

# Symbolic Robustness Analysis

Rupak Majumdar

Indranil Saha

Department of Computer Science  
University of California, Los Angeles

Real-Time Systems Symposium 2009

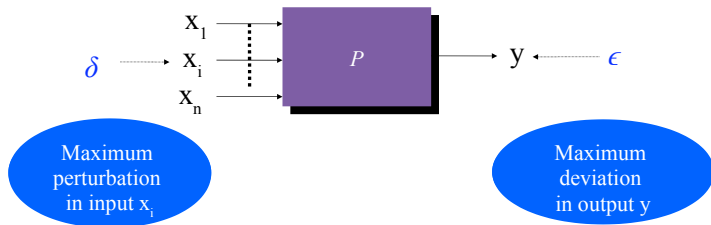
December 4, 2009

- Embedded control programs are used in safety critical systems (avionics, automotive, . . .)
- Uncertainties in measurement of inputs
- **Robustness** ensures correct operation in the presence of uncertainties

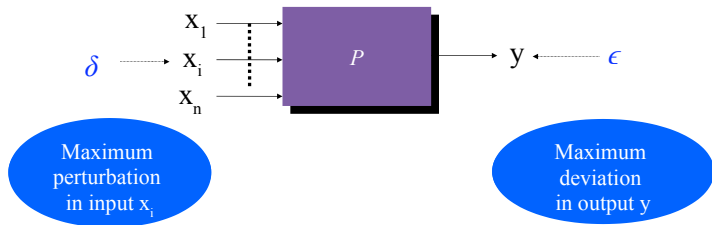
**Question:** Is a software implementation robust?

Small perturbations in the system inputs cause only small changes in its outputs

Small perturbations in the system inputs cause only small changes in its outputs

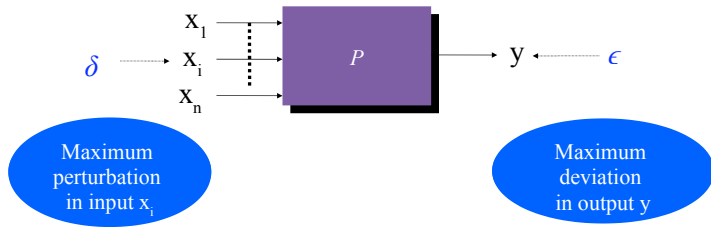


Small perturbations in the system inputs cause only small changes in its outputs



$P$  is  $(\delta, \epsilon)$ -robust in the  $i$ -th input.

Small perturbations in the system inputs cause only small changes in its outputs



$P$  is  $(\delta, \epsilon)$ -robust in the  $i$ -th input.

$\epsilon$  is  $\delta$ -sensitivity of the output w.r.t.  $i$ -th input

**Question:** Is a software implementation robust?

- **Input:** Software implementation, tolerance  $\delta$
- **Output:** **Test cases** demonstrating  $(\delta, \epsilon)$ -robustness for each input  $x_i$

## Why is this interesting?

- There can be a semantic gap between control theory and the software implementation
- Focus on **tests**: want to have regressions for when the code changes

## 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

## Why is this easy?

- Loop bounds are statically known, loops can be unrolled

# Robustness through Example

```
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);
}
```

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

Due to perturbation in input *speed*, the output *pressure1* does not deviate

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

# Robustness through Example

```
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

gear	pressure1
1	0
2	0
3	1000
4	1000

gear	pressure2
1	0
2	0
3	0
4	1000

Due to perturbation in input *speed*, the output *pressure2* may deviate from 0 to 1000

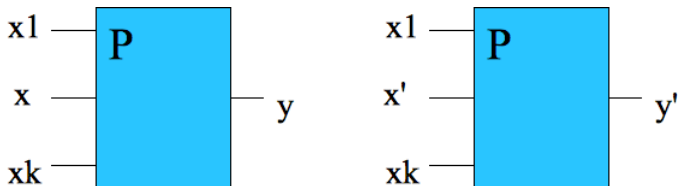
# Problem Definition

Given: Program  $P$ , input  $x$ , maximum uncertainty  $\delta_x$  in measuring  $x$ .

Find:

- **maximum difference**  $\delta_{yx}$  in the output  $y$  over all pairs of executions in which  $x$  differs by at most  $\delta_x$  (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 may be the same or different.

**Question:** How do we enumerate path pairs?

- Run the program with symbolic inputs
- Each execution maintains
  - A **symbolic store**: maps program variables to symbolic expressions
  - A **path constraint**: 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

# Symbolic Execution through 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:  $a : a_0, b : b_0$

# Symbolic Execution through 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:  $a : a_0, b : b_0$   
 $x : b_0 + 10$

# Symbolic Execution through 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:  $a : a_0, b : b_0$

$x : b_0 + 10$

Constraint:  $b_0 + 10 > 0$

# Symbolic Execution through 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:  $a : a_0, b : b_0$

$x : b_0 + 10$

$y : (b_0 + 10) * a_0$

Constraint:  $b_0 + 10 > 0$

# Symbolic Execution through 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:  $a : a_0, b : b_0$

$x : b_0 + 10$

$y : (b_0 + 10) * a_0$

Constraint:  $b_0 + 10 > 0$

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

Symbolic testing in practice is performed through concolic execution [GodefroidKlarlundSen,SenMarinovAgha,XuMajumdarGodefroid] where,

- testing start with a random concrete input
- input for each consecutive execution is generated by appropriately modifying and solving the constraints generated during symbolic execution

# Concolic Execution through 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;  
}
```

Concrete inputs  $a = 0, b = 0$

Symbolic inputs  $a_0$  and  $b_0$

Constraint:  $b_0 + 10 > 0$

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

# Concolic Execution through 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;  
}
```

Concrete inputs  $a = 0, b = 0$

Symbolic inputs  $a_0$  and  $b_0$

Constraint:  $b_0 + 10 > 0$

Negate the constraint:

$b_0 + 10 \leq 0$

# Concolic Execution through 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;  
}
```

Concrete inputs  $a = 0, b = 0$

Symbolic inputs  $a_0$  and  $b_0$

Negate the constraint:

$$b_0 + 10 \leq 0$$

Solve the constraints:

$$a_0 = 0, b_0 = -20$$

# Concolic Execution through 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;  
}
```

Concrete inputs  $a = 0, b = -20$

Symbolic inputs  $a_0$  and  $b_0$

Constraint:  $b_0 + 10 \leq 0$

Symbolic output:  $(b_0 + 10) * a_0^2$

# Overall Algorithm

- Perform concolic execution and collect the symbolic output and path constraint for all the paths
- For each pair of paths, set up an optimization problem to find the sensitivity of the output to input perturbation

# 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:

$$y_0 = (b_0 + 10) * a_0$$

Constraint:  $b_0 + 10 > 0$

## Path 2:

Symbolic Output:

$$y_0 = (b_0 + 10) * a_0^2$$

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

$$\text{maximize } |(b_1 + 10) * a_1 - (b_2 + 10) * a_2^2|$$

Subject to

$$b_1 + 10 > 0$$

$$b_2 + 10 \leq 0$$

$$a_1 = a_2$$

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

Note: Optimization problems are nonlinear

- Symbolic execution engine: **Splat** [XuMajumdar]
  - Splat uses **Stp** [GaneshDill] as the back-end solver
- Optimization: **Lindo** [Lindo Systems, Inc.]
  - Optimization problems are non-linear

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

- The system controls a 4-speed automatic transmission system
- The system has two inputs: **throttle angle** and **vehicle speed** and five output **clutch pressures**
- Outputs are either 0 or 1000
  - $(\_, 0)$ -robust means that the program is **robust**
  - $(\_, 1000)$ -robust means that the program is **not robust**

# Transmission-Shift Control

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	1836	181	33
3-2	64	2080	1852	153	35
2-3	64	2080	1704	376	0

Result For Input **throttle angle** and Output **clutch pressure 1**

# Transmission-Shift Control

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
2-3	64	2080	1704	363	13

Result For Input **throttle angle** and Output **clutch pressure 2**

- Tool currently supports only integer programs
  - Evaluating **HySat** [Fränzle et al.] for floating-point programs
- 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?

# Conclusion and Extension

- A preliminary step towards analyzing software programs for robustness
- Method extends to other metrics on inputs/outputs
- $\delta_{yx}$  is calculated from constant  $\delta_x$ 
  - Ideally  $\delta_{yx}$  should be calculated as a symbolic function of  $\delta_x$
  - Possible using parametric optimization