* generated by TargetLink
- Robustness of each output is analyzed separately for different gear conditions

Transmission-Shift Control (Cont.)

Robustness Analysis For Input *throttle angle* and Output *Clutch Pressure 1*

Current Gear Condition	# Paths	# Path Pairs	# Infeasible Path Pairs	# Insensitive Path Pairs	# Sensitive Path Pairs
1	16	136	105	31	0
2	24	300	249	51	0
3	24	300	247	45	8
4	16	136	102	30	4
1-2	64	2080	1878	202	0
2-3	64	2080	1842	208	30
3-4	64	2080	1851	190	39
4-3	64	2080	1866	181	33
3-2	64	2080	1852	153	35
2-1	64	2080	1704	376	0

Transmission-Shift Control (Cont.)

Robustness Analysis For Input *throttle angle* and Output *Clutch Pressure 2*

Current Gear Condition	# Paths	# Path Pairs	# Infeasible Path Pairs	# Insensitive Path Pairs	# Sensitive Path Pairs
1	16	136	105	31	0
2	24	300	249	45	6
3	24	300	247	45	8
4	16	136	102	30	4
1-2	64	2080	1878	184	18
2-3	64	2080	1842	193	45
3-4	64	2080	1851	190	39
4-3	64	2080	1836	181	33
3-2	64	2080	1852	183	45
2-1	64	2080	1704	363	13

Some Generalizations

- Method extends to other metrics on inputs/ outputs (e.g., streaming programs)
- δ_{yx} is calculated from constant δ_x
 - Ideally δ_{yx} should be calculated as a symbolic function of δ_x
 - Possible using parametric optimization
- Can check for stronger conditions
 - Is the function a **contraction map**?

Limitations

- Loops:
 - In our case study, the loops were unrolled
 - But in general, need some way to summarize information across loops
 - For example, consider a Lipschitz function with constant 2. If it is put in a loop, then two close inputs eventually move far apart
- Non-linear constraint solvers are not very mature

Limitations (Cont.)

- We do not use any control theory information
- Treat the control program as an arbitrary piece of code
- Question: How can we use the control-theoretic insights when analyzing code?

Future Directions

Overall goal of the research is to carry the **mathematical arguments** for correctness for control systems down to **software implementations**

Combine control-theoretic arguments with program analysis

- **Properties**: Stability/performance properties of the controllers
- Push insights from the control design level (Lyapunov functions, reachable sets) to the code level

What we Need

Challenge Problems from Industry!!

- Software verification tools gained a lot of mileage verifying open source software (Linux device drivers, security critical applications, ...)
- No “open source” real-time control software

Thank You