

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

EnerJ: Approximate Data Types for Safe and General Low-Power Computation

Adrian Sampson Werner Dietl Emily Fortuna Danushen Gnanapragasam
Luis Ceze Dan Grossman
University of Washington, Department of Computer Science & Engineering
<http://sampa.cs.washington.edu/>

Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels

Sasa Misailovic Michael Carbin Sara Achour Zichao Qi Martin Rinard
MIT CSAIL
{misailo,mcarbin,sachour,zichaoqi,rinard}@csail.mit.edu

Abstract

Energy is increasingly a first-order concern in computer systems. Exploiting energy-accuracy trade-offs is an attractive choice in

in data-centers. More fundamentally, current trends point toward a “utilization wall,” in which the amount of active die area is limited by how much power can be fed to a chip.

Abstract

Target application domains include computations that either 1)

Resource Efficient but has Probabilistic Error behaviors

to precise data. Importantly, employing static analysis eliminates the need for dynamic checks, further reducing overhead. As a proof of concept, we develop EnerJ, an extension to Java that adds approximate data types. We also propose a hardware architecture that offers explicit approximate storage and operations to support several applications to EnerJ and show that our extensions are

MIT CSAIL
{mcarbin,misailo,rinard}@csail.mit.edu

show that applications have portions that are more tolerant and can be protected on approximate hardware platforms. Given a combined reliability and accuracy specification, Chisel uses approximate kernel operations to synthesize an approximate computation that minimizes energy consumption while satisfying its reliability and accuracy specifications.

Towards Reversed Approximate Hardware Design

pressing and analyzing computations that run on approximate hardware platforms. These platforms use unreliable and unreliable versions of standard arithmetic and logical instructions as well as reliable and unreliable memories. Rely enables a developer to manually identify unreliable instructions and vary (grosse,drechsle}@cs.uni-bremen.de

Abstract

Emerging high-performance architectures are anticipated to contain unreliable components that may exhibit *soft errors*, which silently corrupt the results of computations. Full detection and masking of soft errors is challenging, expensive, and, for some applications, unnecessary. For example, approximate computing applications (such as multimedia processing, machine learning, and big data analytics) can often naturally tolerate soft errors.

We present Rely, a programming language that enables developers to reason about the quantitative reliability of an application – namely, the probability that it produces the correct result when executed on unreliable hardware. Rely allows developers to specify the reliability requirements for each value that a function produces.

We present a static quantitative reliability analysis that verifies quantitative requirements on the reliability of an application, enabling a developer to perform sound and verified reliability engineering. The analysis takes a Rely program with a reliability specification and a hardware specification

1. Introduction

System reliability is a major challenge in the design of emerging architectures. Energy efficiency and circuit scaling are becoming major goals when designing new devices. However, aggressively pursuing these design goals can often increase the frequency of *soft errors* in small [67] and large systems [10] alike. Researchers have developed numerous techniques for detecting and masking soft errors in both hardware [23] and software [20, 53, 57, 64]. These techniques typically come at the price of increased execution time, increased energy consumption, or both.

Many computations, however, can tolerate occasional unmasked errors. An *approximate computation* (including many multimedia, financial, machine learning, and big data analytics applications) can often acceptably tolerate occasional errors in its execution and/or the data that it manipulates [16, 44, 59]. A *checkable computation* can be augmented with an efficient checker that verifies the acceptability of the computation’s results [8, 9, 35, 55]. If the checker does detect an error, it can reexecute the computation to

Abstract—Approximate computing is an emerging design paradigm for trading off computational accuracy for computational effort. Due to their inherited error resilience many applications significantly benefit from approximate computing. To realize approximation, dedicated approximate circuits have been developed and provide a solid foundation for energy and time efficient computing. However, when it comes to the design and integration of the approximate HW, complex error analysis is required to determine the effect of the error with respect to application specific error norms. This frequently leads to sub-optimal results. In this work, we propose to reverse the typical design flow for approximate HW and demonstrate the new flow for a first application: LU-Factorization, which is one of the most basic and most popular numerical algorithm known. The general idea of the reversed flow for approximate HW design is to start with the application and determine the required computational accuracy and application specific error bound. This allows us to push the approximate HW to its limits, while guaranteeing that the result is correct by construction wrt. the requirements. The effectiveness of our approach for LU-Factorization is shown on a well-known and large set of benchmarks.

approximate HW component is found which satisfies the application specific error bound. To overcome this deficiency, we *reverse the conventional approximate HW design flow* by starting from the application specific error bound such that we can take full advantage of the approximate HW which is *correct by construction* and thus guarantees that the application specific error bound holds.

To demonstrate our proposed reversed approximate HW design flow, we consider as a first application the problem of solving linear equations. This problem is a challenge faced in many applications: electronic circuits when using Kirchhoff’s rules or network analysis when analyzing traffic flows, to mention only a few (a lot of examples can be found in [15]). In this context, the *LU-Factorization* is one of the most common ways to solve a linear system of equations of the form $Ax = b$ directly, i.e. not using an iterative method. Essentially, LU-Factorization is based on decomposing the matrix representing the linear system into a product of two triangular matrices (one *Lower* and one *Upper* triangular

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: $\langle 0.00199, 0.9 \rangle, \langle 0.00499, 0.1 \rangle$

- 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: $\langle 0.00199, 0.9 \rangle, \langle 0.00499, 0.1 \rangle$

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() real valued program  
  else  
    doNothing() finite precision program  
  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>



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  
}
```

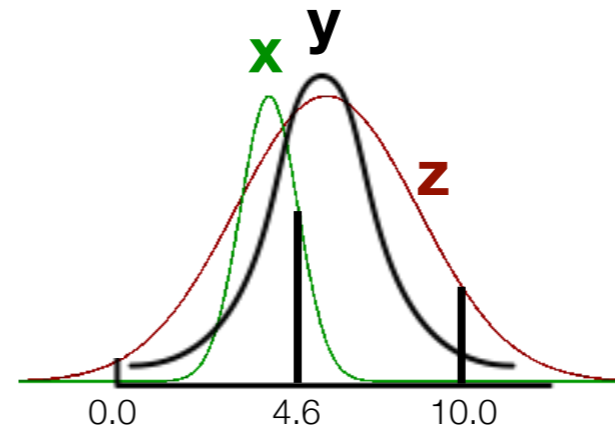
} How often?

Compute **Wrong Path Probability**

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)
```

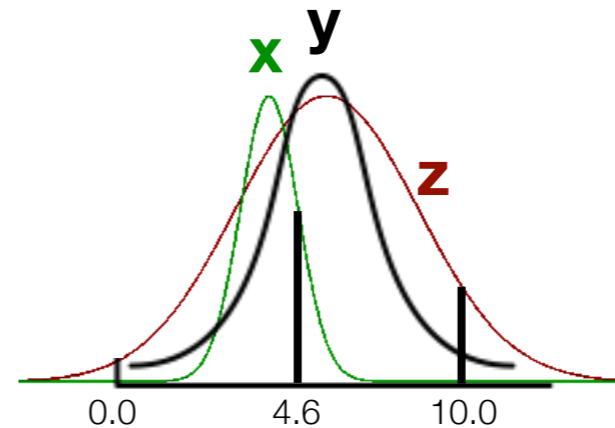


```
  val res = -3.79*x - 5.44*y + 9.73*z + 4.52  
  if (res <= 0.0)  
    raiseAlarm()  
  else  
    doNothing()  
  return res  
}
```

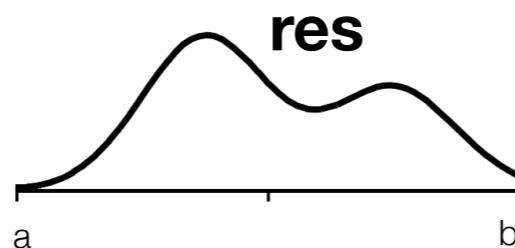
Our Goal: Probabilistic Analysis

```
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 **Wrong Path Probability**

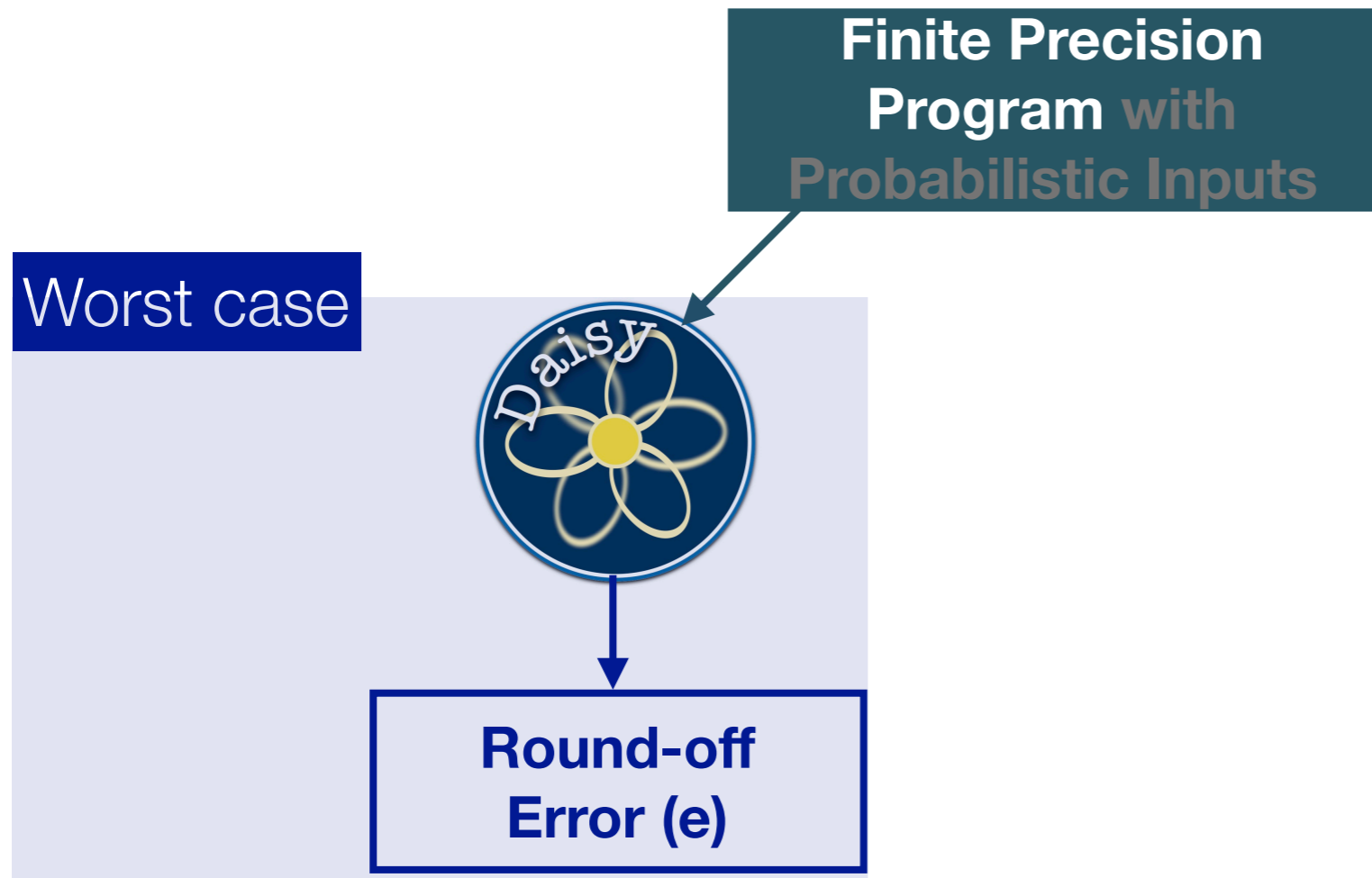
```
}
```

Overview: Sound Analysis

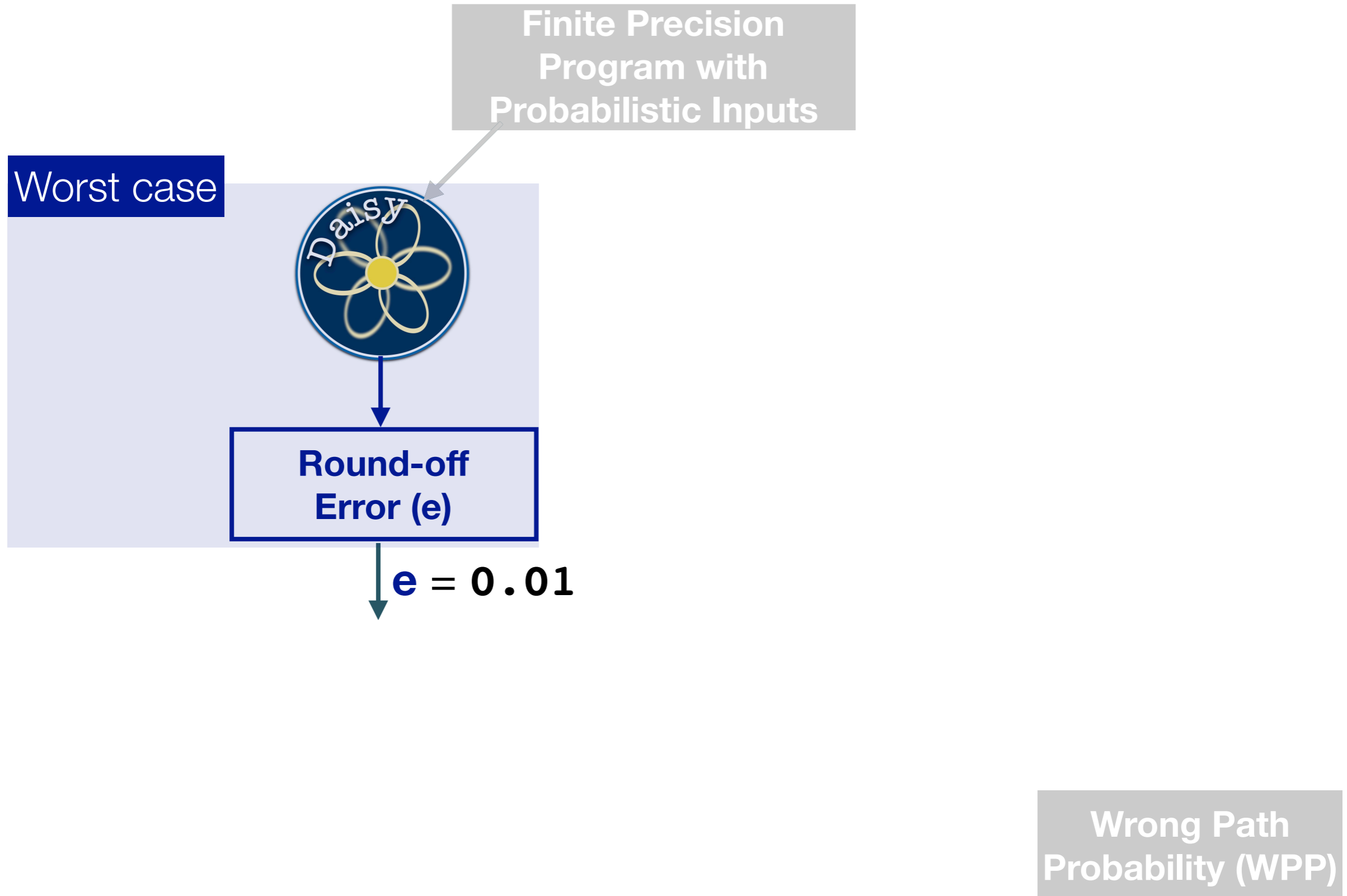
**Finite Precision
Program with
Probabilistic Inputs**

**Wrong Path
Probability (WPP)**

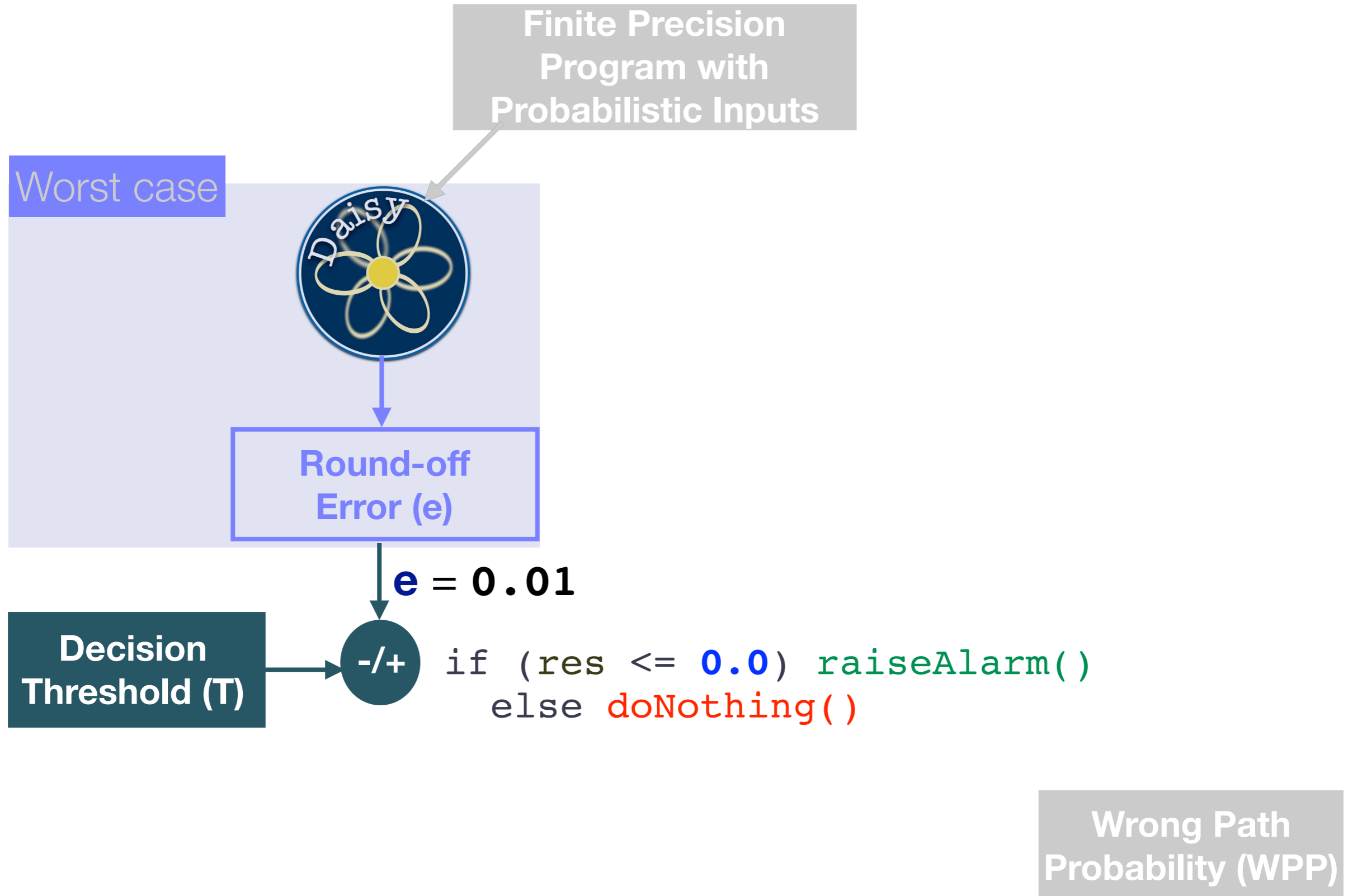
Round-off Error Analysis



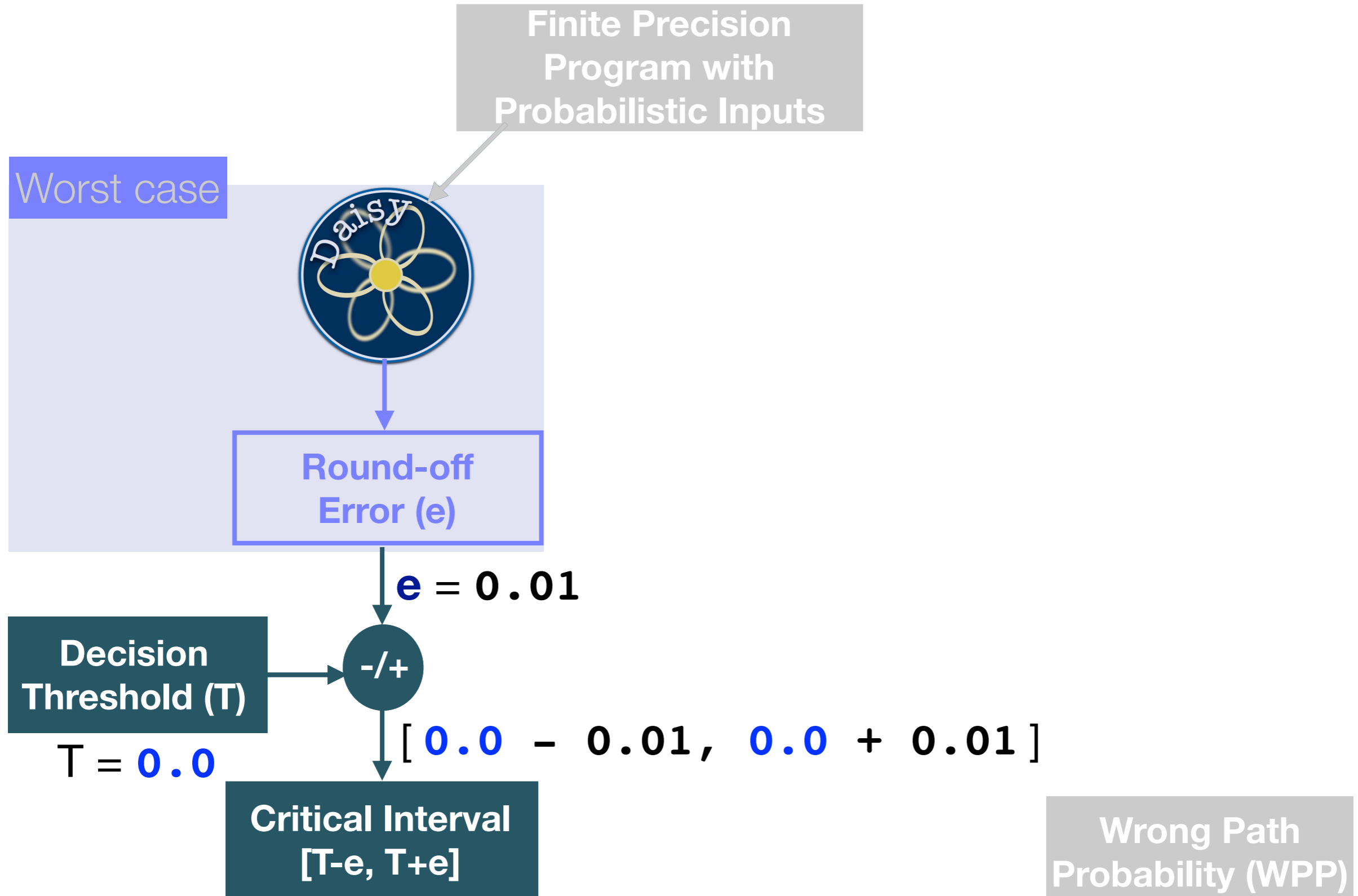
Overview: Sound Analysis



Overview: Sound Analysis



Overview: Sound Analysis



Distribution Propagation

Finite Precision Program with Probabilistic Inputs

Worst case



Round-off Error (ϵ)

Decision Threshold (T)

-/+

Critical Interval
 $[T-\epsilon, T+\epsilon]$

Probabilistic Analysis

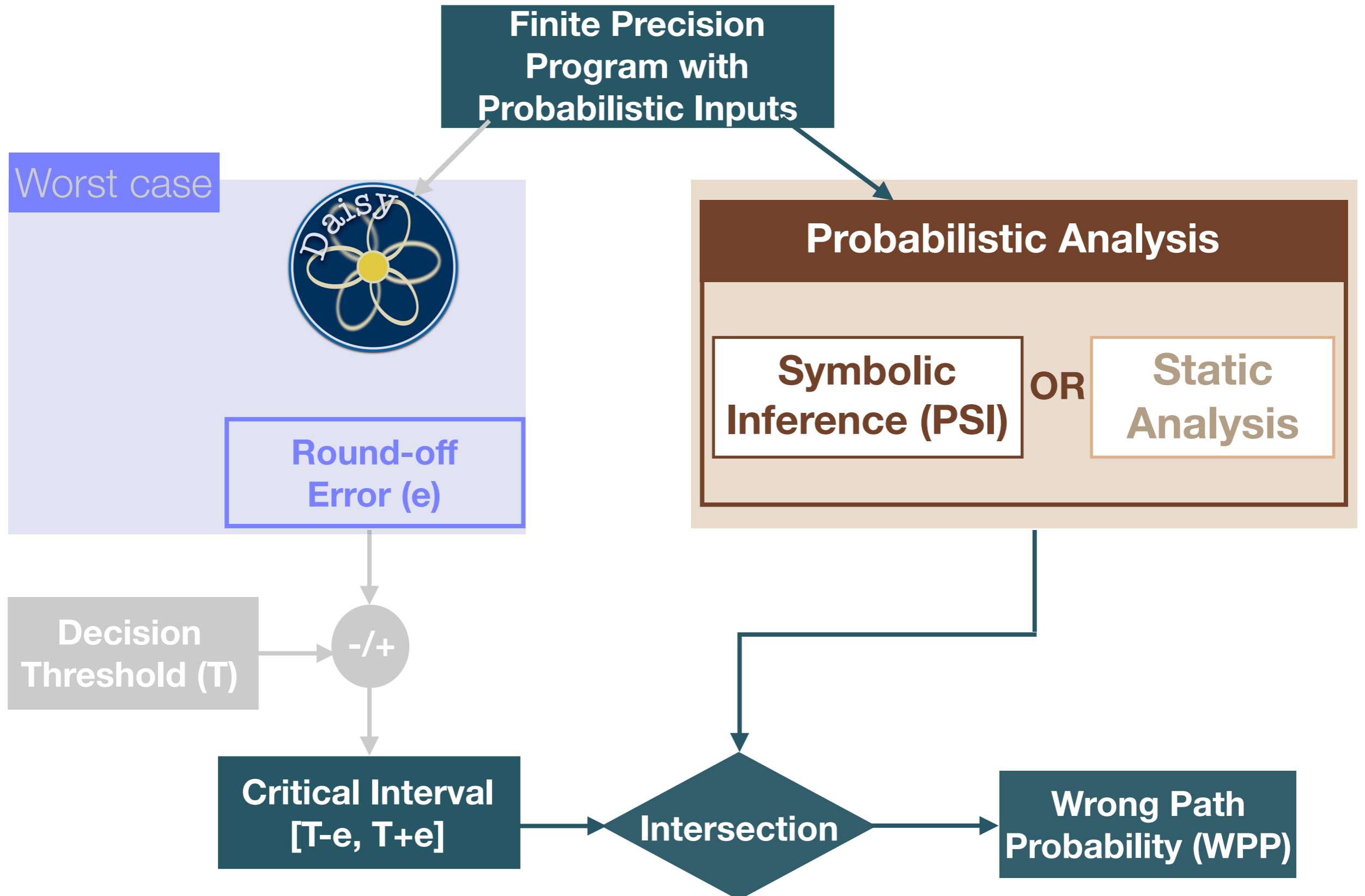
Symbolic Inference (PSI)

OR

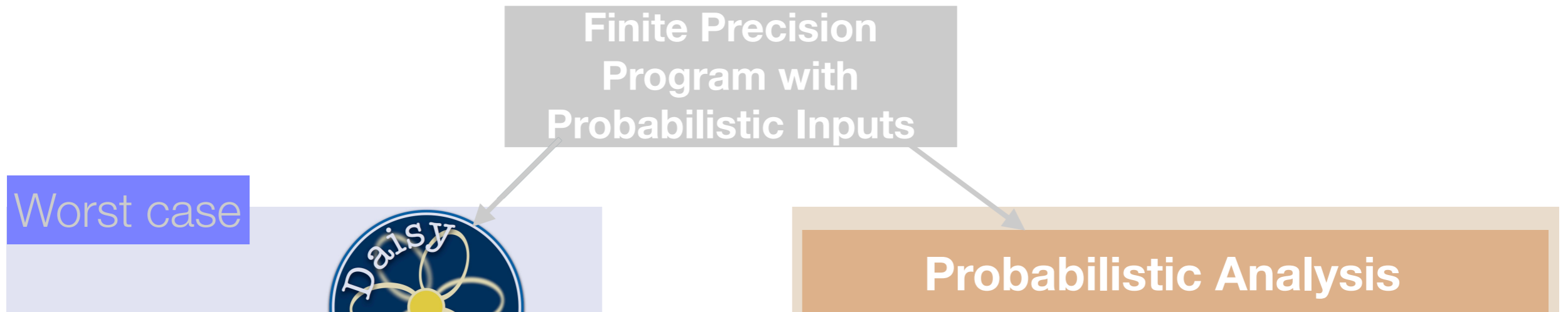
Static Analysis

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

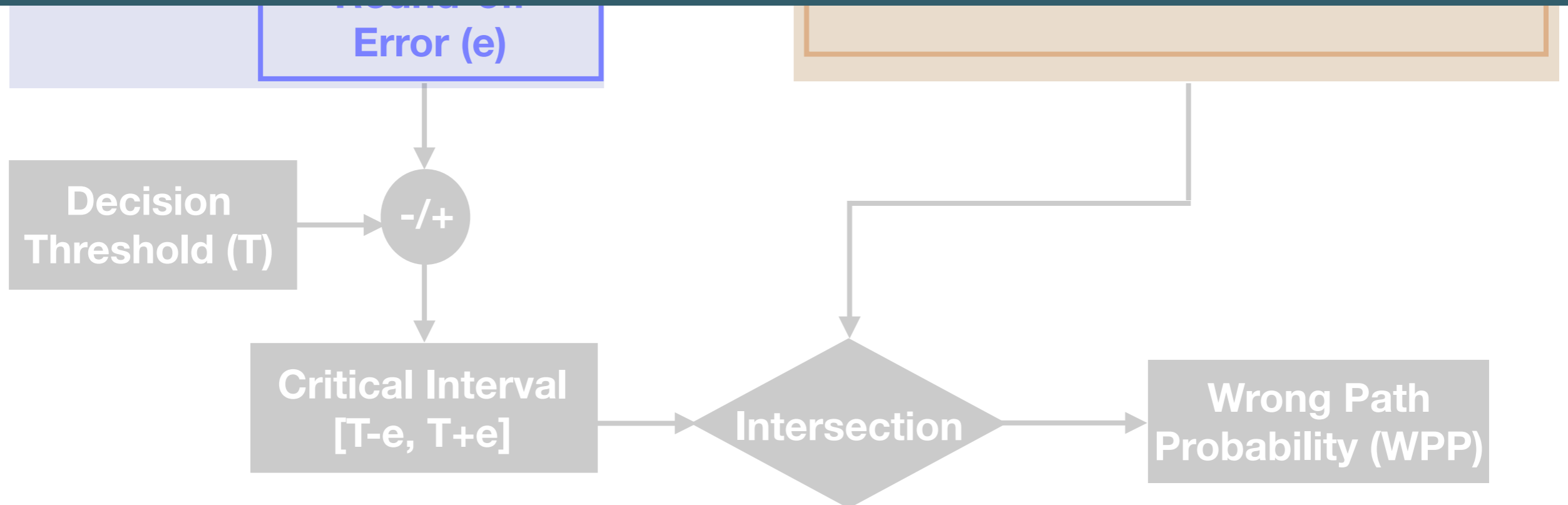
Distribution Propagation



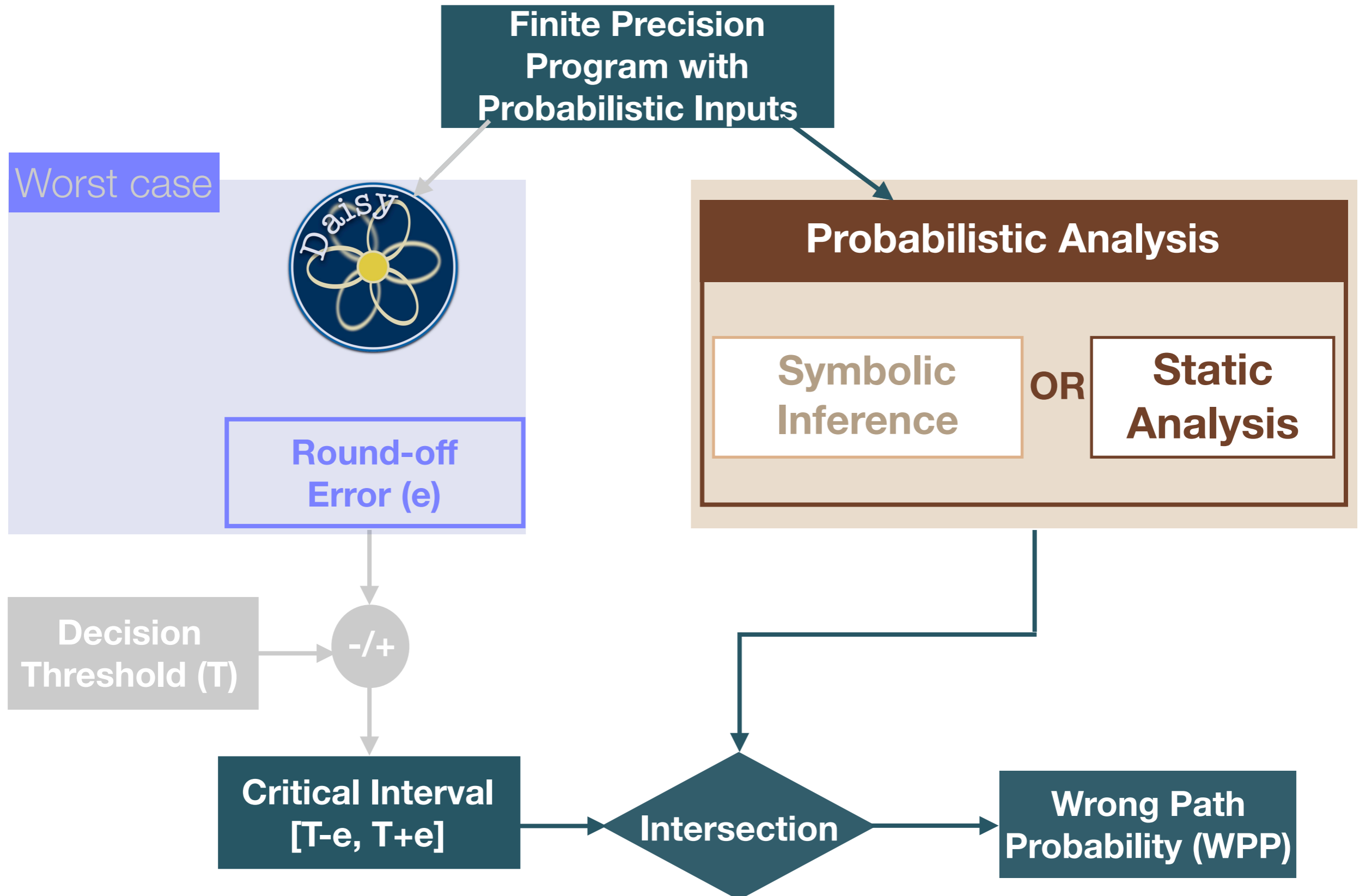
Distribution Propagation



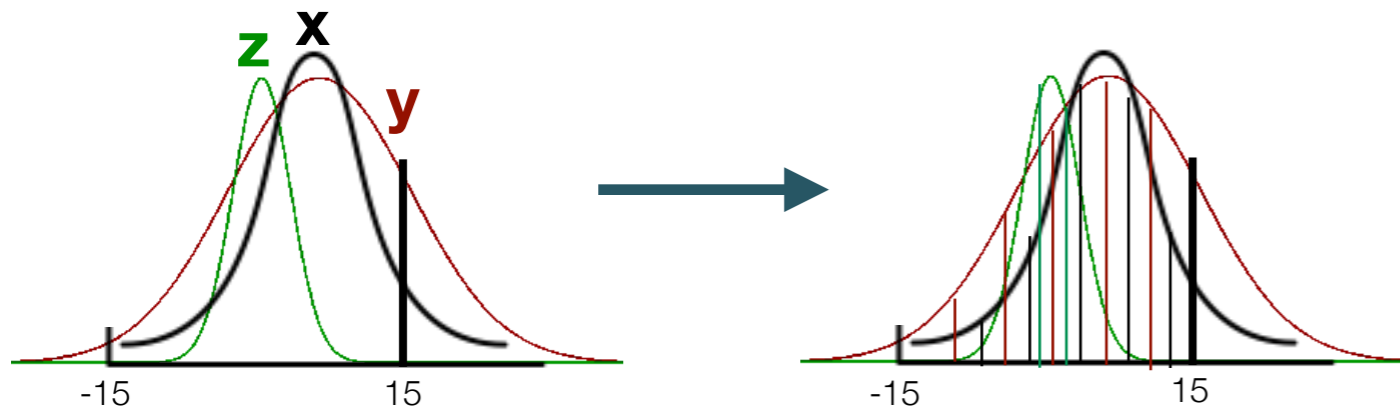
Does not scale!



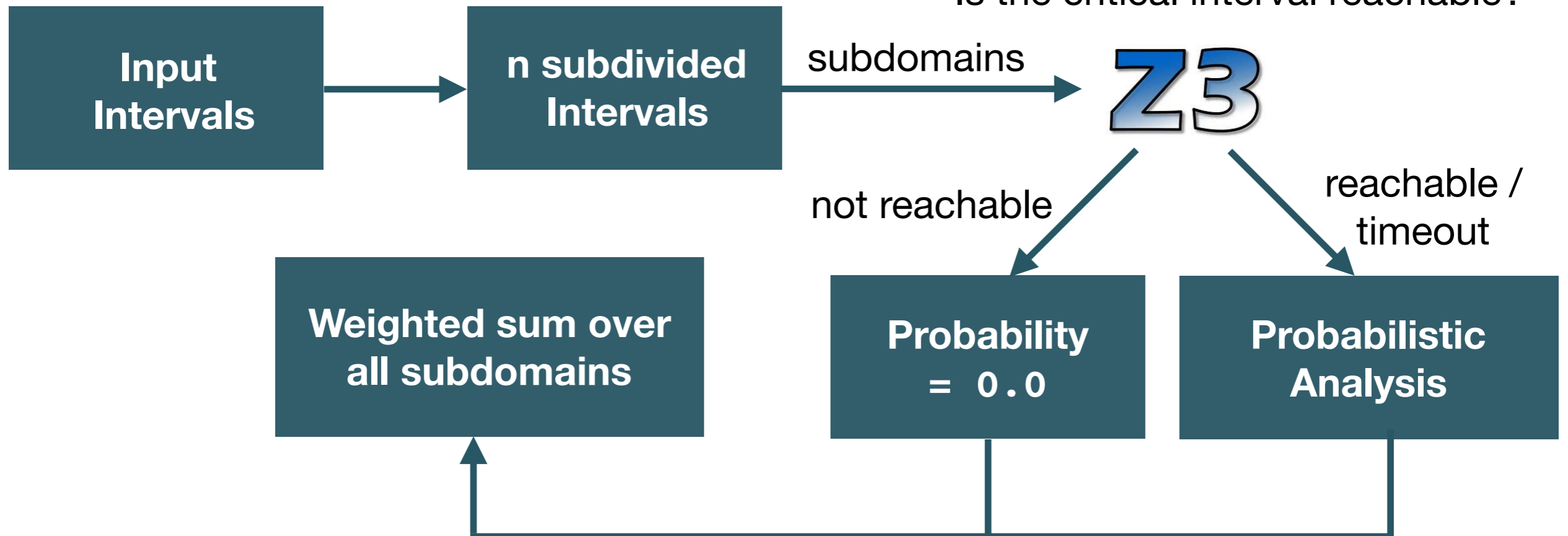
Distribution Propagation



Using Probabilistic Static Analysis



Is the critical interval reachable?



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!

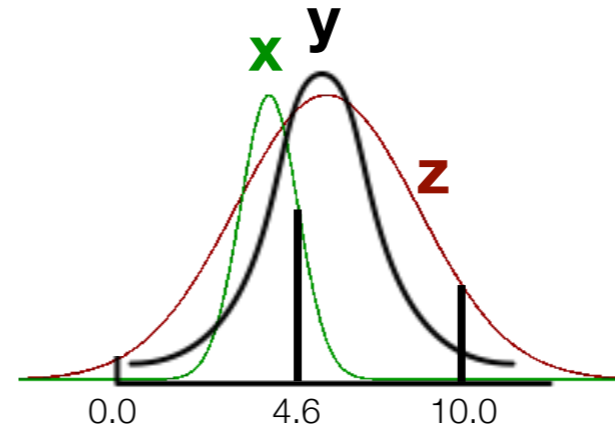
Our Goal

```
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)
```

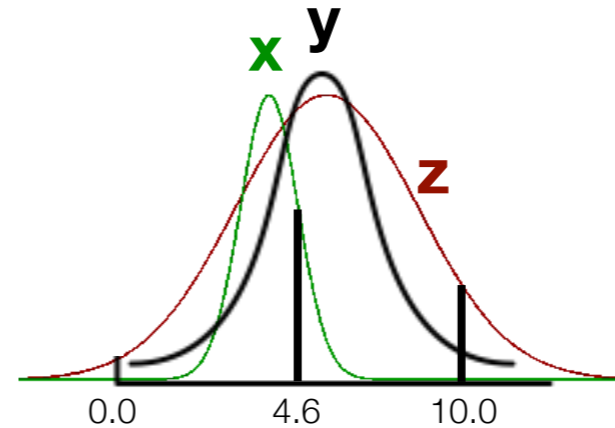
Our Goal

```
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)
```

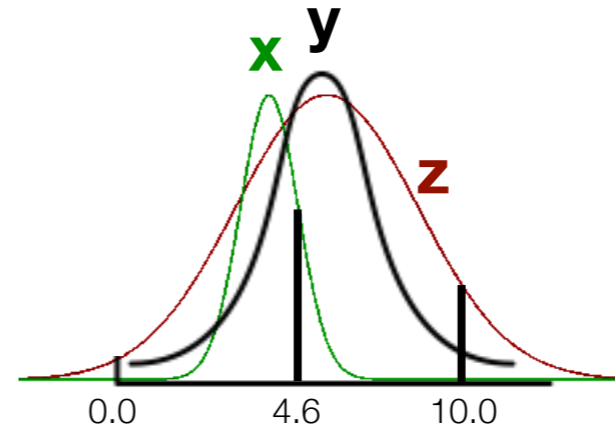


- Compute **probability distribution** of **error**

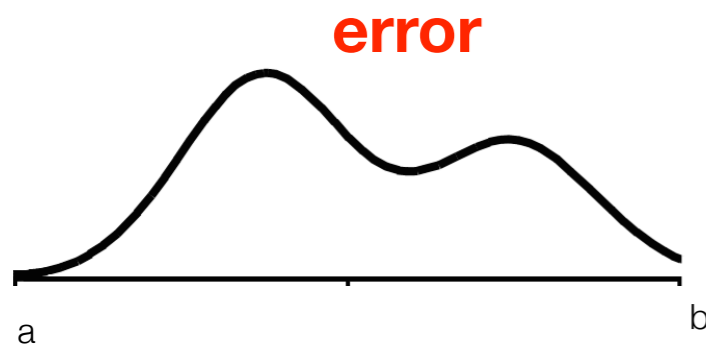
Our Goal

```
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)
```

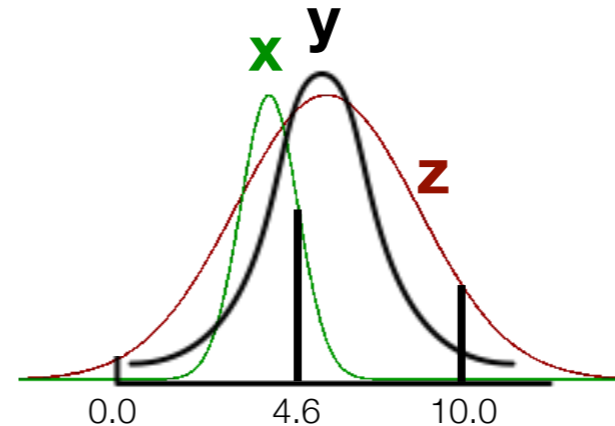


- Compute **probability distribution** of **error**
- Compute a **smaller error** given a **threshold**

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

Finite Precision
Program with
Probabilistic Inputs

```
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  
}
```

Probabilistic Round-off
Error Analysis

Error Spec: $\langle 0.019, 0.9 \rangle, \langle 0.049, 0.1 \rangle$

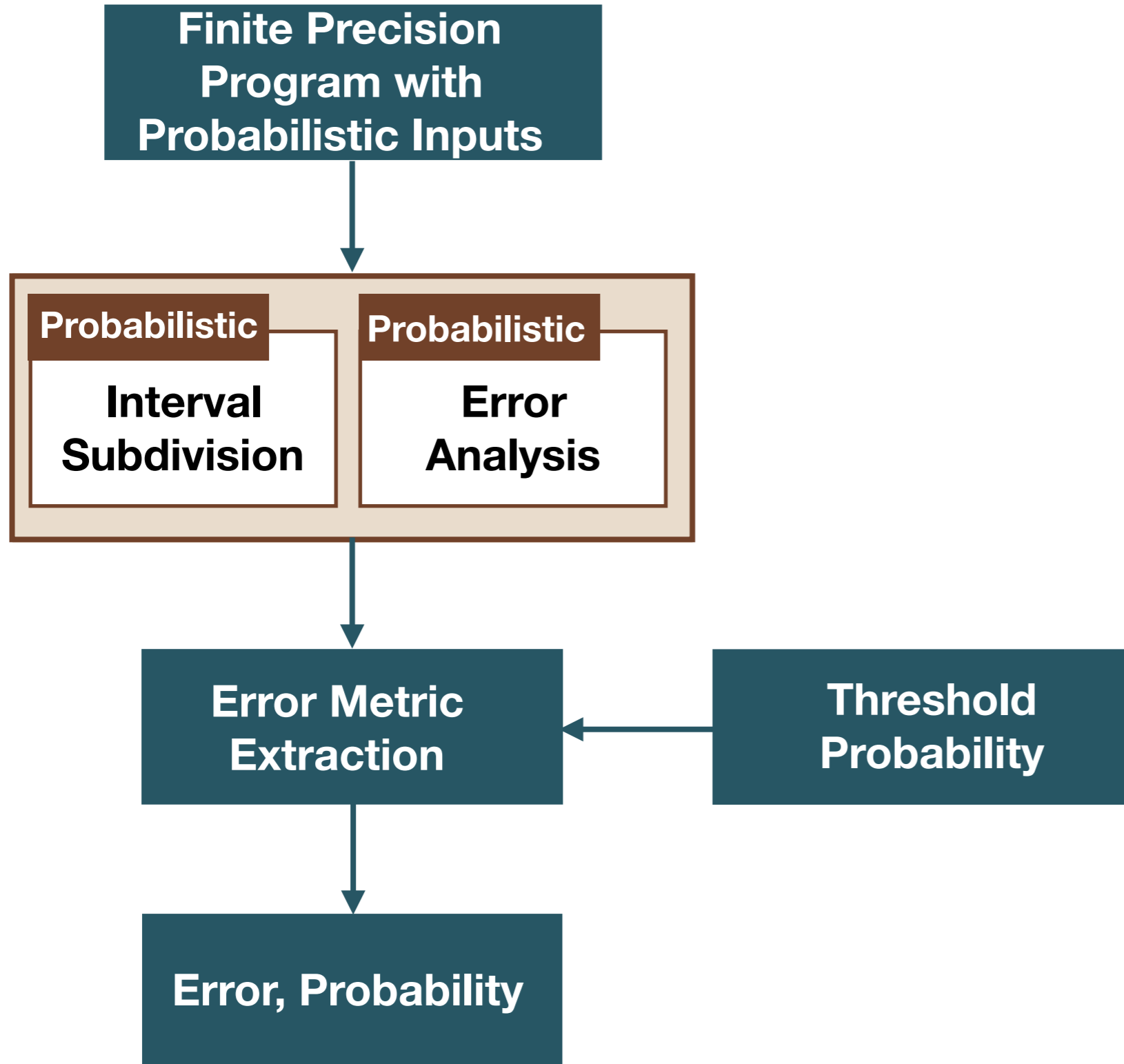
Error Metric
Extraction

Threshold
Probability

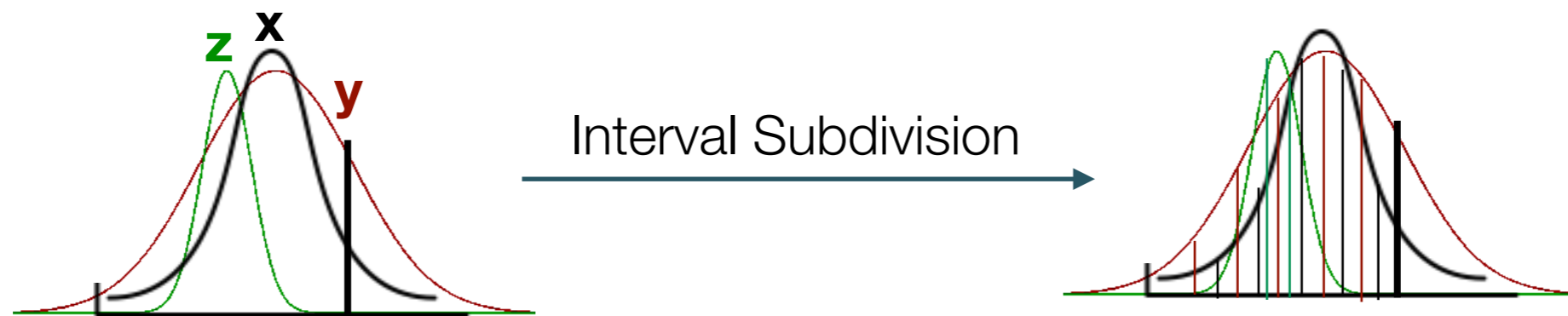
0.85

Error, Probability

Probabilistic Error Analysis

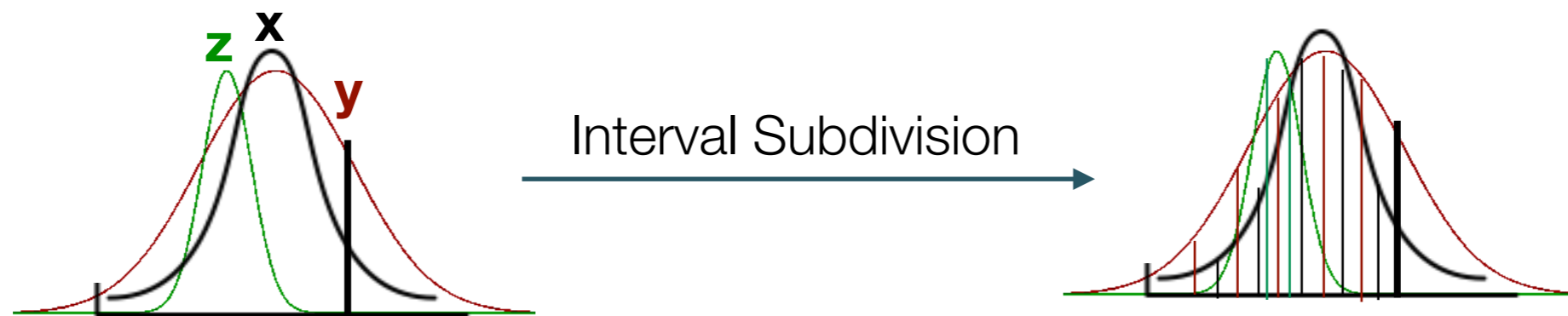


Probabilistic Interval Subdivision



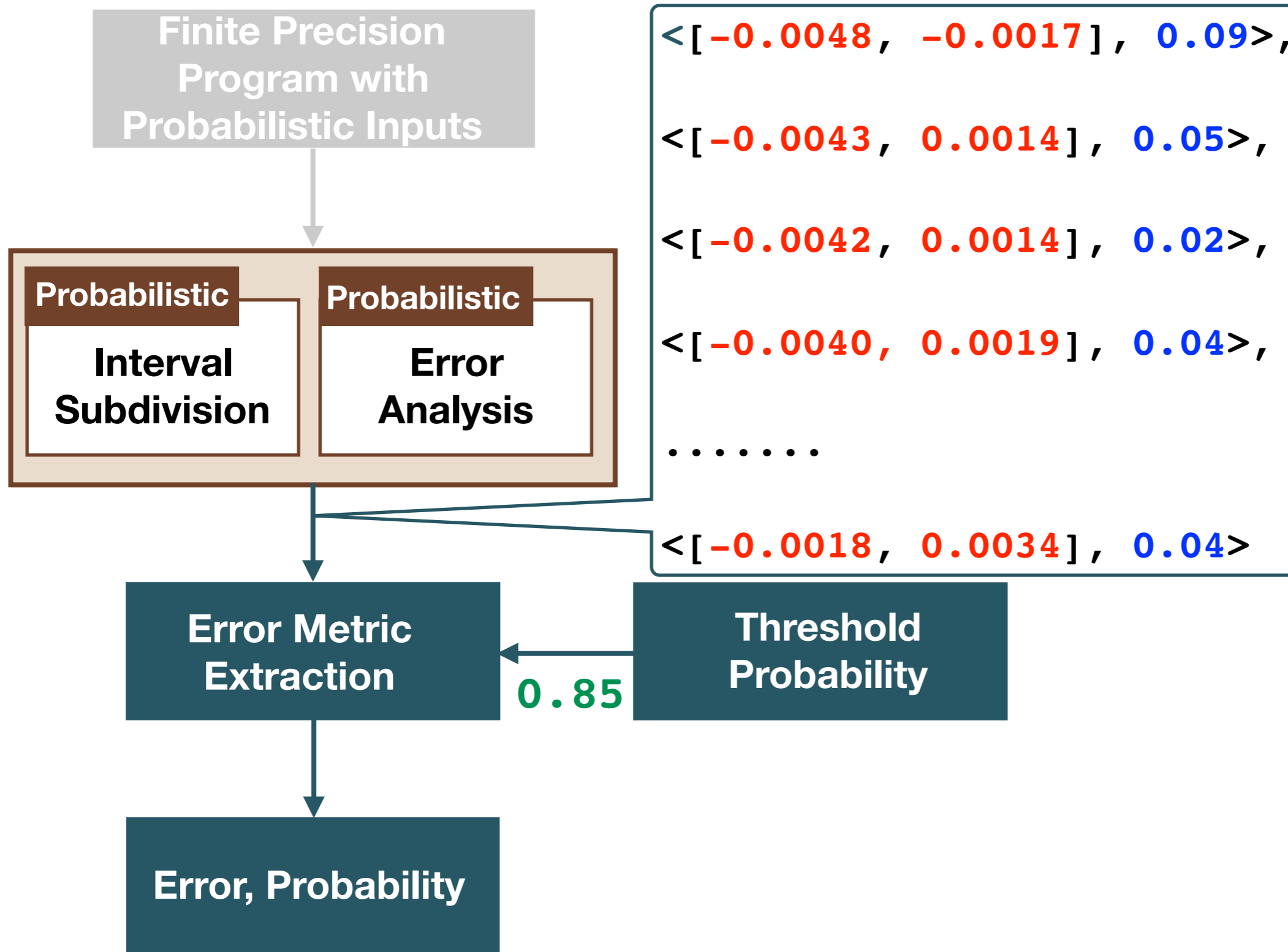
- Keeps the **probabilities** of the subdomains
- Generates a set of subdomains with their probabilities

Probabilistic Error Analysis



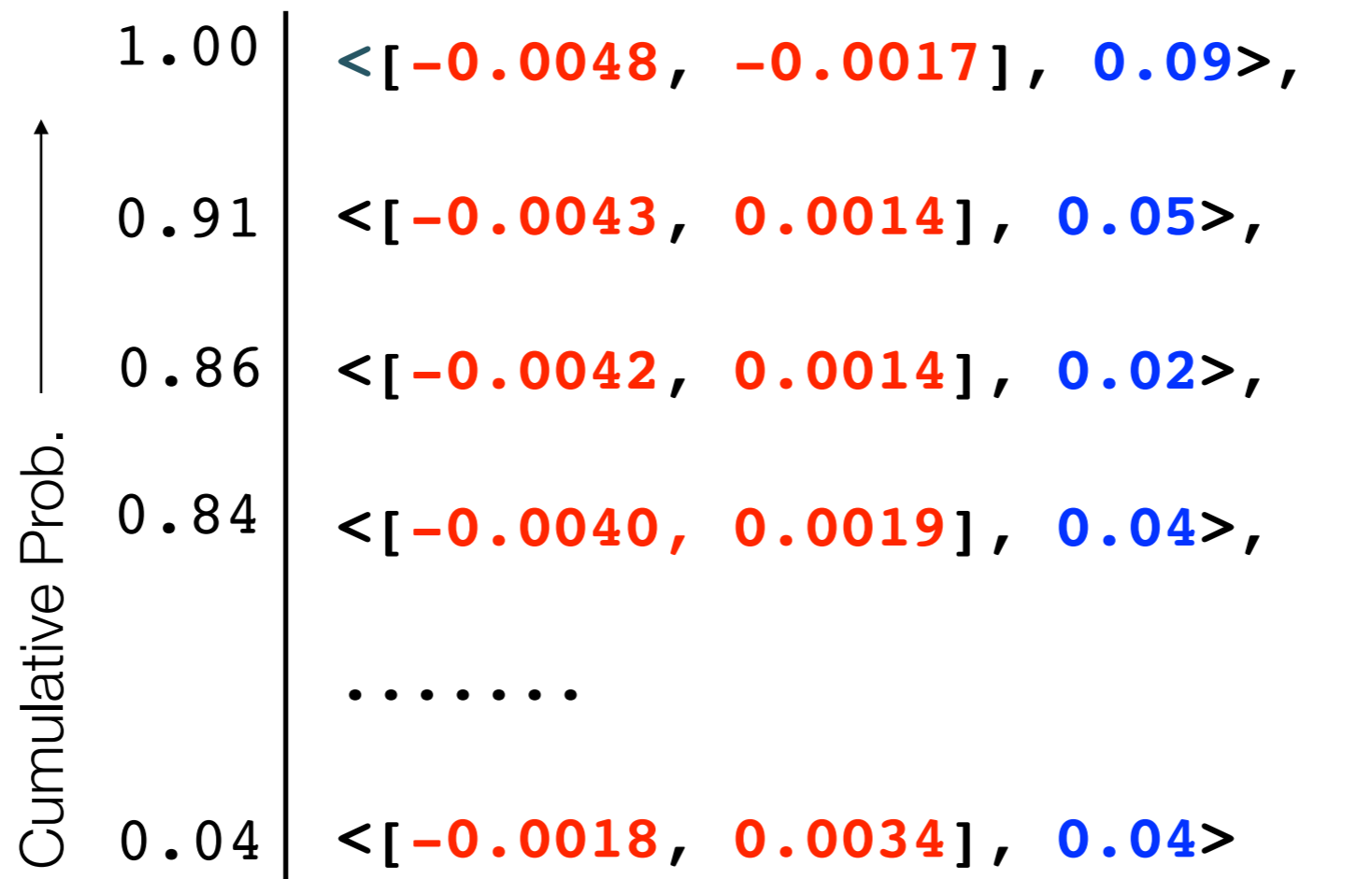
- Keeps the **probabilities** of the subdomains
- Generates a set of subdomains with their probabilities
- **Probabilistic Error Analysis** for each subdomain
- Normalize error distribution with probabilities of subdomains

Probabilistic Error Analysis



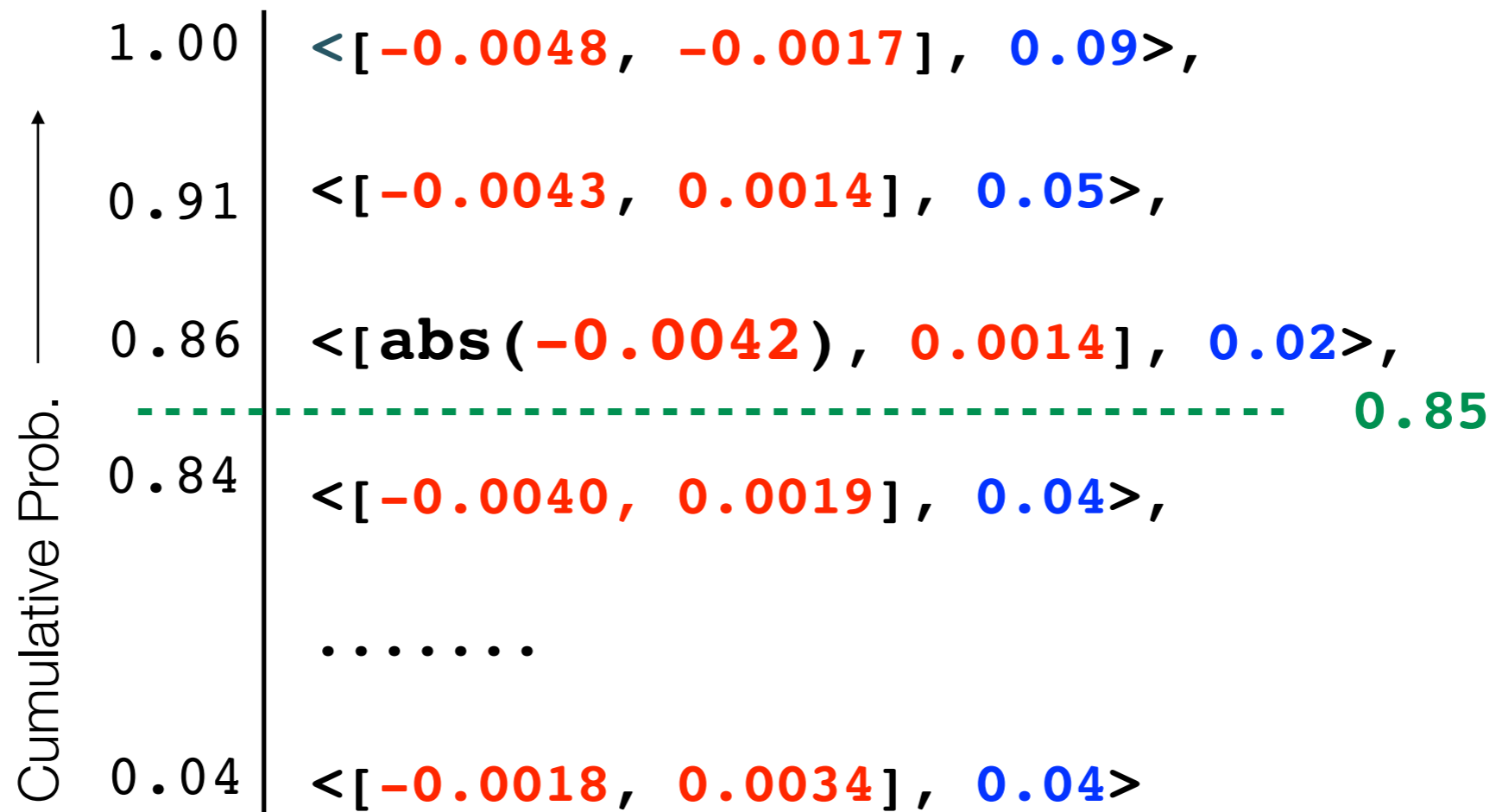
Error Metric Extraction

Threshold probability = **0.85**



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?