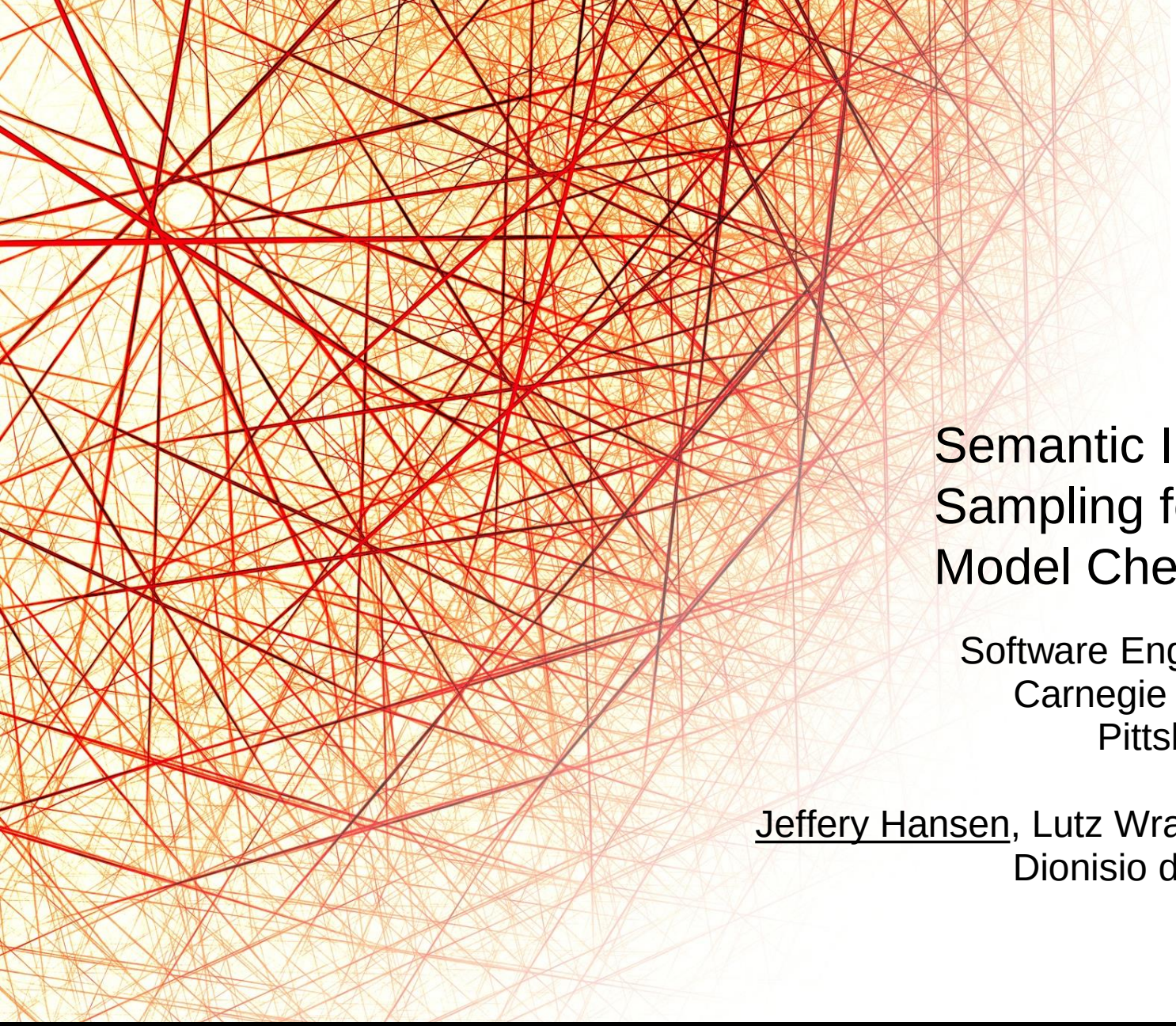


Semantic Importance Sampling for Statistical Model Checking

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Jeffery Hansen, Lutz Wrage, Sagar Chaki,
Dionisio de Niz, Mark Klein
April 15, 2015

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

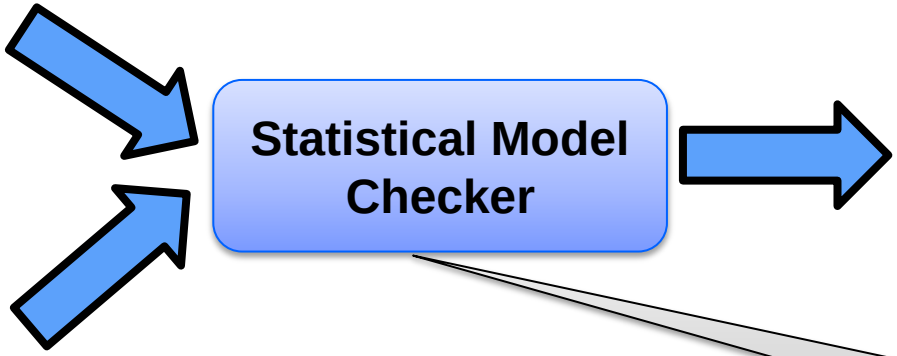
DM-0002317

Statistical Model Checking (SMC)

$$\textbf{Relative Error} = \frac{\textbf{Std. Dev.}}{\textbf{Mean}}$$

Any system $\mathcal{M}$ that takes random inputs

Probabilistic Temporal Logic Formula $\boldsymbol{\phi}$



Estimated Probability that $\mathcal{M} \vDash \boldsymbol{\phi}$ with relative error $\boldsymbol{RE}$

Based on Monte-Carlo Simulation

- System properties described in formal language (UTSL, BLTL, etc.)
- Property is tested on “sample trajectories” (sequence of states)
- Each outcome can be treated as a Bernoulli random variable (i.e., coin flip)

Statistical Model Checking

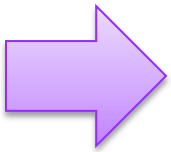
Goal: Calculate the probability p that some property holds:

$$p = E[I_{\mathcal{M} \vDash \Phi}(\vec{x})]$$

Where:

- $\vec{x}$ = vector of random variables
  - Represents all of the inputs or all random samples
- $I_{\mathcal{M} \vDash \Phi}(\vec{x})$ = indicator function that returns 1 iff $\mathcal{M} \vDash \Phi$
  - Composition of system under test and property being tested

```
total = 0;
for (i = 1;i <= 10;i++)
    total += rand();
assert(total <= 8);
```



$$I_{\mathcal{M} \vDash \Phi}(\vec{x}) = \begin{cases} 1 & if \sum_{i=1}^{10} x_i > 8 \\ 0 & otherwise \end{cases}$$

We consider the property Φ to be a "failure" condition

Statistical Model Checking with Crude Monte-Carlo

The probability that condition Φ holds in model $\mathcal{M}$ when the input $\vec{x}$ is distributed according to joint pdf $f(\vec{x})$ is the expected value of that indictor function and can be calculated as:

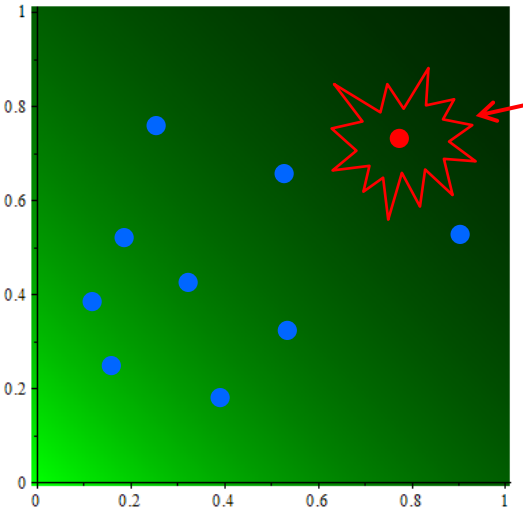
$$p = E[I_{\mathcal{M}\vDash\Phi}(\vec{x})] = \int I_{\mathcal{M}\vDash\Phi}(\vec{x}) f(\vec{x}) d\vec{x}$$

This can be estimated with Crude Monte-Carlo simulation as:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\vDash\Phi}(\vec{x}_i)$$

where each $\vec{x}_i$ is a sample vector drawn from $f(\vec{x})$. As N gets large, $\hat{p}$ will converge to p.

Estimated Failure Probability



# of samples in fault region

$$\hat{p} = \frac{1}{10} = 0.1$$

total # of samples

Relative Error: How large should N be?

Measure of accuracy for a prediction

Defined as ratio of standard deviation to mean. For a probability estimate, the estimated relative error is:

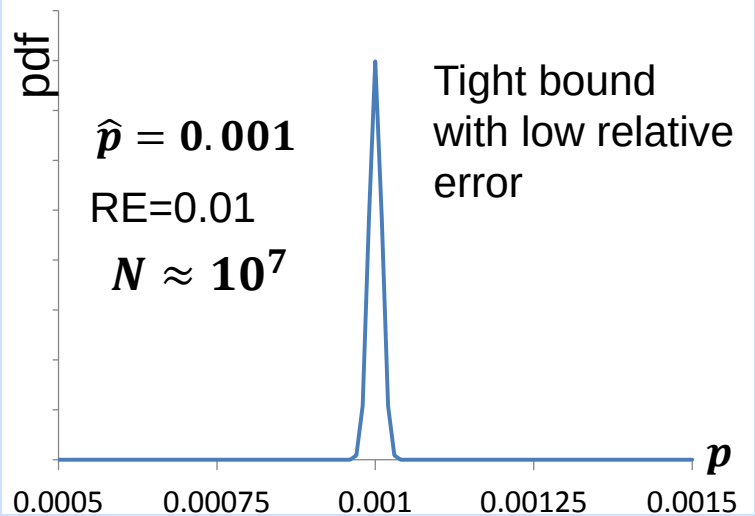
$$\widehat{RE} = \frac{\hat{\sigma}}{\hat{p}}$$

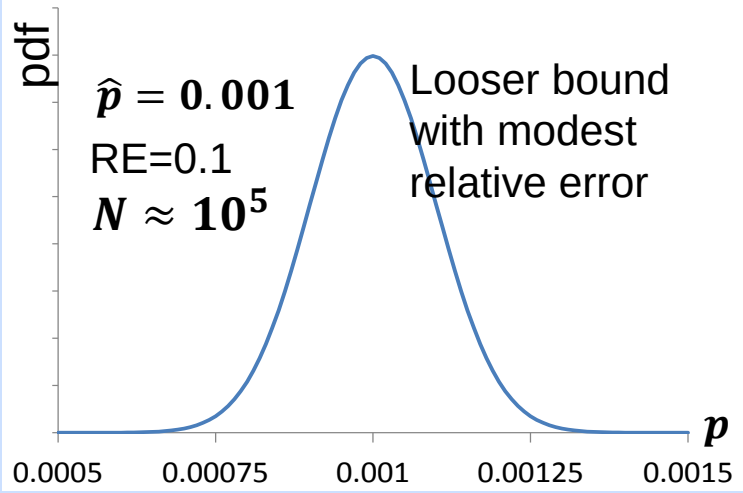
Number of samples to achieve a target relative error increases

- as target relative error decreases, or
- as estimated probability decreases

$$N \approx \frac{1}{p(RE)^2}$$

Distribution of actual p given estimated $\hat{p}$ and target relative error





Importance Sampling: Same RE with smaller N

Problem:

Estimating probabilities of rare events with low RE requires many samples

- To estimate failure probability of $p = 10^{-5}$ with relative error of 0.01 would require one billion simulation runs

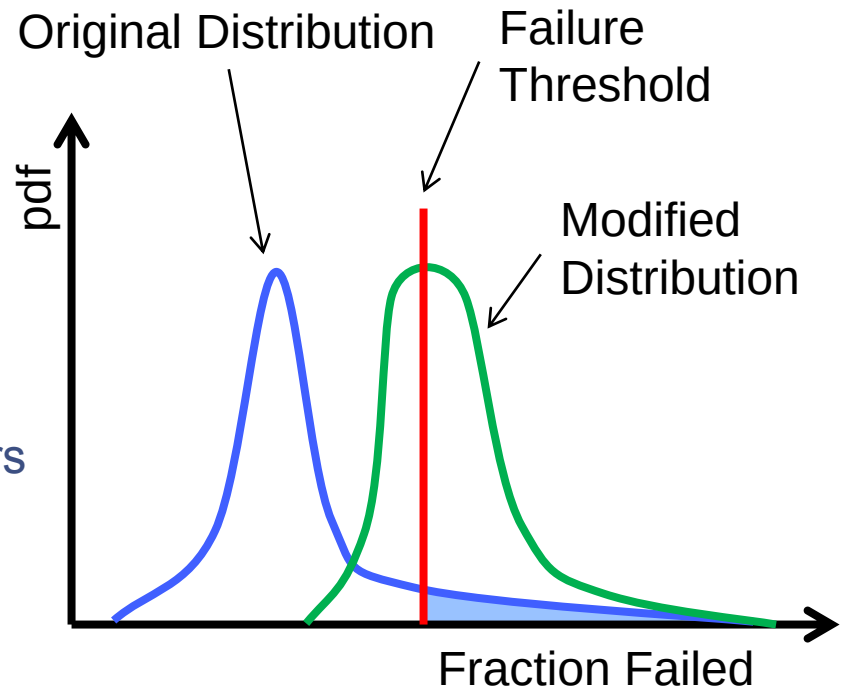
Number of samples required to estimate probability p event at relative error RE:

$$N \approx \frac{1}{p(RE)^2}$$

Solution:

Use Importance Sampling to sample "important" area of a distribution

- Sample with an "modified" distribution
  - Often by "tilting" the distribution parameters
- Map back to original distribution
- Can dramatically reduce number of experiments needed to verify "rare" events



Importance Sampling

Recall probability of failure is:

$$p = \int I_{\mathcal{M} \vDash \Phi}(\vec{x}) f(\vec{x}) dx$$

We can introduce an arbitrary density function $g(x)$ and rewrite as:

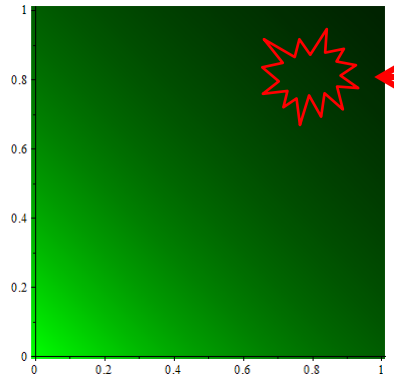
$$p = \int I_{\mathcal{M} \vDash \Phi}(\vec{x}) \frac{f(\vec{x})}{g(\vec{x})} g(\vec{x}) dx$$

Now if we define $W(\vec{x}) = \frac{f(\vec{x})}{g(\vec{x})}$ we get:

$$p = \int I_{\mathcal{M} \vDash \Phi}(\vec{x}) W(\vec{x}) g(\vec{x}) dx$$

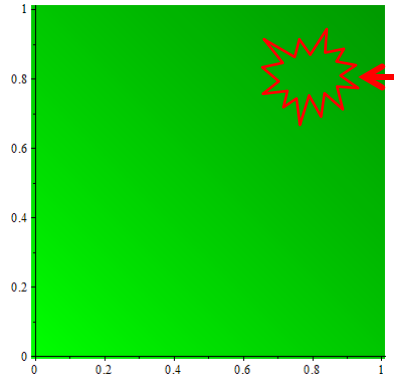
which is just the expected value of $I_{\mathcal{M} \vDash \Phi}(\vec{x}) W(\vec{x})$ sampled with $g(\vec{x})$.

$f(\vec{x})$



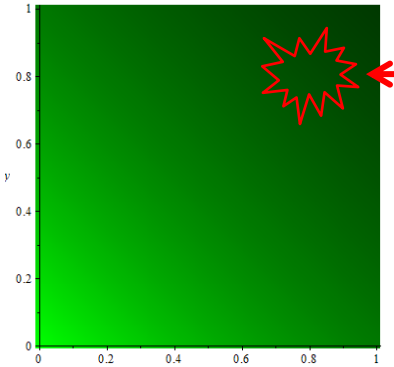
Fault region

$g(\vec{x})$



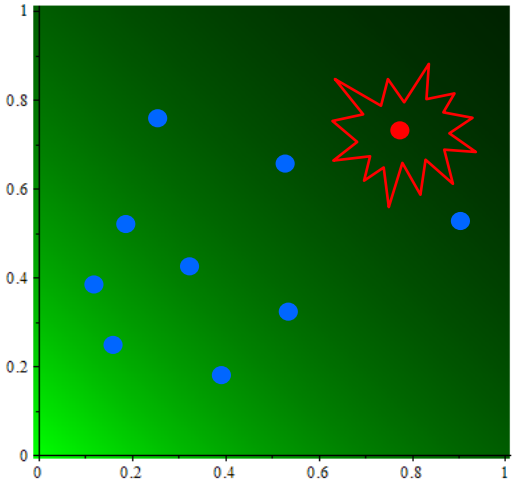
Increased visibility of fault region

$W(\vec{x})$



Weight function helps map back to original distr.

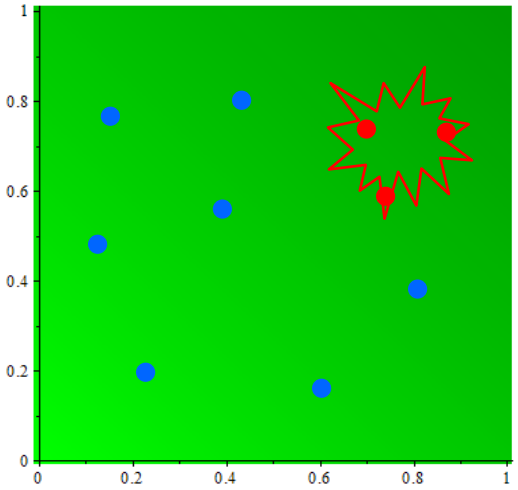
Importance Sampling



Basic SMC

- Indicator function $I(\vec{x}) = 1$ iff property holds for input $\vec{x}$.
- Relative Error $RE(\hat{p}) = \frac{\sqrt{var(\hat{p})}}{E[\hat{p}]}$ is measure of accuracy.
- Draw random samples from input distribution $f(\vec{x})$ until target Relative Error is met.
- Estimated probability that property holds is:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I(\vec{x}_i) = \frac{1}{10} = 0.1 \qquad RE(\hat{p}) = \frac{0.32}{0.1} = 3.2$$



SMC with Importance Sampling

- Modify input distribution to make rare properties more visible.
- Goal is variance reduction.
- Weighting function $W(\vec{x})$ maps solution to original problem.
- Reduced relative error with same number of samples.

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I(\vec{x}_i)W(\vec{x}_i) = \frac{0.2 + 0.5 + 0.3}{10} = 0.1$$

$$RE(\hat{p}) = \frac{0.18}{0.1} = 1.8$$

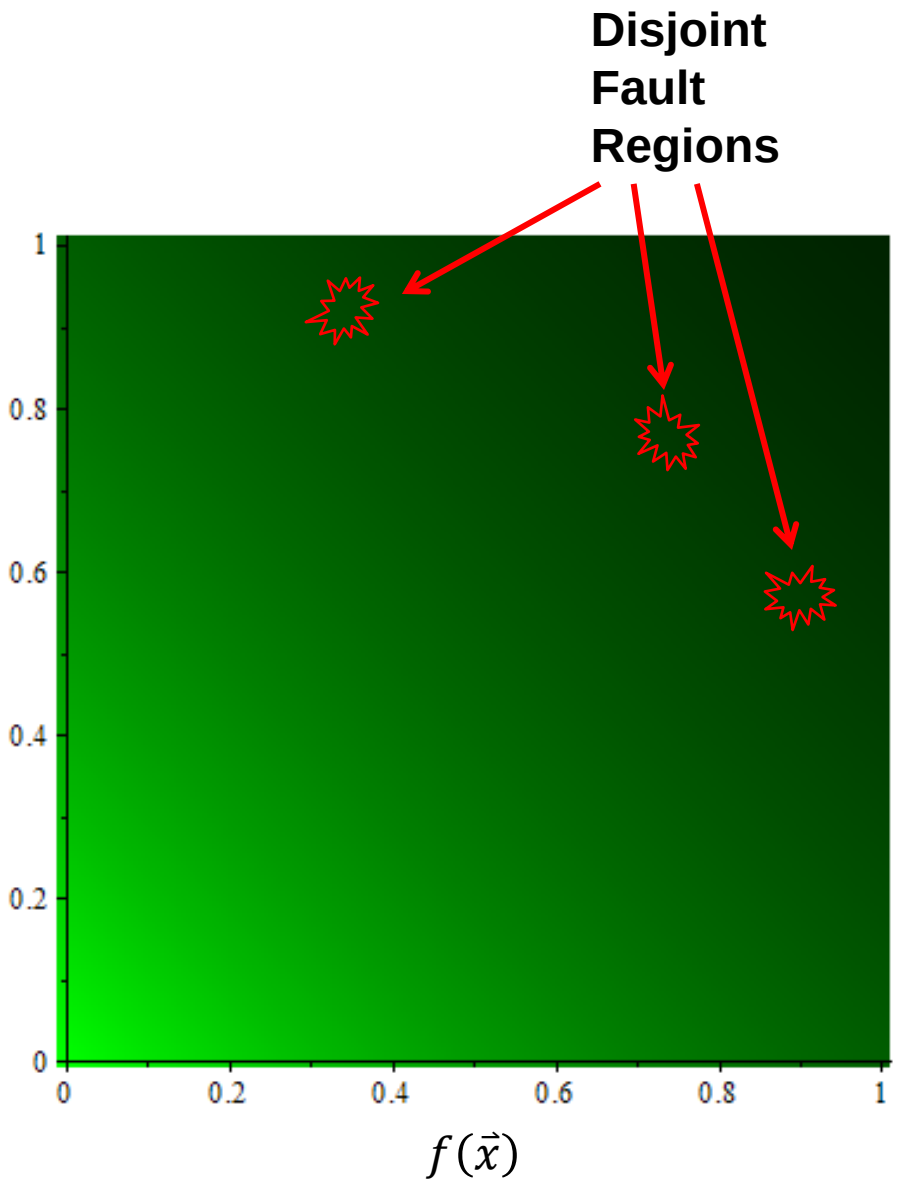
Disjoint Fault Regions

Problem

- Real software is complex with many paths.
- Inputs leading to fault may be disjoint in the input space.
- Importance sampling based on “tilting” distribution parameters may be limited in ability to cover fault regions.

Solution

- Need a method to do Importance Sampling that can better target the fault regions.



Optimal Importance Sampling (IS) Distribution

A well known result from Importance Sampling is that the optimal IS distribution (in the context of statistical model checking) is given by:

$$g(\vec{x}) = \frac{I(\vec{x})f(\vec{x})}{p}$$

where $I(\vec{x})$ is the indicator function for the system, $f(\vec{x})$ is the original input distribution, and p is the probability a sample falls in the indicator function. This results in a weight function of:

$$W(\vec{x}) = \frac{f(\vec{x})}{g(\vec{x})} = \frac{p}{I(\vec{x})}$$

And so the estimator becomes:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I(\vec{x})\frac{p}{I(\vec{x})} = p$$

This means no samples are needed, but this optimal distribution can only be found if your already know the solution.

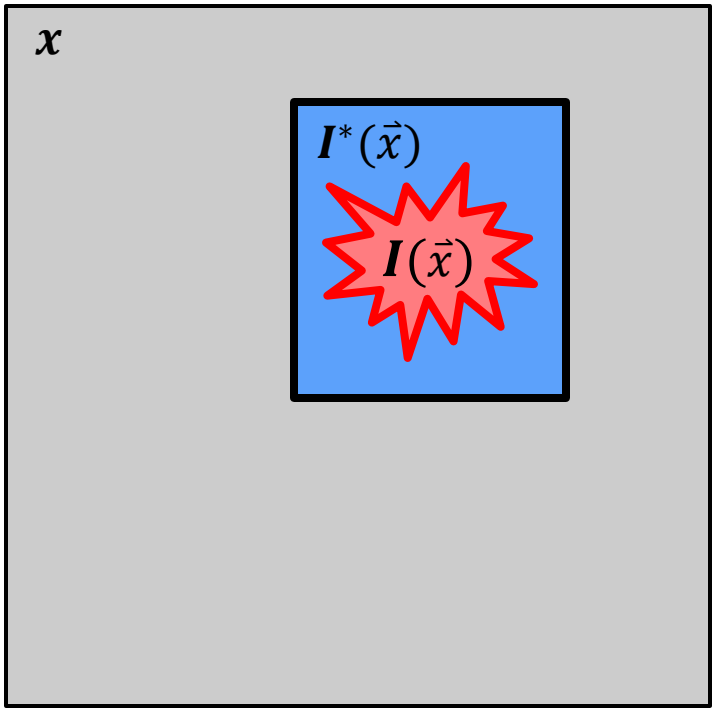
Semantic Approximation

Instead of using $I(\vec{x})$ directly, we can use abstraction on the original model (typically source code) to generate a semantic approximation $I^*(\vec{x})$.

We seek to find $I^*(\vec{x})$ such that:

- Any inputs that satisfy $I(\vec{x})$ also satisfy $I^*(\vec{x})$. That is $I(\vec{x}) \subseteq I^*(\vec{x})$
- $I^*(\vec{x})$ rejects as much of $\vec{x}$ as possible
- It is easy to generate vectors satisfying $I^*(\vec{x})$
- It is easy to calculate $p^* = E[I^*(\vec{x})]$

We refer to $I^*(\vec{x})$ as the *Abstract Indicator Function* (AIF).



Semantic Importance Sampling (1)

Using the same construction as the “optimal” IS distribution, we can construct an importance sampling distribution:

$$g^*(\vec{x}) = \frac{I^*(\vec{x})f(\vec{x})}{p^*}$$

where $I^*(\vec{x})$ is the AIF for the approximate system, $f(\vec{x})$ is the original input distribution, and p^* is the probability a sample falls in the indicator function for the approximate system. This results in a weight function of:

$$W^*(\vec{x}) = \frac{f(\vec{x})}{g^*(\vec{x})} = \frac{f(\vec{x})p^*}{I^*(\vec{x})f(\vec{x})} = \frac{p^*}{I^*(\vec{x})}$$

Semantic Importance Sampling (2)

Applying IS, we can write the following estimator for $p = \mathrm{E}[I(\vec{x})]$:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I(\vec{x}_i)W^*(\vec{x}_i) = \frac{1}{N}\sum_{i=1}^{N} I(\vec{x}_i)\frac{p^*}{I^*(\vec{x}_i)}$$

where the $\vec{x}$ are drawn from $\mathrm{g}^*(\vec{x})$. Since $g^*(\vec{x}) = \frac{I^*(\vec{x})f(\vec{x})}{p^*}$, for any x drawn from this distribution $I^*(\vec{x}) = 1$. This allows us to further reduce this estimator as:

$$\hat{p} = \frac{p^*}{N}\sum_{i=1}^{N} I(\vec{x}_i)$$

Osmosis SMC Tool

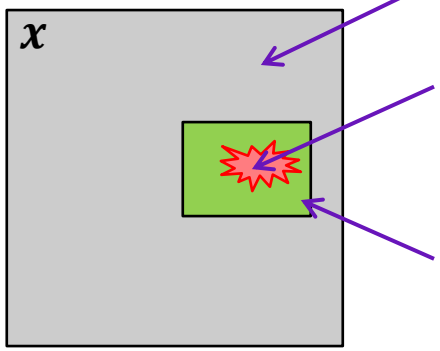
Osmosis is a tool for Statistical Model Checking (SMC) with Semantic Importance Sampling.

- Input model is written in subset of C.
- ASSERT() statements in model indicate conditions that must hold.
- Input probability distributions defined by the user.
- Osmosis returns the probability that at least one of the ASSERT() statements does not hold.
- Uses dReal[1] solver to build $I^*(\vec{x})$.
- Simulation halt condition based on:
  - Target relative error, or
  - Set number of simulations

[1] http://dreal.cs.cmu.edu/

Osmosis Main Algorithm

1. Generate approximation of fault region

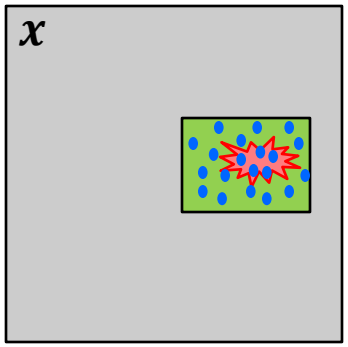


Input space

$I(\vec{x})$: Indicator Function defines fault region

$I^*(\vec{x})$: Abstract Indicator Function defines over-approximation

2. Conduct SMC and calc. failure prob.



a) $\hat{p}_{raw} = \frac{5}{20}$ ← 5: # in Fault; 20: Total #

b) $p^* = \frac{6}{64}$ Fraction of input $I^*(\vec{x})$ covers

c) Failure prob. estimate is product of two values

$$\hat{p} = \hat{p}_{raw} p^* = 0.0023$$

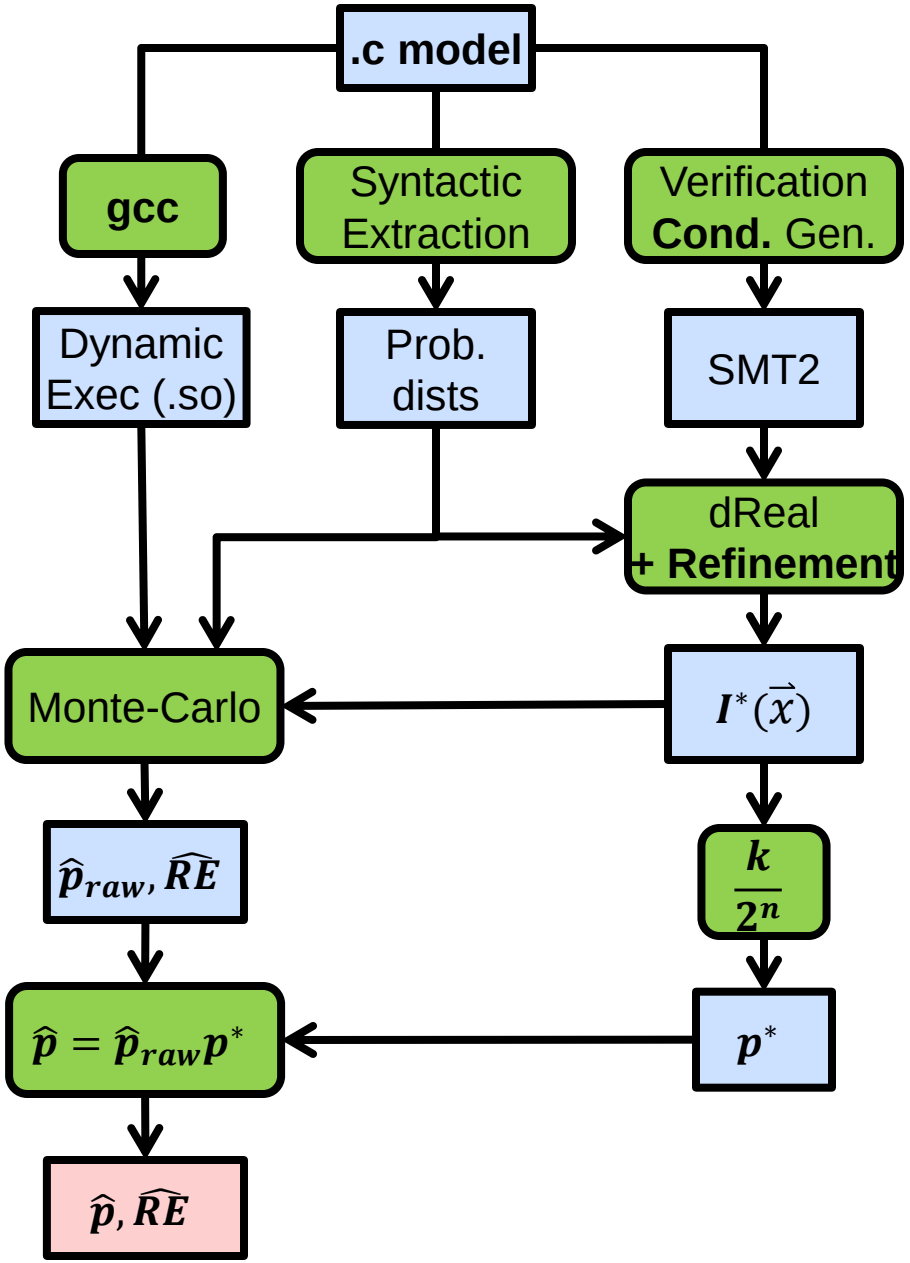
Osmosis Structure

SIS Pre-processing

- C model analyzed to generate SMT2 and input probabilities.
- dReal used to generate $I^*(\vec{x})$.
- $I^*(\vec{x})$ used to calculate p^*.

Statistical Model Checking

- C model is compiled with gcc to dynamically loadable executable (.so file).
- Dynamic executable loaded into osmosis.
- Random inputs generated with $I^*(\vec{x})$ and applied to dynamically loaded function.
- Result is scaled by p^* to obtain final probability estimate.



Osmosis Input Example

Input is subset of C

- No arrays, or pointers
- Supports: `if/while/for`
  - With loop limits

Probabilistic inputs

- Use `//@dist` declaration for each probabilistic input.
  - Uniform, exponential, normal
- Access in C model using `INPUT_D("name")`.

Assertions

- Use `ASSERT()` to indicate conditions that should be true.
- Osmosis will compute probability that at least one `ASSERT()` has failed.

"Fault Box" example

```
#include "osmosis_client.h"

//@dist a=uniform(min=0,max=5)
//@dist b=normal(mean=3,std=1,min=0,max=5)
void simple()
{
  double a = INPUT_D("a");
  double b = INPUT_D("b");
  double c = a + b;
  double d = (a - b)/2.0;

  ASSERT(sin(c)*cos(d) < 0.995);
}
```

SMT2 Generation

C Input translated to SMT2 for use by dReal.

- Assignments translated to assertions.
  - Assignments to same variable increment "generation number"
- Loops unrolled and translated to conditionals.
- Implication assertions used for assignments inside conditionals.
- Input "cube" represented by bounding assertions on each input variable.
  - Varied over multiple dReal calls.
- ASSERT() condition from C is negated in SMT2, since we wish to search for failures.
  - A dReal "SAT" means failure of ASSERT() in C model.

```
(set-logic QF_NRA)
(declare-fun a () Real)
(declare-fun b () Real)
(declare-fun a_1 () Real)
(declare-fun b_1 () Real)
(declare-fun c_1 () Real)
(declare-fun d_1 () Real)
(assert (>= a 0))
(assert (<= a 5))
(assert (>= b 0))
(assert (<= b 5))
(assert (= a_1 a))
(assert (= b_1 b))
(assert (= c_1 (+ a_1 b_1)))
(assert (= d_1 (/ (- a_1 b_1) 2.0)))
(assert (not (< (* (sin c_1)
                   (cos d_1))
                0.995)))
(check-sat)
(exit)
```

Input Cube

ASSERT()

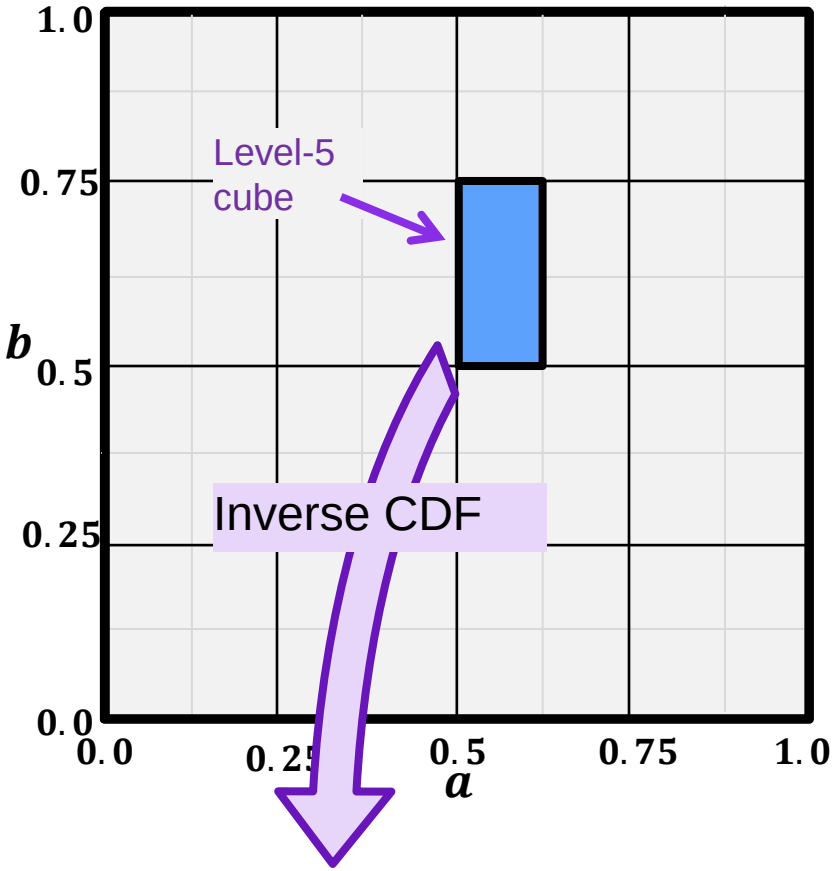
Input Cubes

Cubes

- Cubes defined on CDF (Cumulative Distribution Function) of each input variable.
- Sub-cubes formed by splitting input domain into equal probability halves around an input variable.
- Cube values can be mapped to input values using inverse CDF.

Level

- Level of a cube is number of splits.
- Full input domain is “level-0”.
- All cubes at same level have equal probability of containing input drawn from input distribution.
- There are 2^n cubes at level-n.




```
(assert (>= a 2.5))
(assert (<= a 3.125))
(assert (>= b 1.20013))
(assert (<= b 1.80517))
```

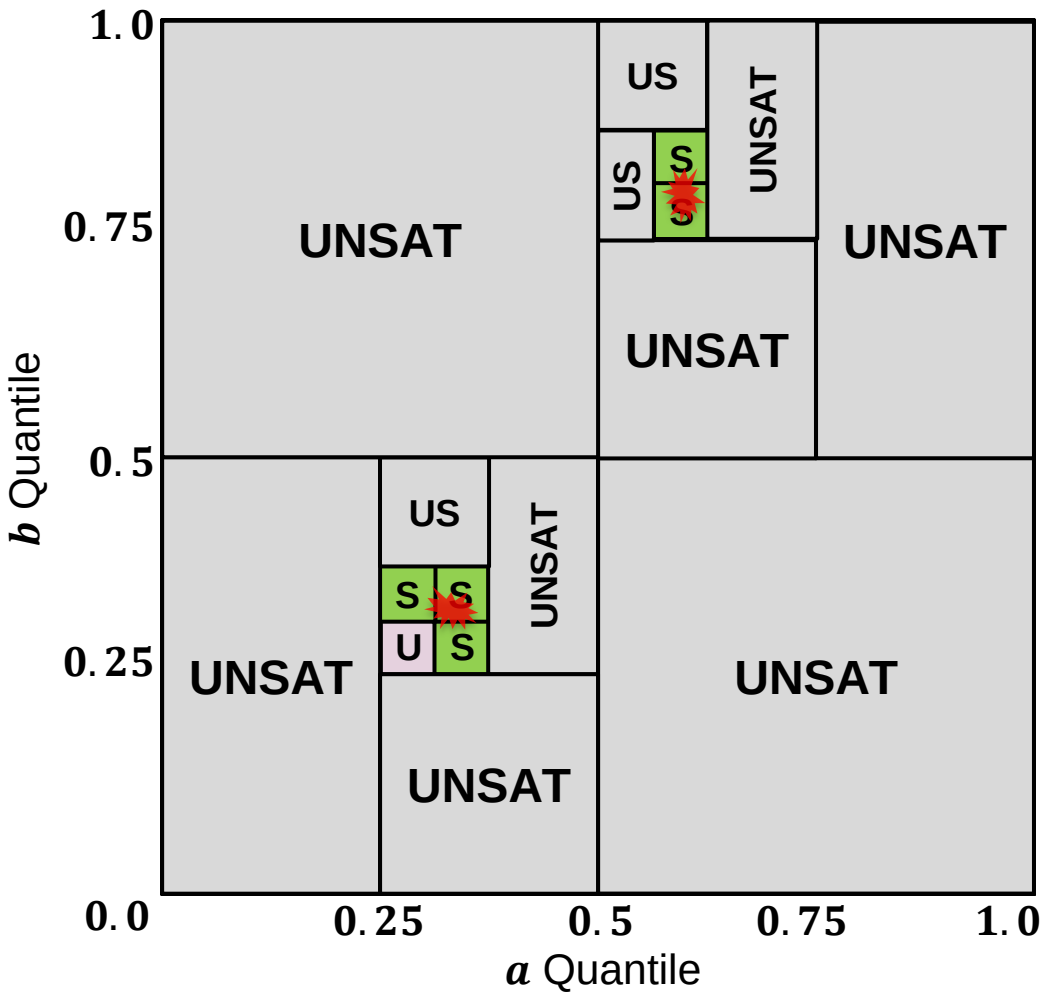
$I^*(\vec{x})$ Generation Algorithm

Algorithm:

1. Set the current “cube” as the full range of all inputs.
2. Apply dReal to the current cube.
3. If the result is “SAT”, split cube into two equal probability cubes on one variable, and recursively apply at Step 2.

Considerations:

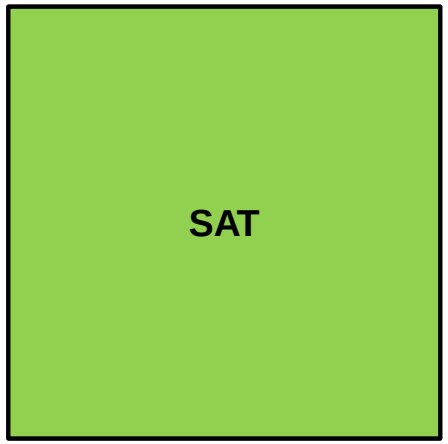
- Maximum recursion depth is tunable parameter.
- By default, we use “round robin” for variable splits.



$I^*(\vec{x})$ Generation Algorithm – Optimization 1

Some calls to dReal are redundant and can be eliminated.

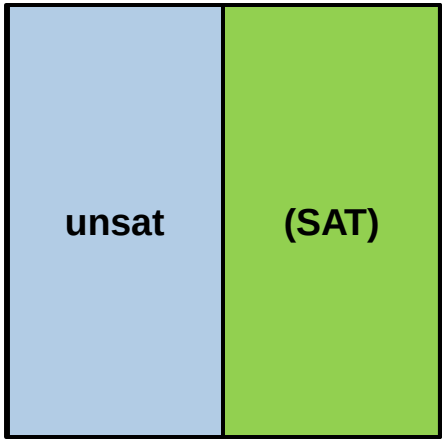
- If recursive call on first sub-cube returns the empty set (i.e., it is UNSAT), the second recursive call must be SAT since the parent is SAT.
- In this case, we pass a flag indicating dReal call can be skipped.
- Can significantly reduce the number of calls to dReal needed.
- This optimization cannot be used when dReal fails on a cube.



1) dReal returns a SAT result on parent cube.

3) 2nd child cube must be SAT, so we can skip dReal test.

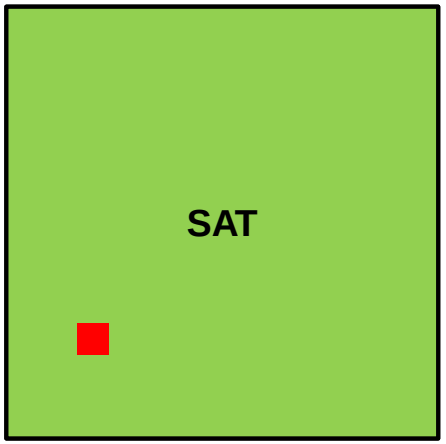
2) dReal returns unsat on first of the two child cubes.



$I^*(\vec{x})$ Generation Algorithm – Optimization 2

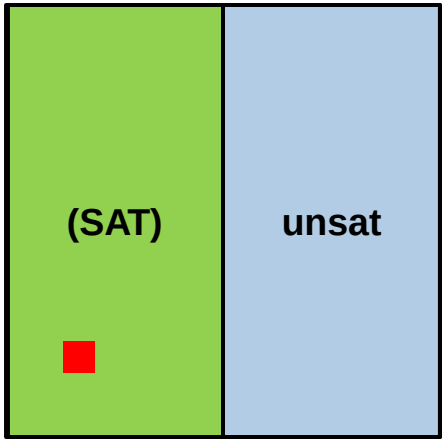
Counter-Example returned by dReal can be used.

- On SAT outcome, dReal returns a box in which the counter-example is contained.
- If this box is completely contained in a child cube, then that child cube must also be SAT, and no dReal test is required.
- CAVEAT
  - if the counter-example cube straddles the boundary between the child cubes, we cannot use this optimization.
  - Optimization 1 may still apply in this case.



1) dReal returns a SAT result on parent cube, including counter example.

3) Must use dReal to test other child.



2) Skip dReal test if counter example contained in child.

Calculation of p*

Since

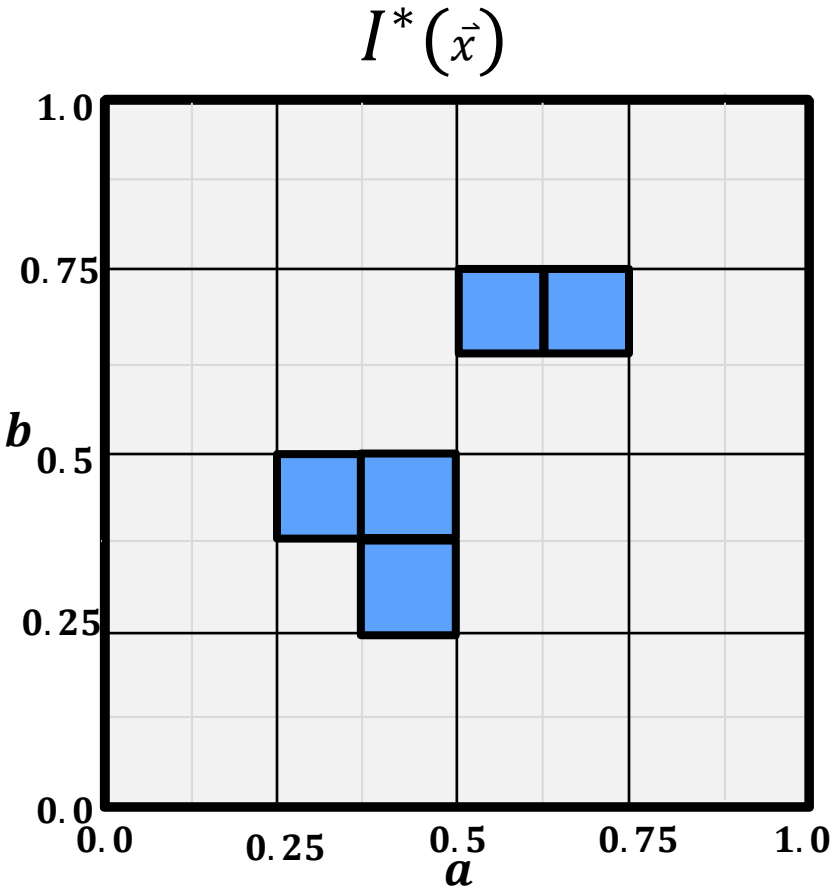
- All cubes at a given level have same probability of containing a sample drawn from the input distribution $f(\vec{x})$.
- Our $I^*(\vec{x})$ generation algorithm returns a set of cubes at a specified level.

Then

- The probability an input $\vec{x}$ is in $I^*(\vec{x})$ is:

$$p^* = \frac{k}{2^n}$$

where k is the number of cubes in our representation of $I^*(\vec{x})$, and n is the maximum number of levels we used.



$$p^* = \frac{5}{2^6} = 0.078125$$

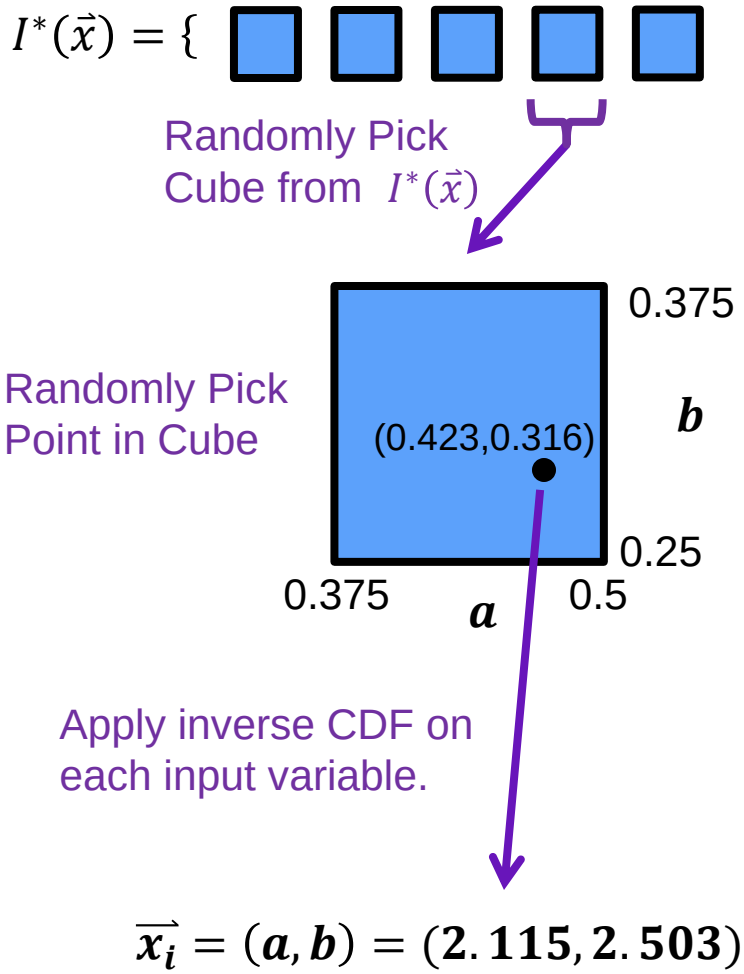
Random Input Generation

Cube Selection

- Since all cubes cover equal probability, we pick a cube at random with uniform probability.

Point Selection

- Within a cube, we pick a point at random with uniform probability on each dimension.
- We apply the inverse CDF of each input variable to obtain the values for each input variable.



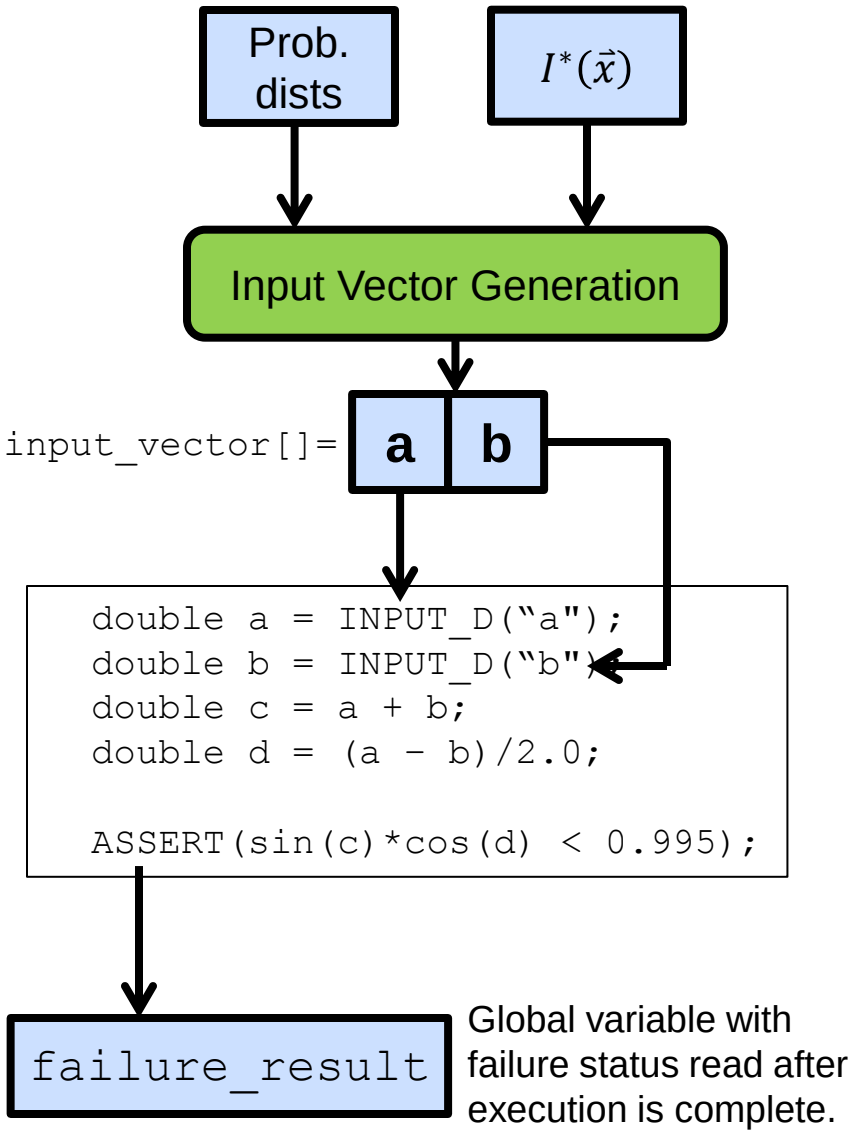
Monte-Carlo Simulation

Dynamic Executable Generation

- Use gcc to compile input .c file into a dynamic executable (.so file)
- Use dlopen() to load executable into Osmosis.

Monte-Carlo Simulation

- Generate random inputs using $I^*(\vec{x})$ and input distributions and place into an array.
- C code for model directly executed.
- INPUT_D("name") function calls in source model extract values from that array.
- ASSERT() macro sets global variable if assertion fails.



Final Probability Estimation

Probability estimate $\hat{p}$ of $p = \mathrm{E}[I(\vec{x})]$ is product of:

- Raw probability $\hat{p}_{raw}$ from Monte-Carlo simulation using AIF.
- Scale factor p^* calculated from cube set.

$$\hat{p} = p^* \hat{p}_{raw}$$

Relative error is preserved. i.e, $RE(\hat{p}) = RE(\hat{p}_{raw})$

- Preserved because scale factor p^* applies equally to mean and standard deviation of $\hat{p}_{raw}$.
- Implies number of samples need to estimate $\hat{p}$ to a specific relative error is same a number needed to estimate $\hat{p}_{raw}$.

Example: Air Hockey Problem

Air Hockey Problem

- Table with a moving puck and a fixed target.
- Puck rebounds without friction.

Inputs

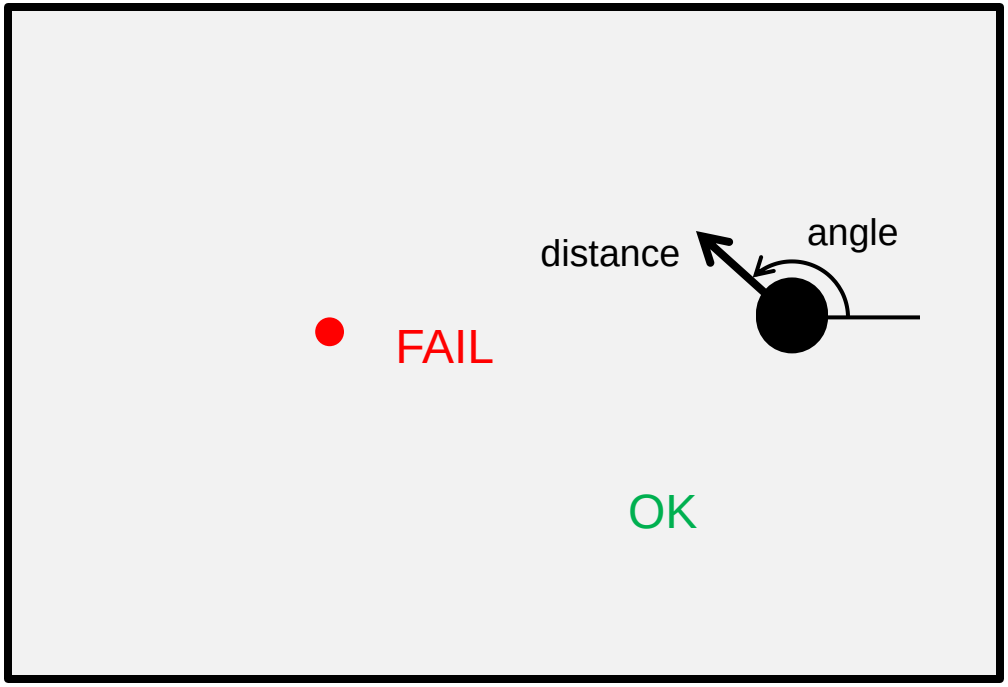
- Angle – Initial angle at which puck is hit.
- Distance – Total distance of travel for puck.

Failure Condition

- Puck stops on target (red dot).

Challenges

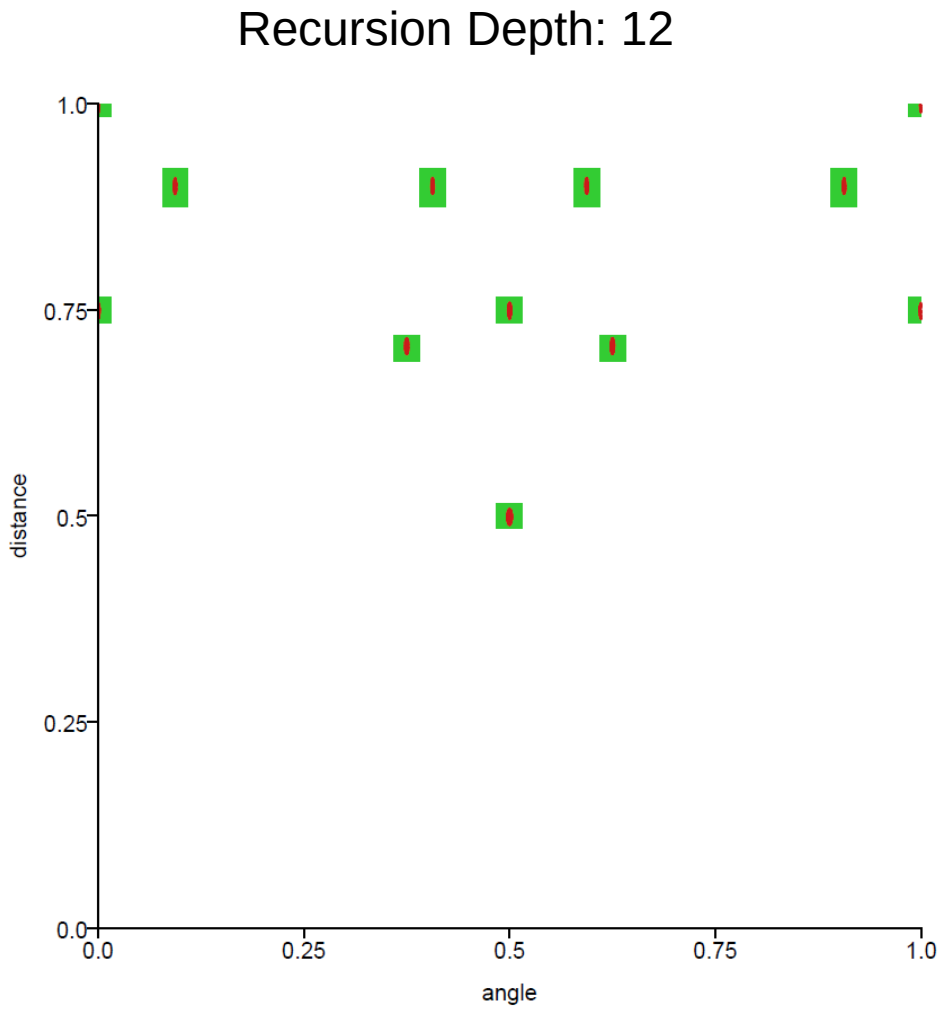
- Multiple failure areas in input space.
- Complex structure of failure area.



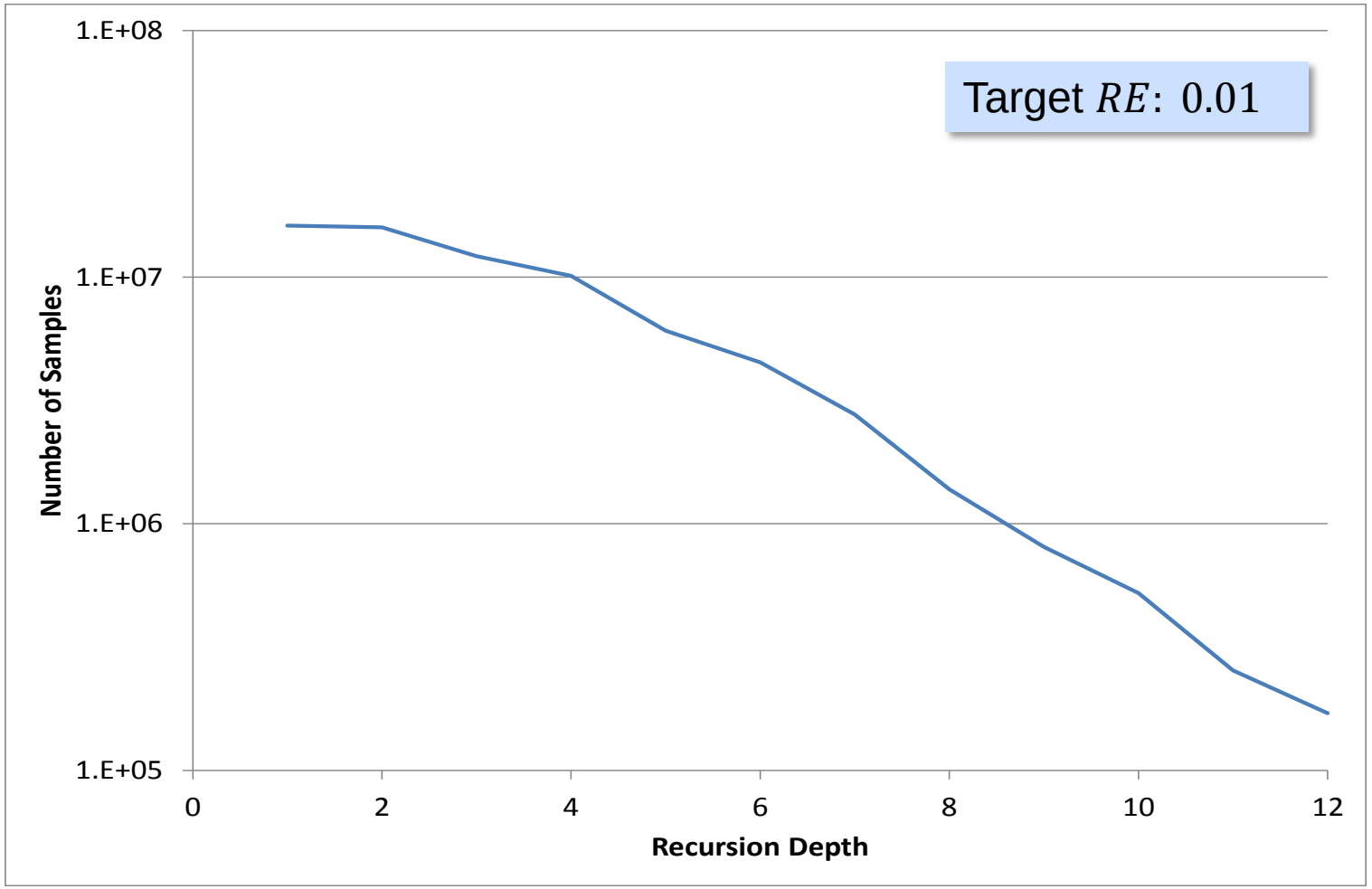
Fault Map for Air Hockey Problem

Fault map shows area of input space where faults are located.

- Plotted in CDF space.
- **Green** area indicates input space included in $I^*(x)$.
- **Red** area indicates input space include in $I(x)$.

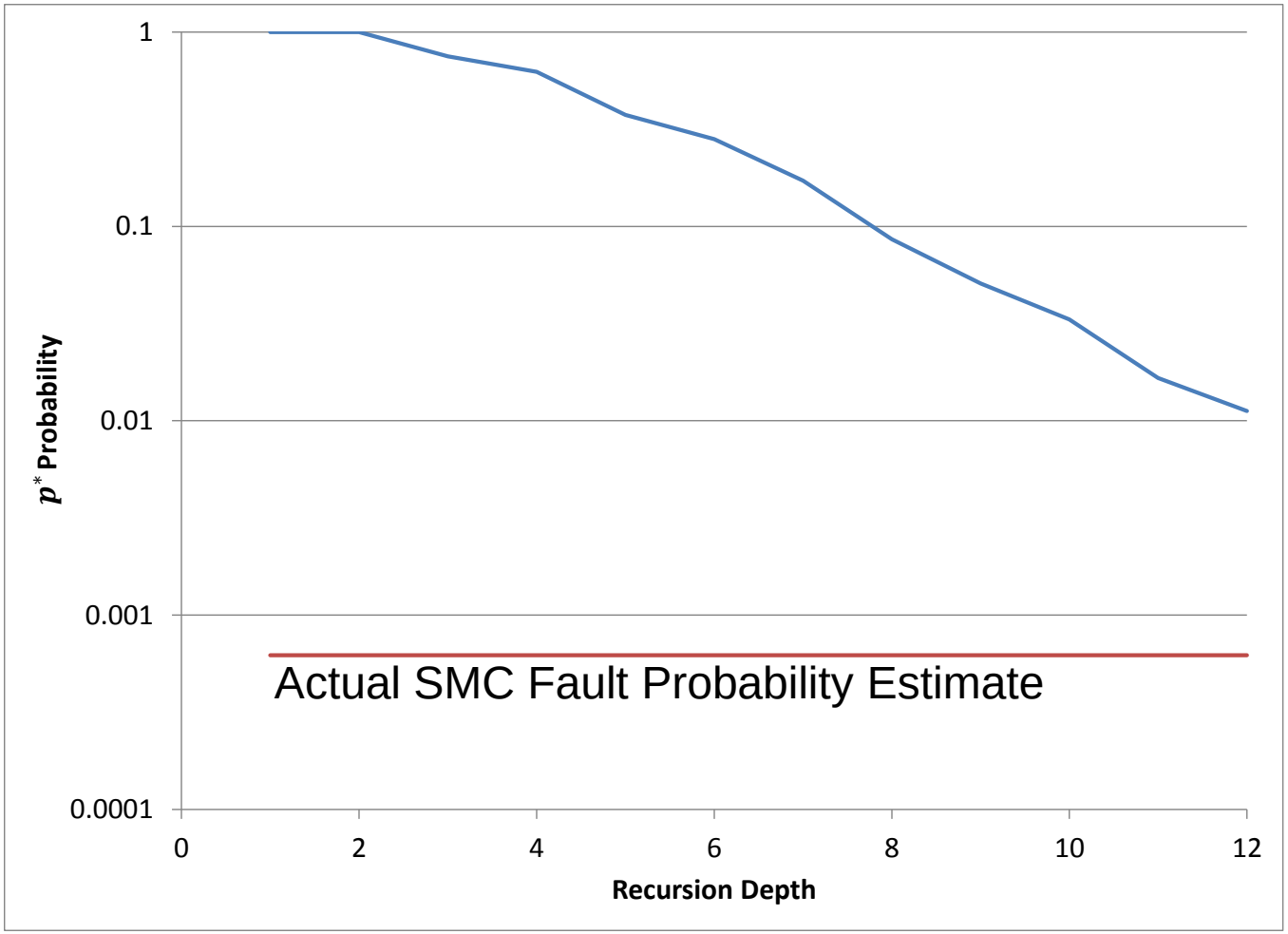


Sample Size vs Recursion Depth (Air Hockey)



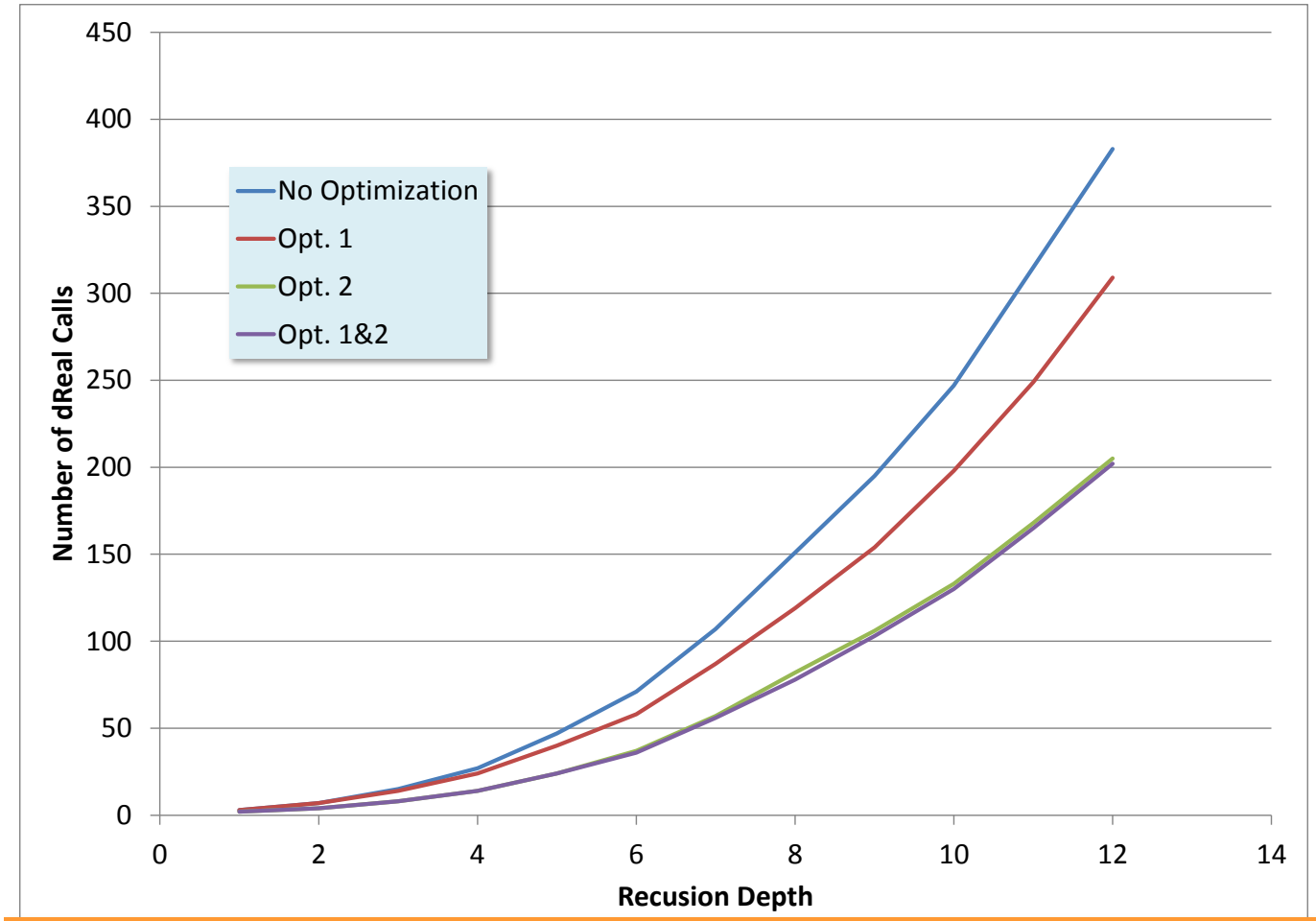
Simulation effort with SIS decreases exponentially with recursion depth.

p^* vs Recursion Depth (Air Hockey)



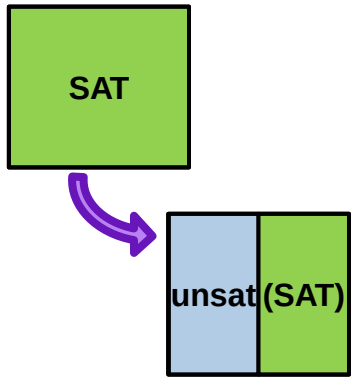
Upper-bound p^* becomes more accurate as recursion depth increases.

Effect of SIS Optimizations



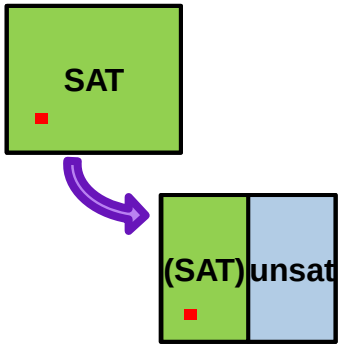
Optimization 1

No call to dReal if first child call is unsat.



Optimization 2

Use counter-example from parent to avoid dReal calls on children.



Optimization 2 results in greatest benefit with factor of two reduction in number of calls to dReal. Small additional benefit by combining both methods.

Conclusion

Semantic Importance Sampling

- Create approximation of fault region using abstraction.
- Create an alternate input distribution for importance sampling.
- Level of approximation (recursion depth) is user tunable.
- Can reduce SMC sample size by orders of magnitude.

Osmosis tool

- Applies semantic importance sampling on a C-like specification.
- Uses the dReal SMT solver to build approximate fault region model.
- Can be applied when there are multiple fault regions.
- Optimization techniques can nearly halve number of dReal tests required.