Probabilistic Analysis of Programs with Numerical Uncertainties

Debasmita Lohar

under the supervision of

Eva Darulova

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

UNIVERSITÄT
DES
SAARLANDES

PhD-iFM 2019

Programming with Numerical Uncertainties

```
def func(x:Real, y:Real, z:Real): Real = {
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
}
```

- **Reals** are implemented in **Floating point**/**Fixed point** data type

Programming with Numerical Uncertainties

```
      (x:Float32, y:Float32, z:Float32): Float32
def func(x:Real, y:Real, z:Real): Real = {
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
}
```

- **Reals** are implemented in **Floating point**/**Fixed point** data type
- Introduces **Round-off error** in the computation

Why should we care about Round-off Errors?

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()    real valued program
    else
      doNothing()    finite precision program
    return res
}
```

- Reals are implemented in Floating point/ Fixed point data type
- Introduces Round-off error in the computation
- Program can take a **wrong decision**

State-of-the-art: Worst Case Error Analysis

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
}
```

↓

Daisy FLUCTUAT

Gappa rosa FPTaylor

....

Worst Case Analysis for Discrete Decisions

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()      real valued program
    else
      doNothing()       finite precision program
    return res
}
```

A program **always** takes the wrong path in the **worst case**

Worst Case Analysis for Discrete Decisions

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()      real valued program
    else
      doNothing()       finite precision program
    return res
}
```

A program **always** takes the wrong path in the **worst case**

Need to consider the **probability distributions** of **inputs**

Worst Case Analysis for Discrete Decisions

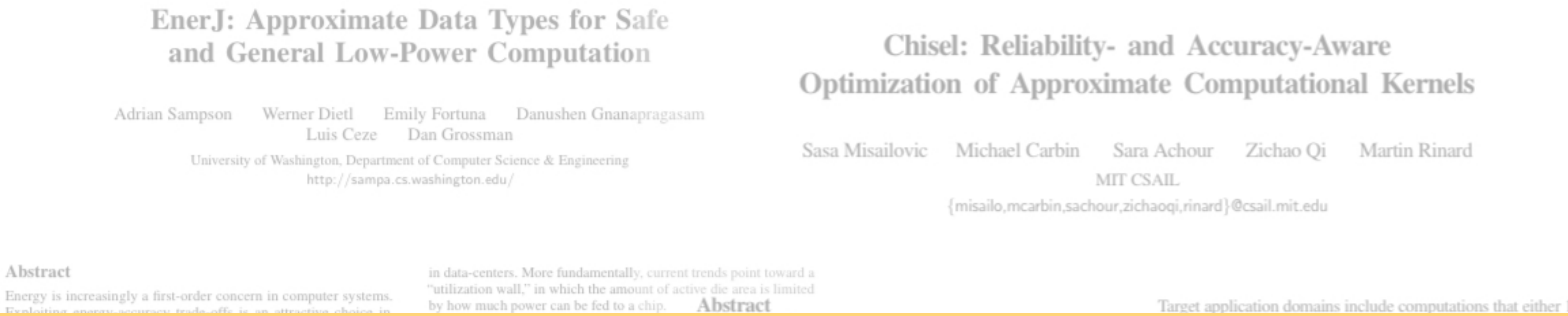
```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()   real valued program
    else
      doNothing()   finite precision program
    return res
}
```

A program **always** takes the wrong path in the **worst case**

Need to consider the **probability distributions** of **inputs**

What happens if we have **Approximate Hardware?**

Approximate Hardware



Resource Efficient but has Probabilistic Error behaviors

Approximate Hardware Specification

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
}
```

Error Specification: `<0.00199, 0.9>, <0.00499, 0.1>`

- Has **Probabilistic Error Specification**

Error Resilient Applications

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
} ensuring (res +/- 0.00199, 0.85)
```

Error Specification: <0.00199, 0.9>, <0.00499, 0.1>

Application tolerates big errors occurring with **0.15** probability

- Has **Probabilistic Error Specification**
- Applications may tolerate large infrequent errors

Worst Case Analysis for Error Resilient Application

```
       (x:Float64, y:Float64, z:Float64): Float64
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
} ensuring (res +/- 0.00199, 0.85)
```

Worst Case Error Analysis

error: **0.002**

Worst Case Analysis = Low Resource Utilization

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    return res
} ensuring (res +/- 0.00199, 0.85)
```

Worst Case Error Analysis

error: **0.002**

Occurs only with probability 0.002 !

Worst Case Analysis = Low Resource Utilization

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   return res
} ensuring (res +/- 0.00199, 0.85)
```

↓

Worst Case Error Analysis

↓

error: **0.002**

Occurs only with probability 0.002 !

Need to consider the **probability distributions** of **inputs**

Two Problems

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
    require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()
    else
      doNothing()
    return res
}
```

real valued program

finite precision program

How often does a program take a wrong decision?

How do we compute a precise bound on the error

by taking into account the probability distribution of inputs

https://github.com/malyzajko/daisy/tree/probabilistic

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

ÉCOLE POLYTECHNIQUE UNIVERSITÉ PARIS-SACLAY

How often does a program take a wrong decision?

"Discrete Choice in the Presence of Numerical Uncertainties", EMSOFT'18

Eva Darulova

Sylvie Putot

Eric Goubault

Our Goal

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
   require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   if (res <= 0.0)
     raiseAlarm()
   else
     doNothing()
   return res
}
```

} How often?

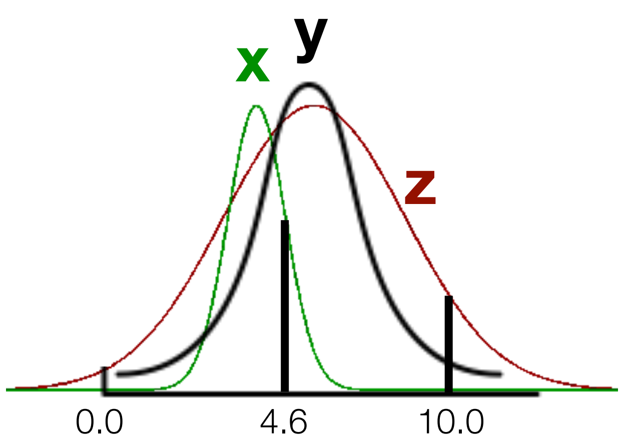
Compute **W**rong **P**ath **P**robability

Input Distributions are important!

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)

    x:= gaussian(0.0, 4.6)
    y:= gaussian(0.0, 10.0)
    z:= gaussian(0.0, 10.0)

    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()
    else
      doNothing()
    return res
}
```



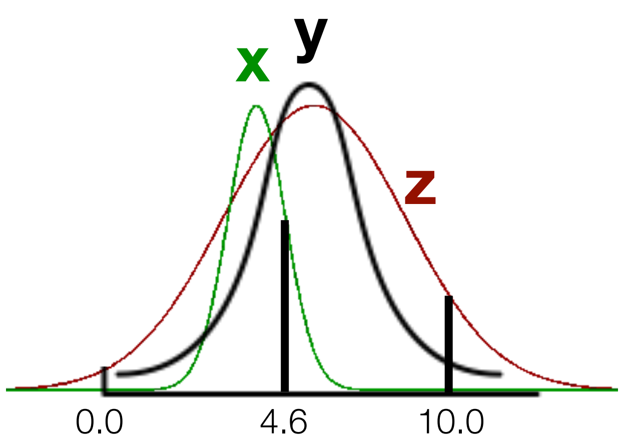
Our Goal: Probabilistic Analysis

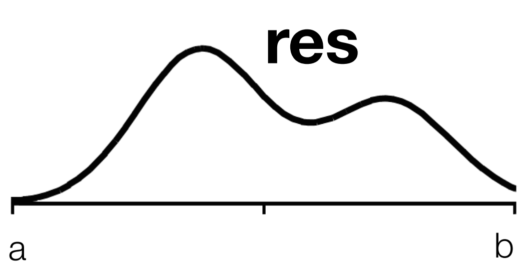
```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)

    x:= gaussian(0.0, 4.6)
    y:= gaussian(0.0, 10.0)
    z:= gaussian(0.0, 10.0)

    val res = -3.79*x - 5.44*y + 9.73*z + 4.52

    if (res <= 0.0)
      raiseAlarm()
    else
      doNothing()
    return res
}
```





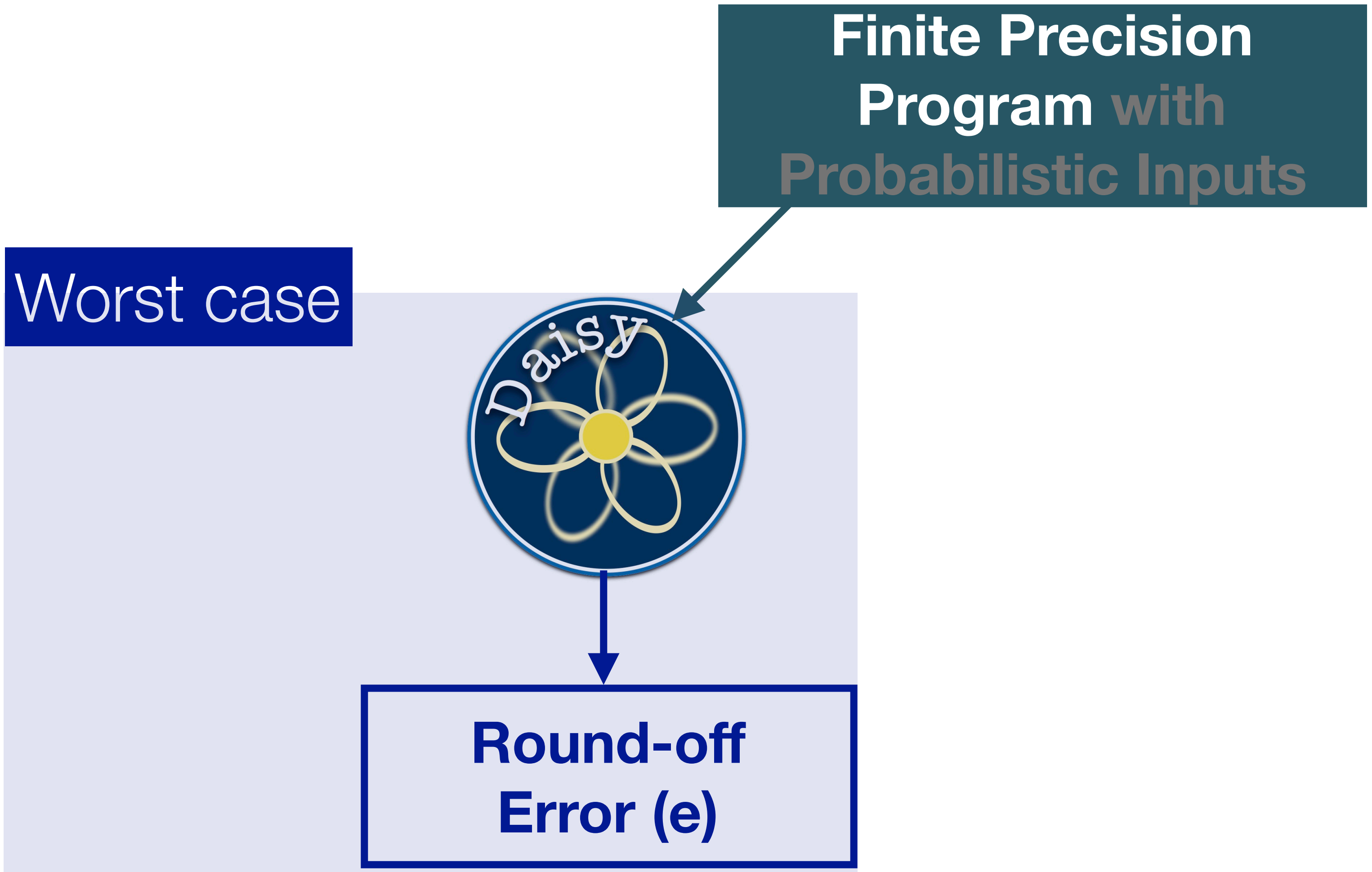
} Compute **W**rong **P**ath **P**robability

Overview: Sound Analysis

Finite Precision Program with Probabilistic Inputs

Wrong Path Probability (WPP)

Round-off Error Analysis



"Daisy - Framework for Analysis and Optimization of Numerical Programs", E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, TACAS 2018

Overview: Sound Analysis

Finite Precision Program with Probabilistic Inputs

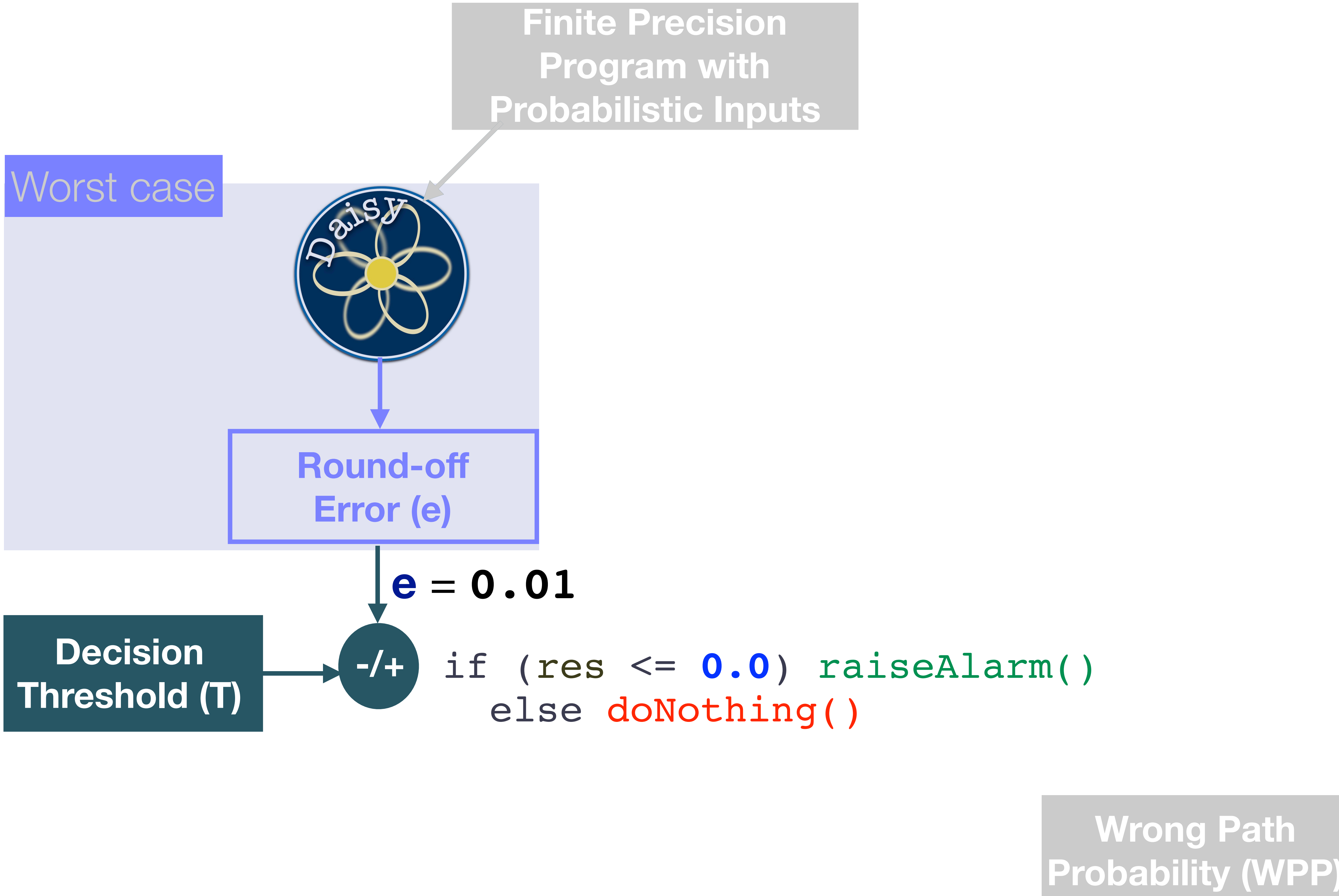
Worst case

Round-off Error (e)

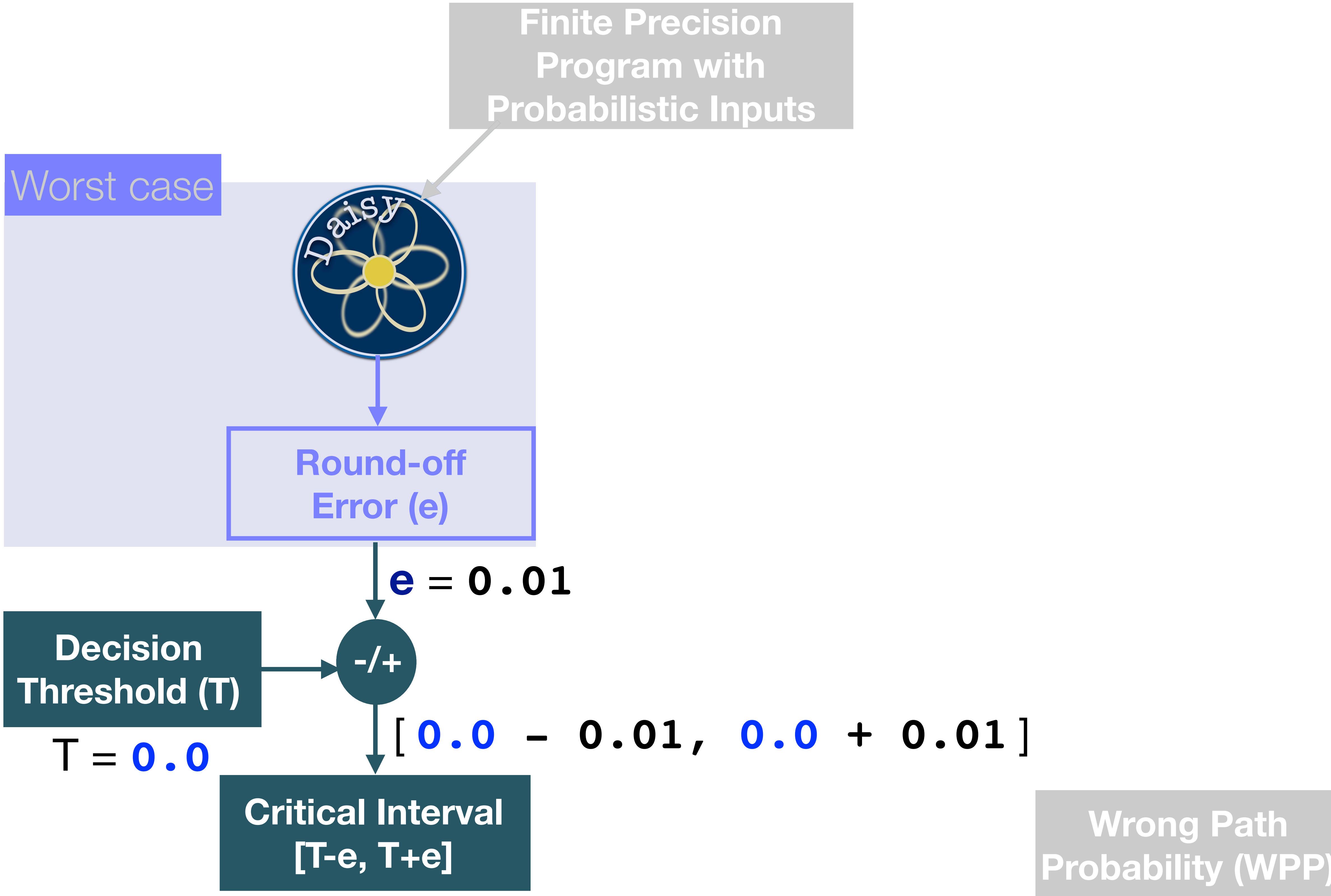
e = 0.01

Wrong Path Probability (WPP)

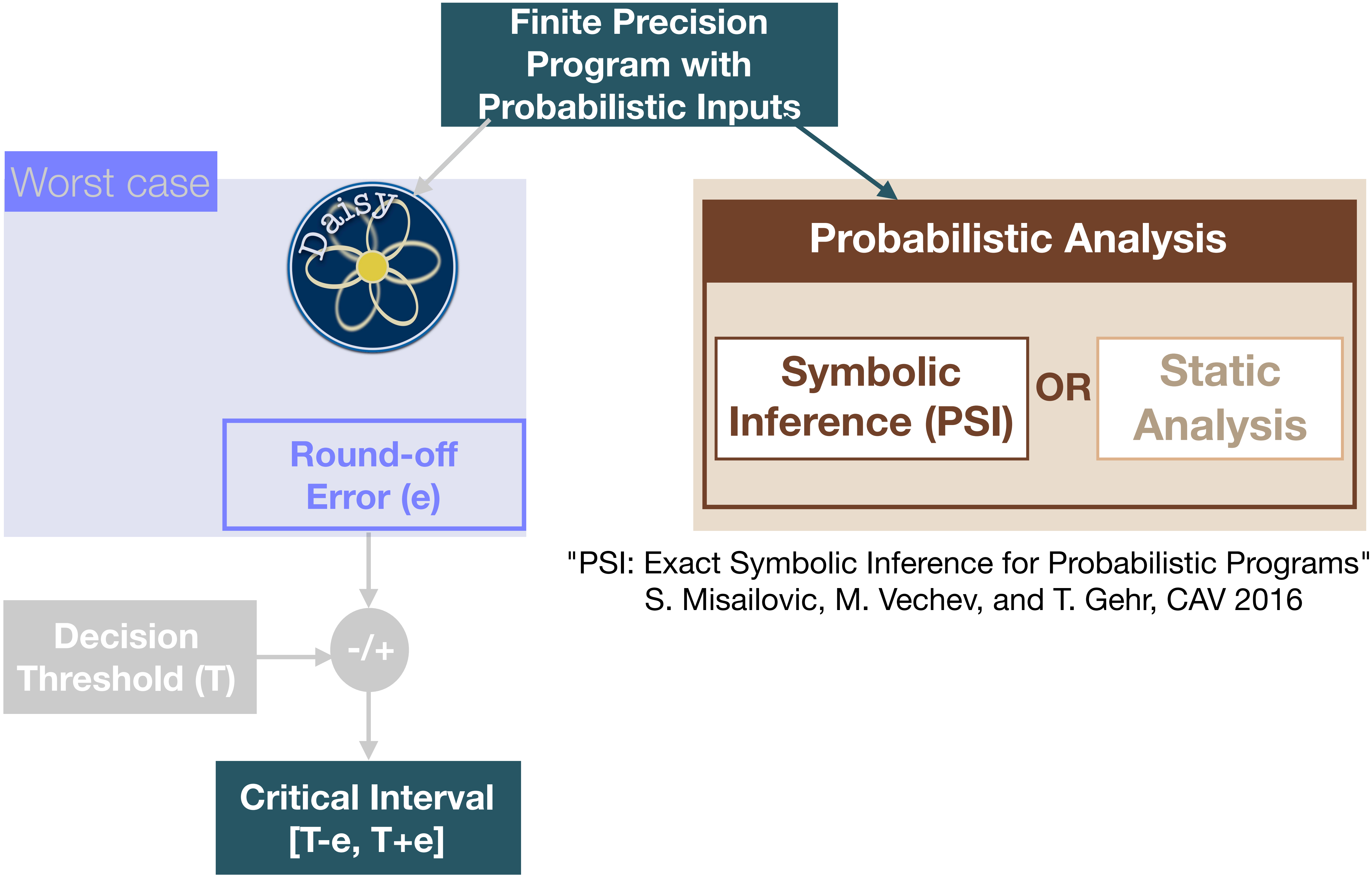
Overview: Sound Analysis



Overview: Sound Analysis

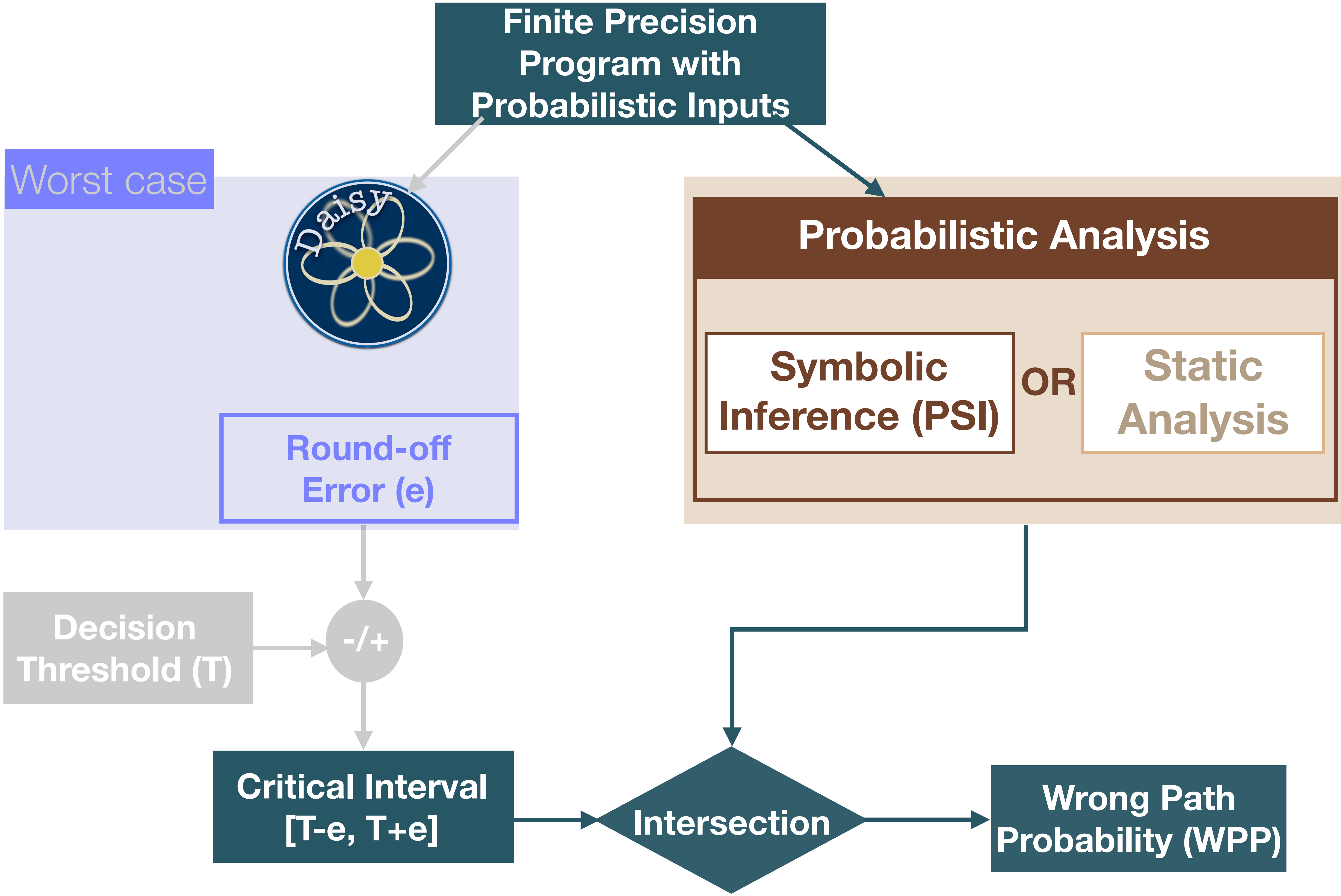


Distribution Propagation

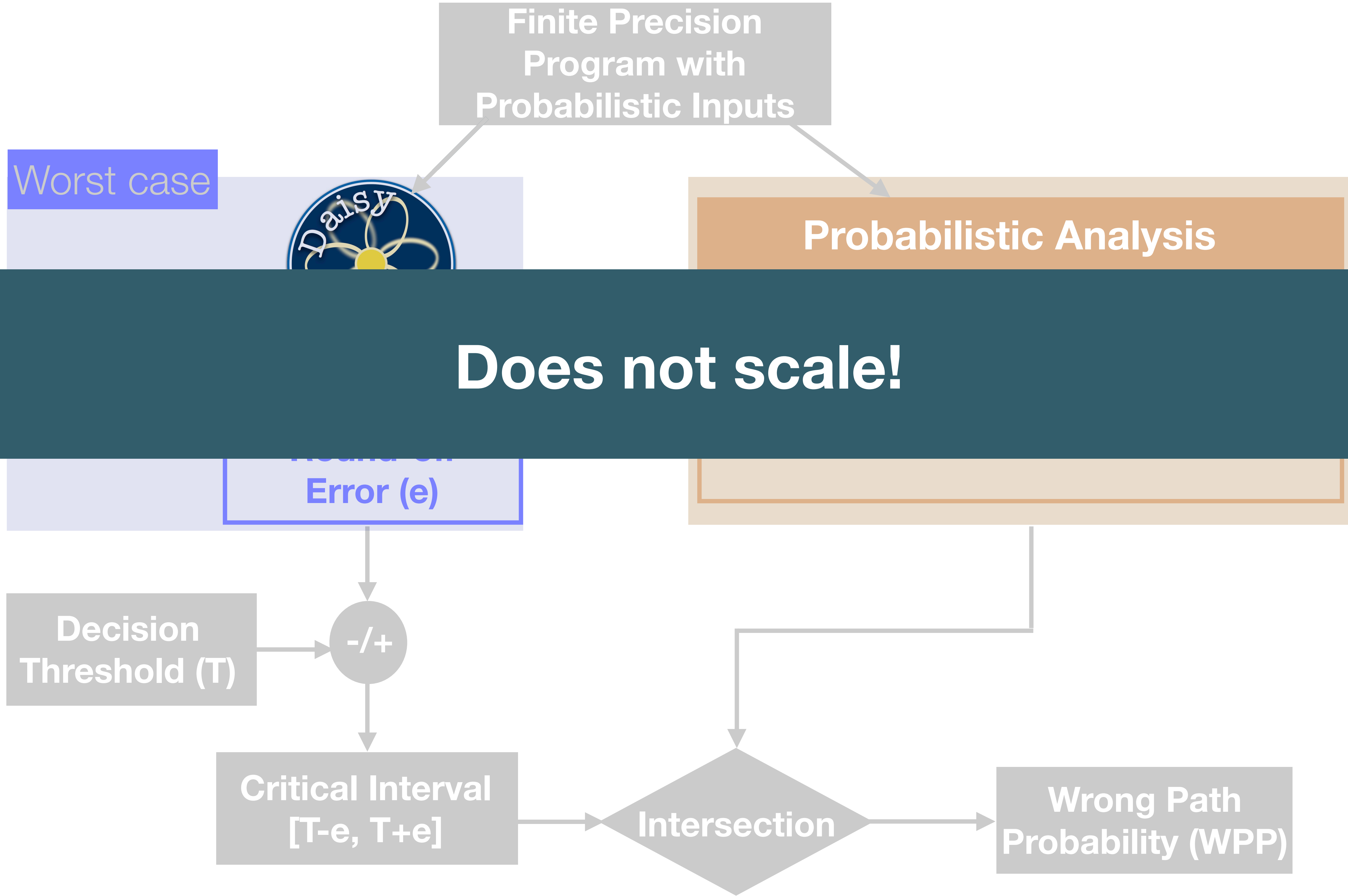


"PSI: Exact Symbolic Inference for Probabilistic Programs", S. Misailovic, M. Vechev, and T. Gehr, CAV 2016

Distribution Propagation

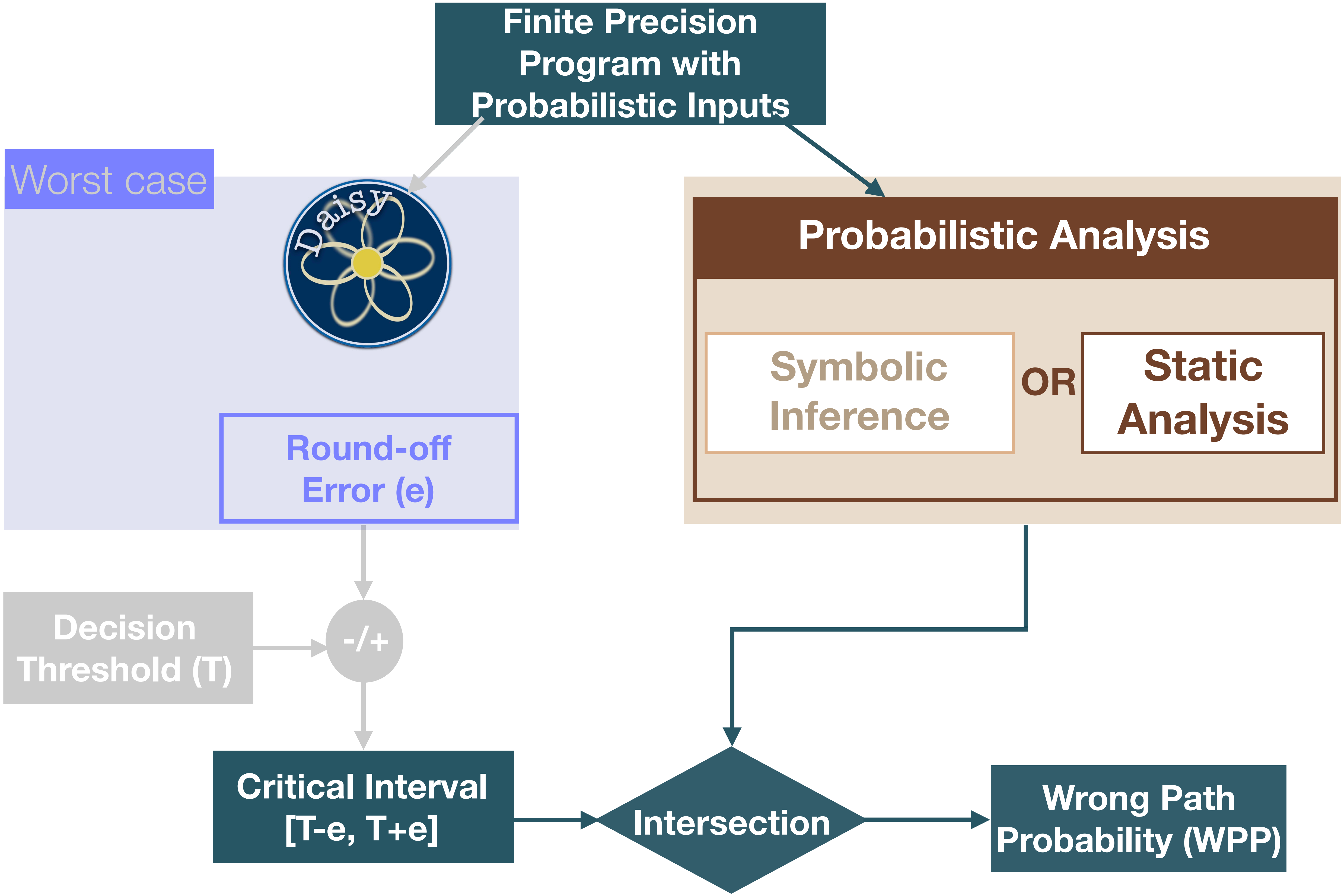


Distribution Propagation

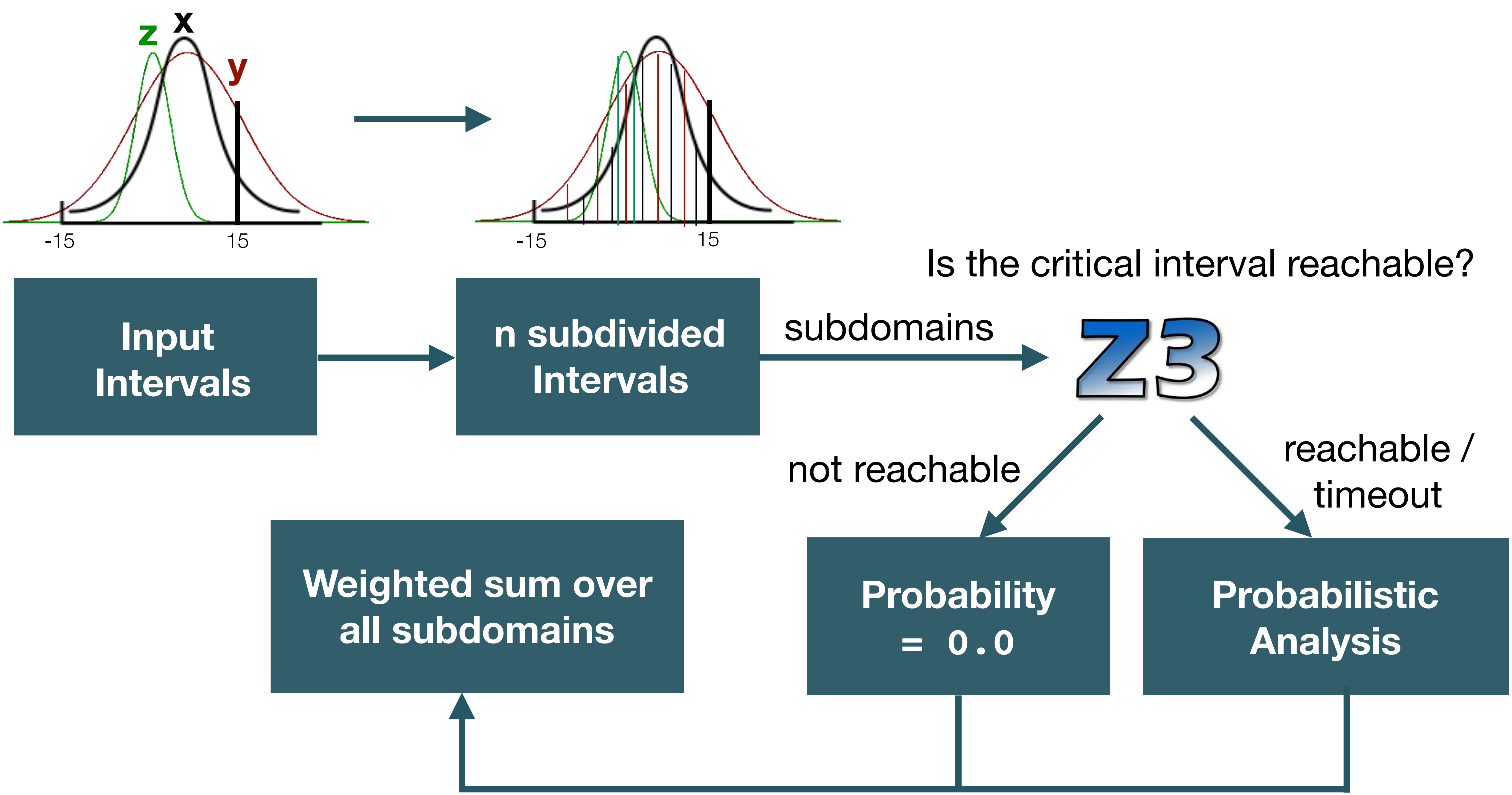


Does not scale!

Distribution Propagation



Using **Probabilistic Static Analysis**



Results: Wrong Path Probability

Benchmarks	#ops	WPP using Sym. Inf.	WPP using Probabilistic with Subdiv
sine	18		
sqrt	14		
turbine1	14		
traincar2	13		
doppler	10		
bspline1	8		
rigidbody1	7		
traincar1	7		
bspline0	6		
sineorder3	4		

Wrong Path Probability for 32 bit floating-point round-off errors and uniform input distributions

Results: Wrong Path Probability

Benchmarks	#ops	WPP using Sym. Inf.	WPP using Probabilistic with Subdiv
sine	18	7.61E-07	
sqrt	14	8.74E-06	
turbine1	14	TO	
traincar2	13	TO	
doppler	10	TO	
bspline1	8	2.54E-06	
rigidbody1	7	TO	
traincar1	7	TO	
bspline0	6	1.05E-05	
sineorder3	4	1.90E-06	

Wrong Path Probability for 32 bit floating-point round-off errors and uniform input distributions

Results: Wrong Path Probability

Benchmarks	#ops	WPP using Sym. Inf.	WPP using Probabilistic with Subdiv
sine	18	7.61E-07	6.45E-05
sqrt	14	8.74E-06	**9.38E-05**
turbine1	14	TO	**4.82E-02**
traincar2	13	TO	9.17E-02
doppler	10	TO	**2.17E-02**
bspline1	8	2.54E-06	1.95E-05
rigidbody1	7	TO	**7.06E-02**
traincar1	7	TO	1.86E-02
bspline0	6	1.05E-05	**6.06E-05**
sineorder3	4	1.90E-06	1.23E-04

Wrong Path Probability for 32 bit floating-point round-off errors and uniform input distributions

Two Problems

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
   require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   if (res <= 0.0)
     raiseAlarm()
   else
     doNothing()
   return res
}
```

How often does a program take a wrong decision?

How do we compute a precise bound on the error?

by taking into account the probability distribution of inputs

https://github.com/malyzajko/daisy/tree/probabilistic

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

How do we compute a precise bound on the error?

"Sound Probabilistic Numerical Error Analysis", iFM'19

Milos Prokop

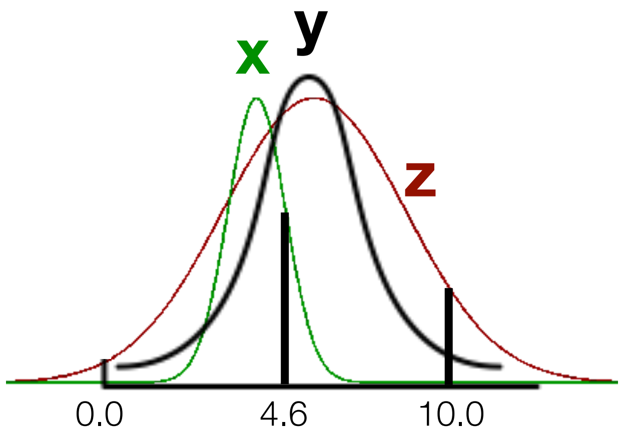
Eva Darulova

The talk is on 6th!

Our Goal

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {

    x:= gaussian(0.0, 4.6)
    y:= gaussian(0.0, 10.0)
    z:= gaussian(0.0, 10.0)
```



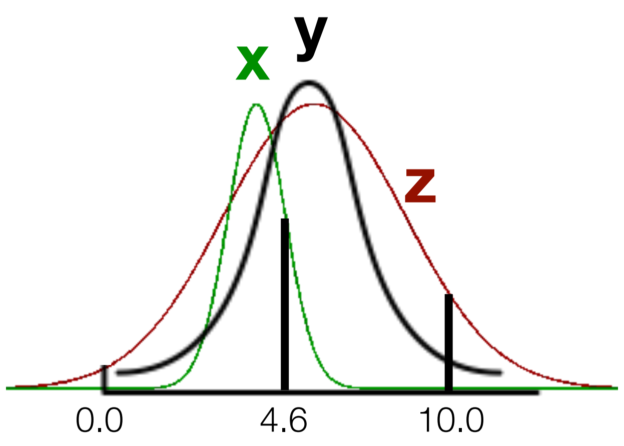

```
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    return res +/- error
} ensuring (error <= 0.00199, 0.85)
```

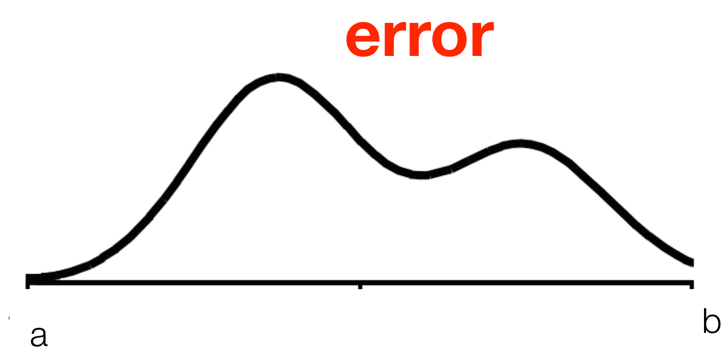
Our Goal

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {

    x:= gaussian(0.0, 4.6)
    y:= gaussian(0.0, 10.0)
    z:= gaussian(0.0, 10.0)

    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    return res +/- error
} ensuring (error <= 0.00199, 0.85)
```





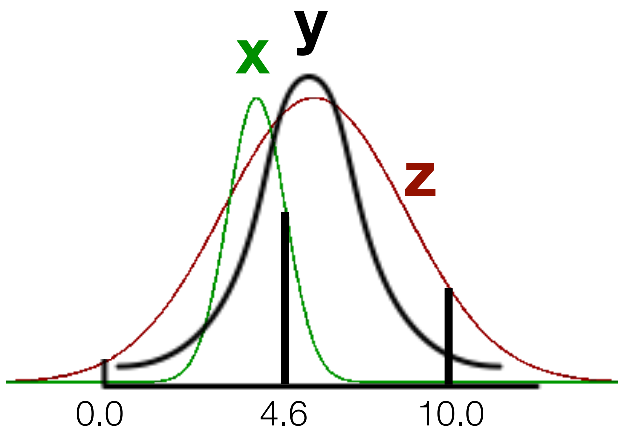
- Compute **probability distribution** of **error**

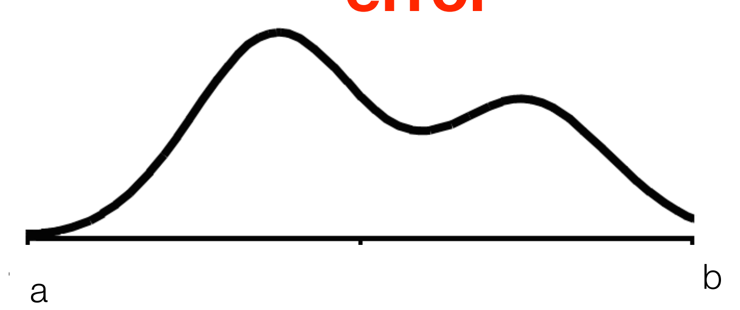
Our Goal

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {

    x:= gaussian(0.0, 4.6)
    y:= gaussian(0.0, 10.0)
    z:= gaussian(0.0, 10.0)

    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    return res +/- error
} ensuring (error <= 0.00199, 0.85)
```



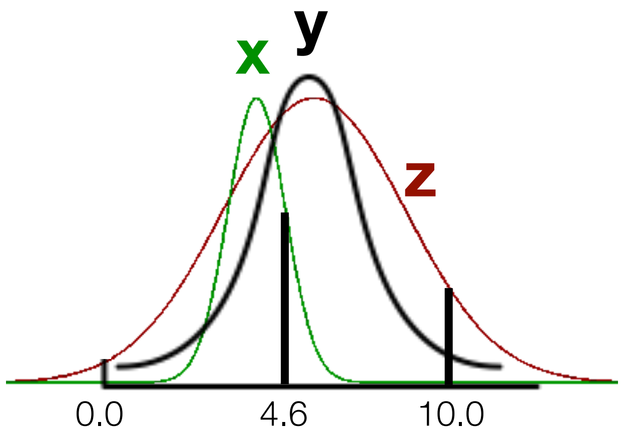


- Compute **probability distribution** of **error**
- Compute a **smaller** **error** given a **threshold**

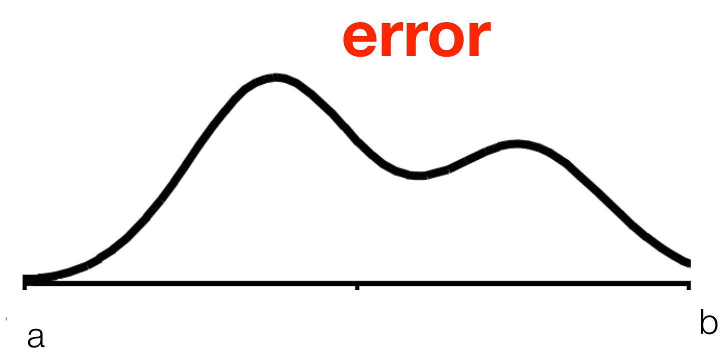
In this talk

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {

    x:= gaussian(0.0, 4.6)
    y:= gaussian(0.0, 10.0)
    z:= gaussian(0.0, 10.0)
```




```
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    return res +/- error
} ensuring (error <= 0.00199, 0.85)
```

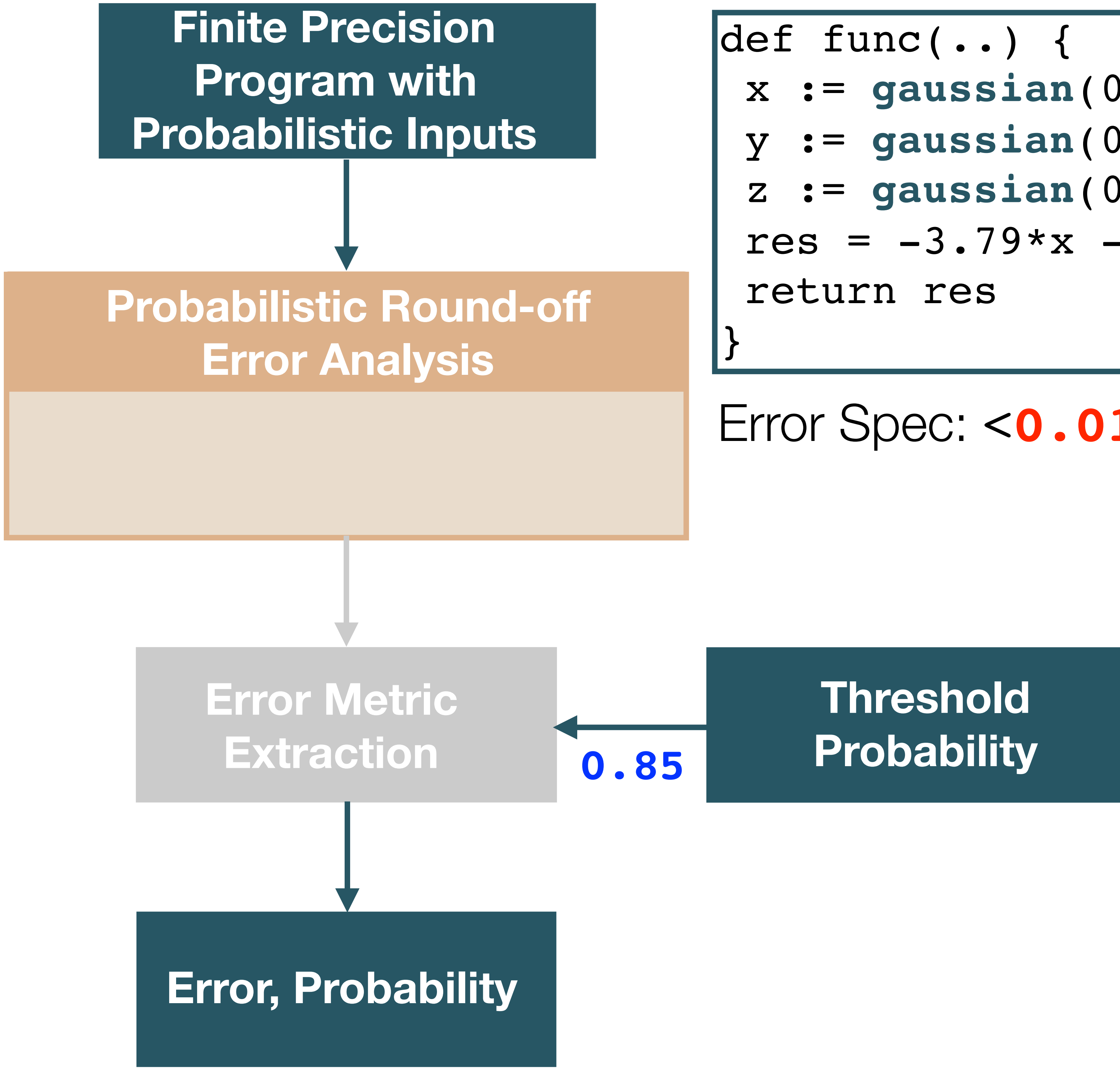


- Compute **probability distribution** of **error**
- Compute a **smaller** error given a **threshold**

considering **Probabilistic Error Specification**

<0.019, 0.9>, <0.049, 0.1>

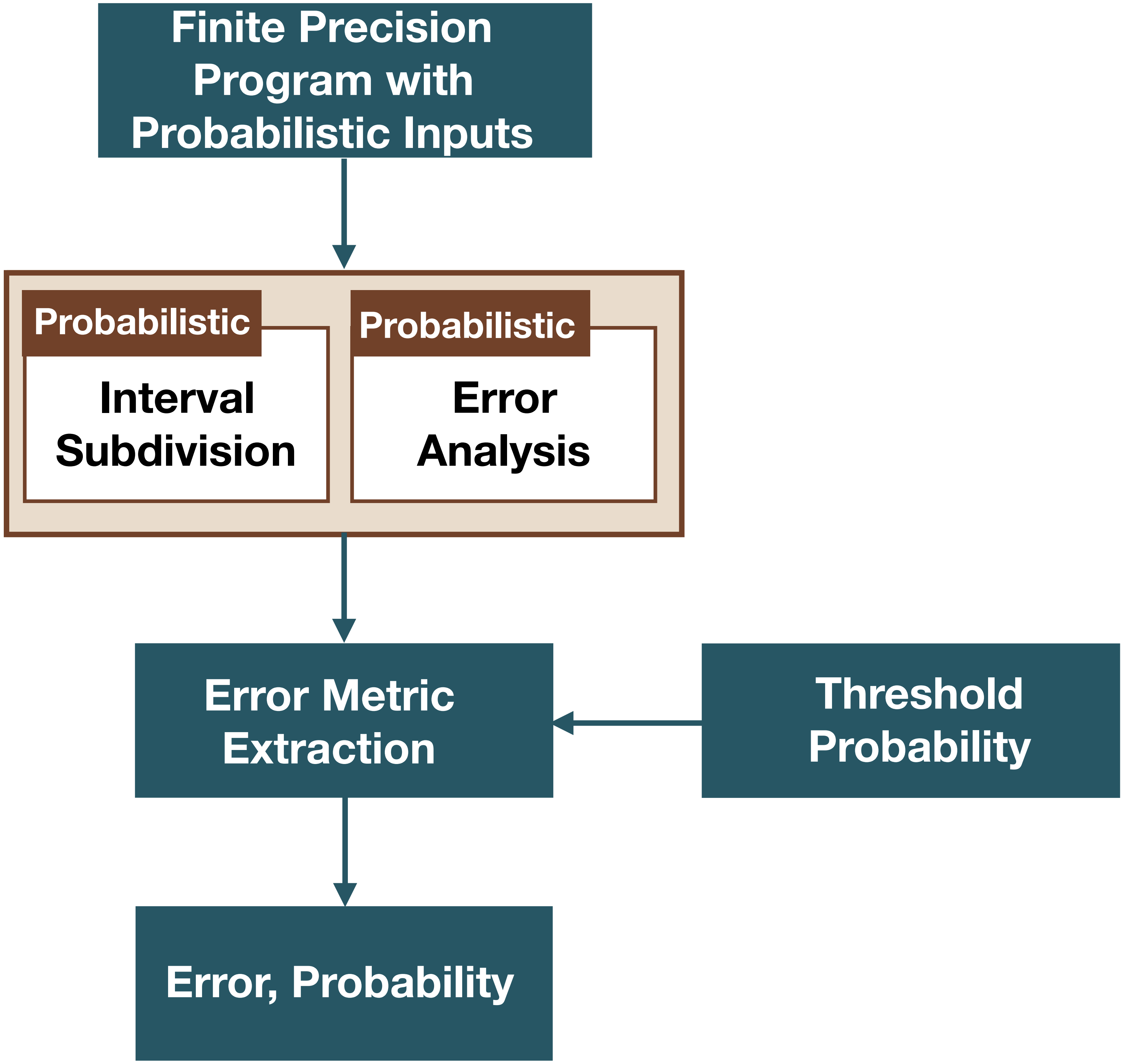
Probabilistic Error Analysis



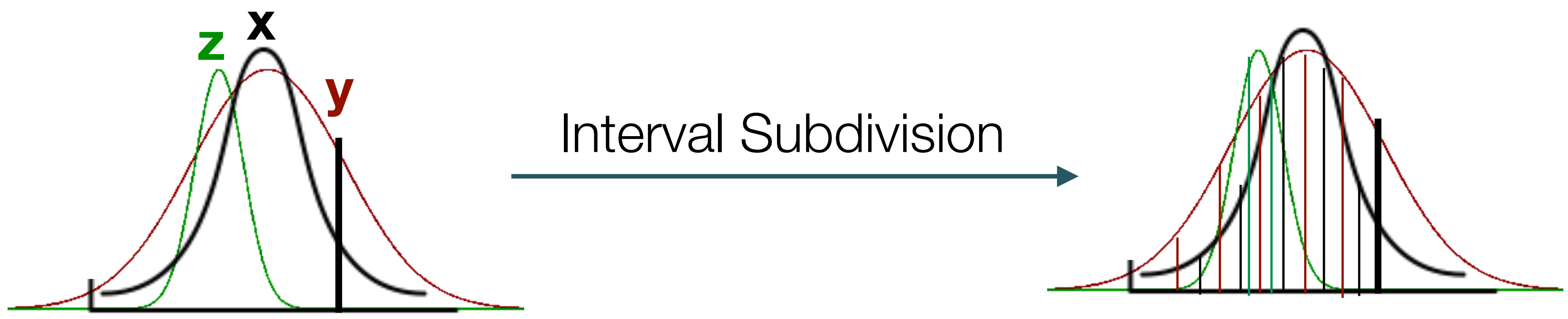

```
def func(..) {
 x := gaussian(0.0, 4.6)
 y := gaussian(0.0, 10.0)
 z := gaussian(0.0, 10.0)
 res = -3.79*x - 5.44*y + 9.73*z + 4.52
 return res
}
```

Error Spec: <0.019, 0.9>, <0.049, 0.1>

Probabilistic Error Analysis

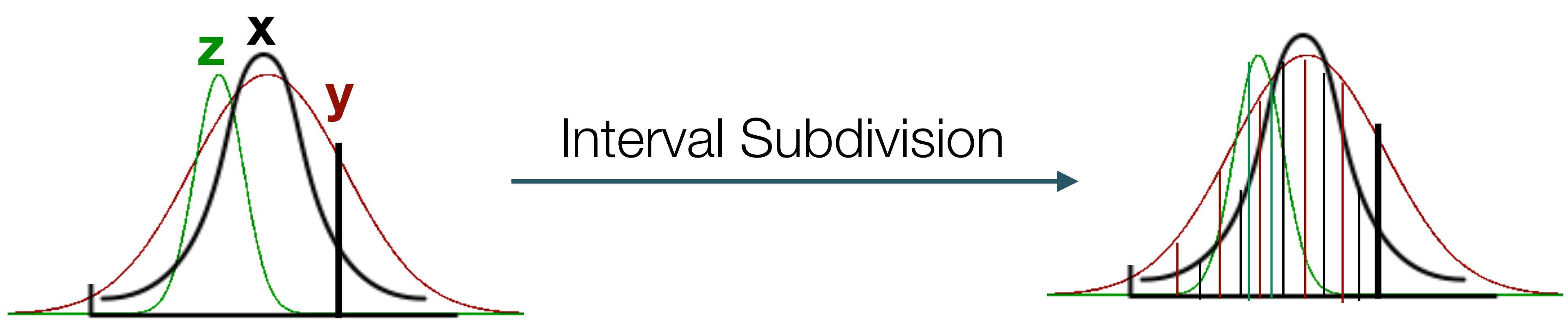


Probabilistic Interval Subdivision



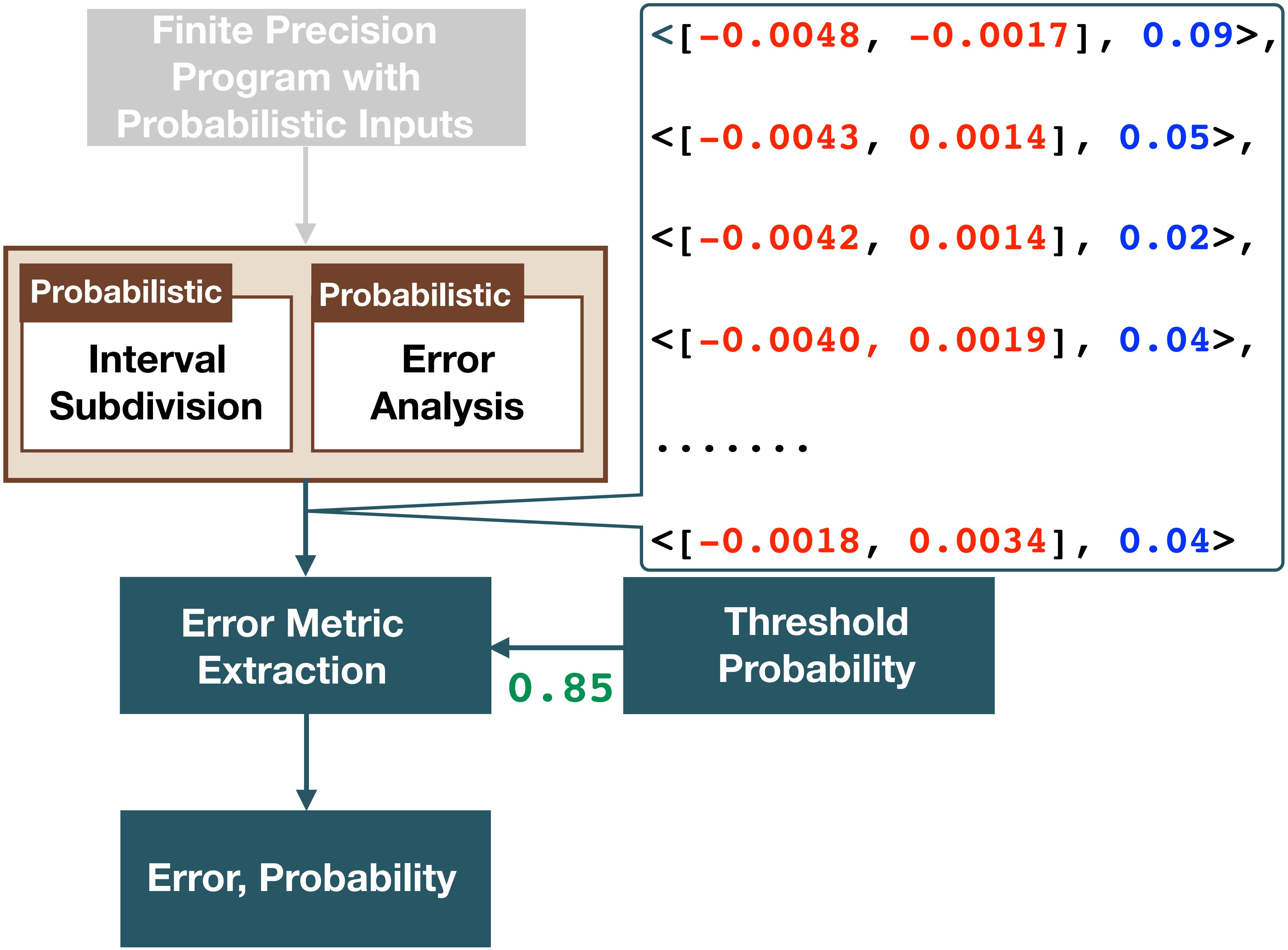
- Keeps the **probabilities** of the subdomains
- Generates a set of subdomains with their probabilities

Probabilistic Error Analysis



- Keeps the **probabilities** of the subdomains
- Generates a set of subdomains with their probabilities
- **Probabilistic Error Analysis** for each subdomain
- Normalize error distribution with probabilities of subdomains

Probabilistic Error Analysis



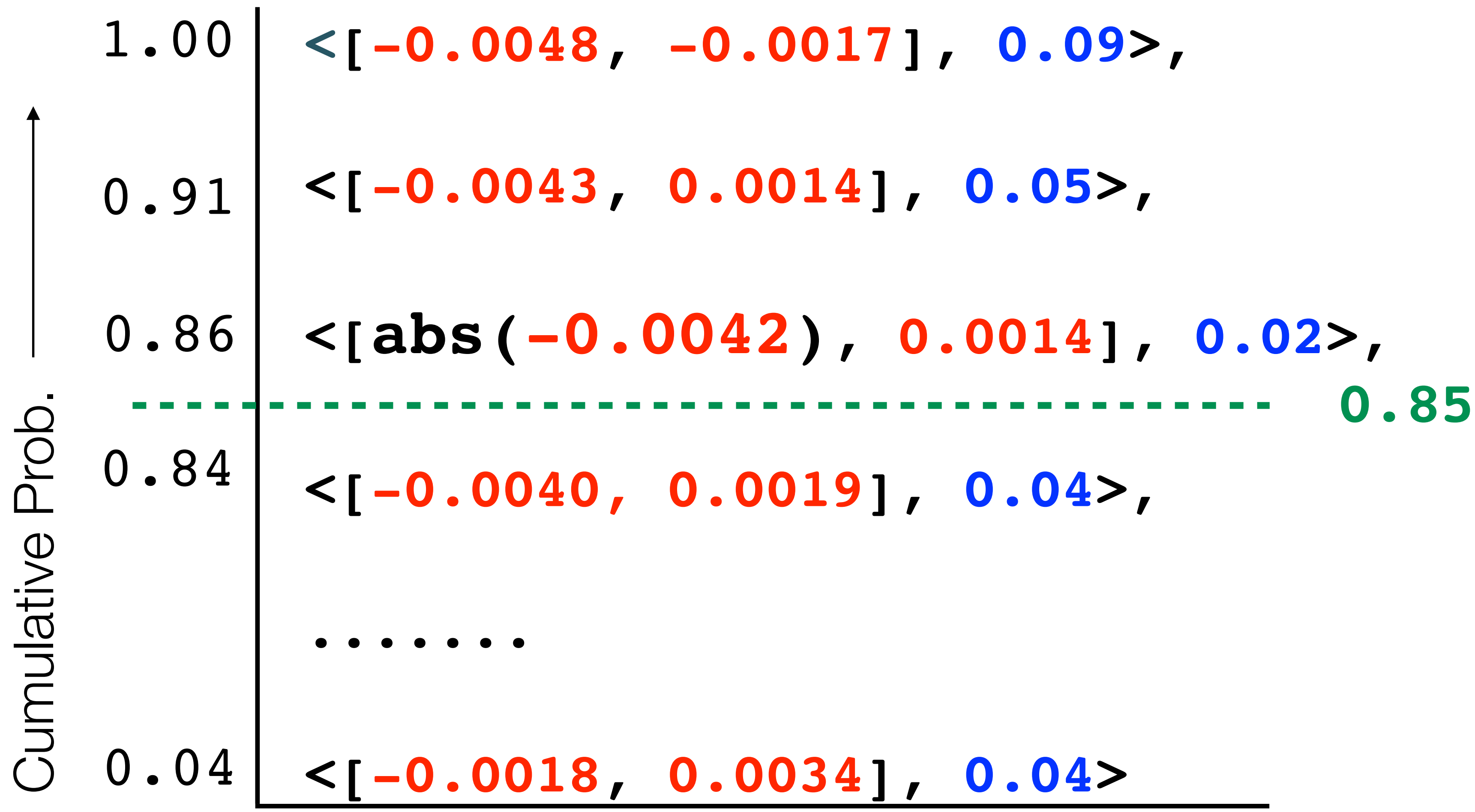
Error Metric Extraction

Threshold probability = **0.85**

Cumulative Prob. ↑	
1.00	<[-0.0048, -0.0017], 0.09>,
0.91	<[-0.0043, 0.0014], 0.05>,
0.86	<[-0.0042, 0.0014], 0.02>,
0.84	<[-0.0040, 0.0019], 0.04>,
	....
0.04	<[-0.0018, 0.0034], 0.04>

Error Metric Extraction

Threshold probability = **0.85**



Return the maximum error with probability

Error, Probability: **0.0042, 0.86**

Results: Probabilistic Error Specification

Benchmarks	Worst Case Error (state-of-the-art)	Prob analysis + Prob subdiv (% reduction)
sineOrder3		
sqrt		
bspline1		
rigidbody2		
traincar2		
filter4		
cubic		
classIDX0		
polyIDX1		
neuron		

Reduction % with 0.85 threshold probability for 32 bit floating-point errors, gaussian input distributions considering $4 \times \epsilon_m$ error happens with 0.1 probability

Results: Probabilistic Error Specification

Benchmarks	Worst Case Error (state-of-the-art)	Prob analysis + Prob subdiv (% reduction)
sineOrder3	1.28E-06	
sqrt	4.16E-04	
bspline1	7.39E-07	
rigidbody2	2.21E-02	
traincar2	3.45E-03	
filter4	3.81E-07	
cubic	3.08E-06	
classIDX0	9.25E-06	
polyIDX1	2.18E-03	
neuron	1.56E-04	

Reduction % with 0.85 threshold probability for 32 bit floating-point errors, gaussian input distributions considering $4 \times \epsilon_m$ error happens with 0.1 probability

Results: Probabilistic Error Specification

Benchmarks	Worst Case Error (state-of-the-art)	Prob analysis + Prob subdiv (% reduction)
sineOrder3	1.28E-06	**-52.9**
sqrt	4.16E-04	**-56.6**
bspline1	7.39E-07	**-40.2**
rigidbody2	2.21E-02	**-13.5**
traincar2	3.45E-03	**-13.6**
filter4	3.81E-07	**-47.5**
cubic	3.08E-06	**-41.9**
classIDX0	9.25E-06	**-18.7**
polyIDX1	2.18E-03	**-10.6**
neuron	1.56E-04	**-41.7**

Reduction % with 0.85 threshold probability for 32 bit floating-point errors, gaussian input distributions considering $4 \times \epsilon_m$ error happens with 0.1 probability

Summary

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
   require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   if (res <= 0.0)
     raiseAlarm()
   else
     doNothing()
   return res
}
```

Sound Analysis to compute Wrong path probability

Sound Analysis to compute a precise bound on the error

by taking into account the probability distribution of inputs

Summary

```
def func(x:Float32, y:Float32, z:Float32): Float32 = {
   require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
   val res = -3.79*x - 5.44*y + 9.73*z + 4.52
   if (res <= 0.0)
     raiseAlarm()
   else
     doNothing()
   return res
}
```

Sound Analysis to compute Wrong path probability

Sound Analysis to compute a precise bound on the error

by taking into account the probability distribution of inputs

Ranges and **distributions** **were provided**

Ongoing Research: Scaling up

```
def func(a:Float32, b:Float32, c:Float32): Float32 = {
    ...
    require (? <= x <= ? && ? <= y, z <= ?)
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()
    else
      doNothing()
    ...
}
```

Goal: Compute the **ranges** **automatically**

Challenges:

- **Static Analysis** provides **sound** domain **bounds**, **does not scale**
- **Dynamic Analysis scales** for real-world programs, **not sound**

Ongoing Research: Scaling up

```
def func(a:Float32, b:Float32, c:Float32): Float32 = {
    ...
    require (? <= x <= ? && ? <= y, z <= ?)
    val res = -3.79*x - 5.44*y + 9.73*z + 4.52
    if (res <= 0.0)
      raiseAlarm()
    else
      doNothing()
    ...
}
```

Our Idea: Combine them to compute the ranges automatically

More ideas?