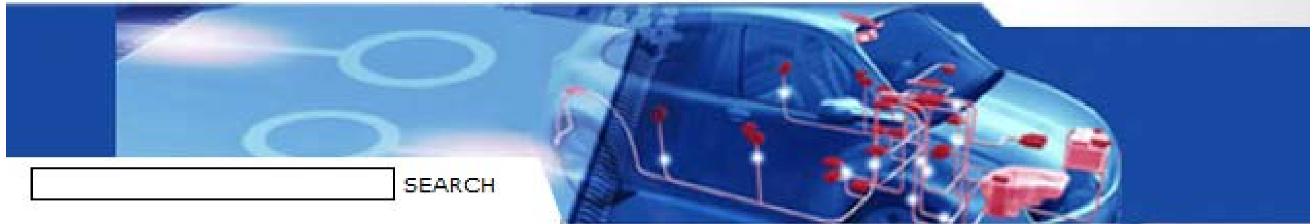
Random Distributions for Property-Based Testing

John Hughes

CHALMERS QuviQ



COM Services

Com

PduR
NmIf
IpduM

LIN
LinNm
LinSm
LinIf
LinTrcv
Lin

CAN
CanNm
CanSm
CanTp
CanIf

FlexRay
FrNm
FrSm
FrTp
FrIf

Ethernet
EthSa
EthNm
EthSm
EthIf

Diagnostic cluster
Dem
DCM
FiM

3,000 pages of specifications

20,000 lines of properties

1,000,000 LOC, 6 suppliers

200 problems

100 problems in the standard

DEMO

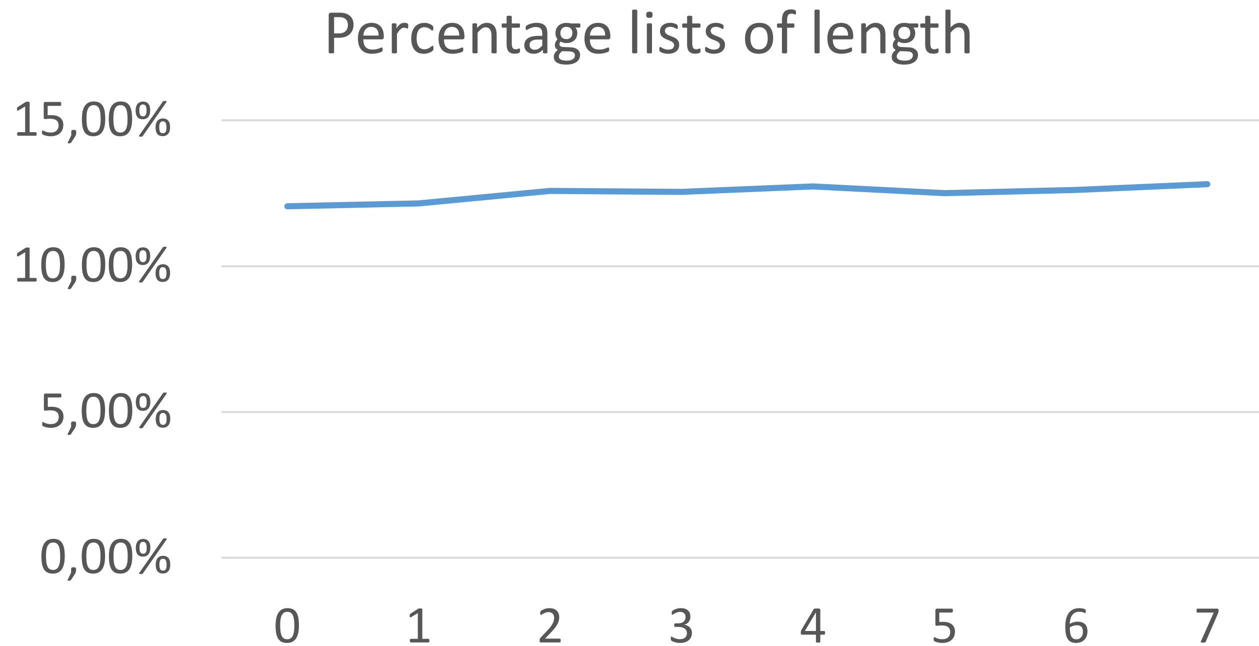
Uniform distributions?

- Lists of 32-bit integers, with 0—7 elements
- For every list of length 6, there are 2^{32} lists of length 7!
- With uniform distribution:

$$P(\text{length} < 7) \approx 2^{-32}$$

- P(prop_zip() fails) ~= 2^{-31}

Actual distribution



- Uniform distribution *of list lengths*, not of test data

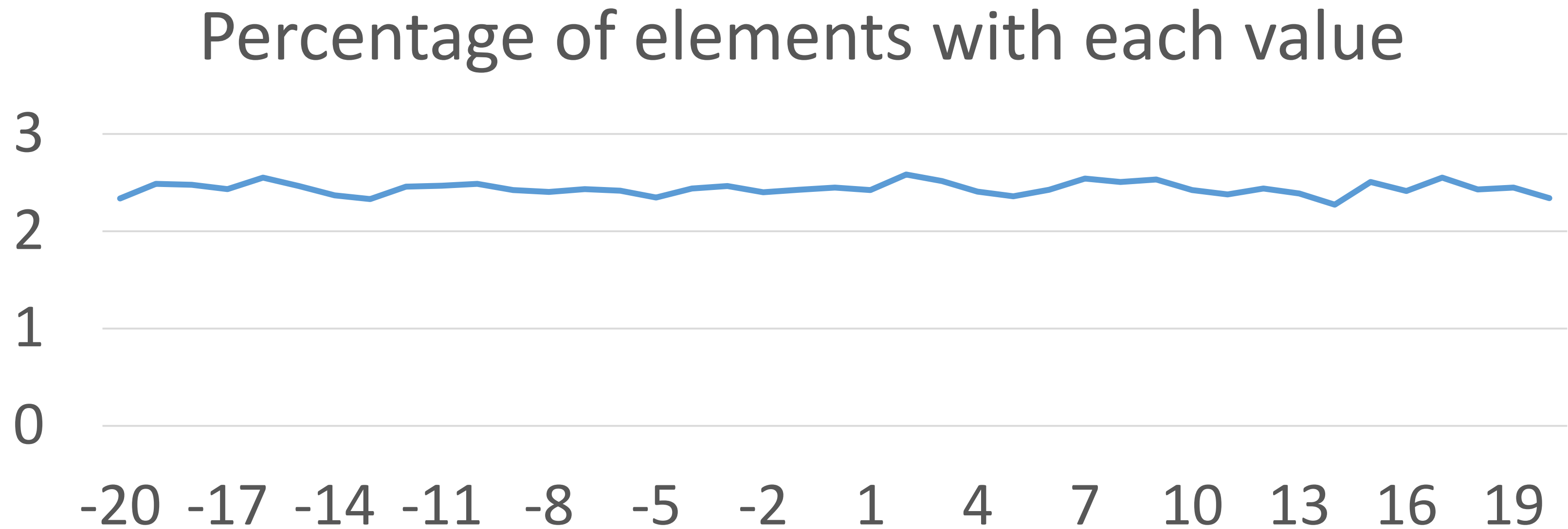
Uniform distribution of elements?

P(list has duplicate elements | length==7)

$$\sim= 21 \times 2^{-32}$$

- P(prop_delete() fails) ~ 21×2^{-64}

Actual distribution of elements



- Uniform distribution over a *limited range*
- P(a list has duplicate elements) ~= 14%

Two general ideas

- Uniform distribution of *a property of the data*, not the data itself

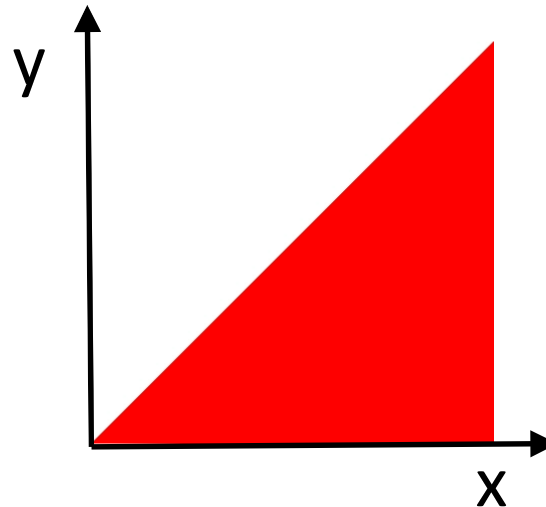
- Choose from a small set, so collisions become more likely

Satisfying invariants

- 0 <= x <= 10, 0 <= y <= 10, y <= x

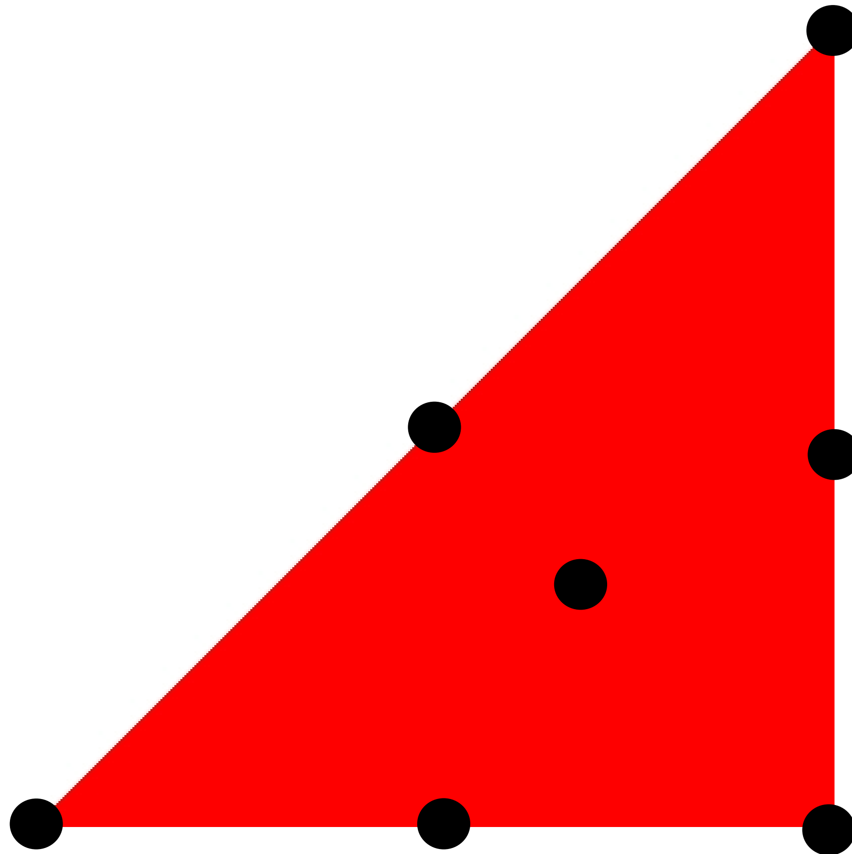
- Generate and filter:

```
?SUCHTHAT({X,Y},
          {choose(1,10),choose(1,10)},
          Y =< X)
```

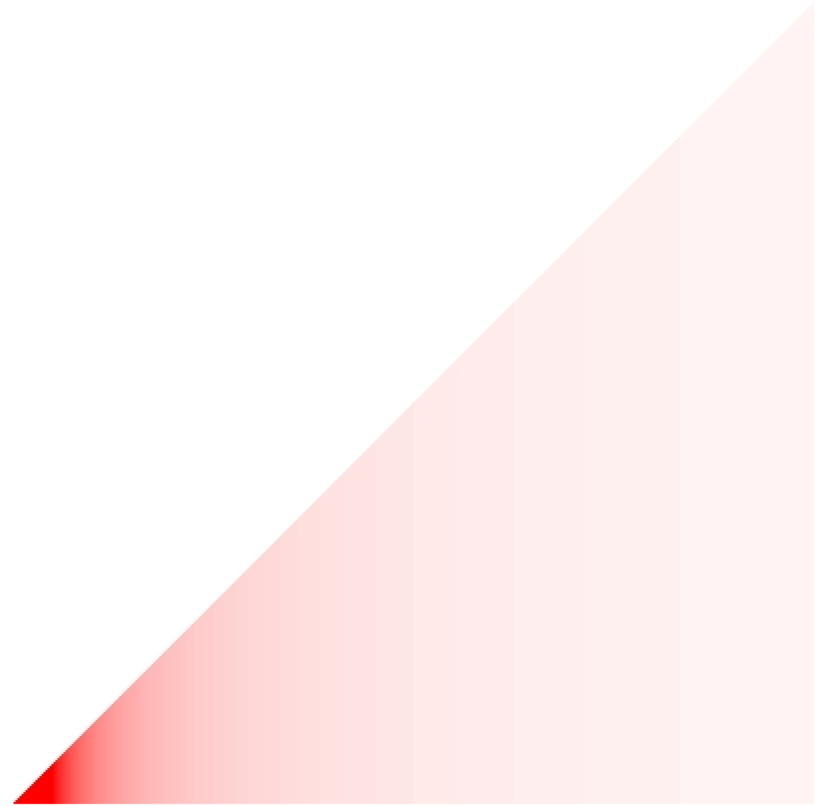


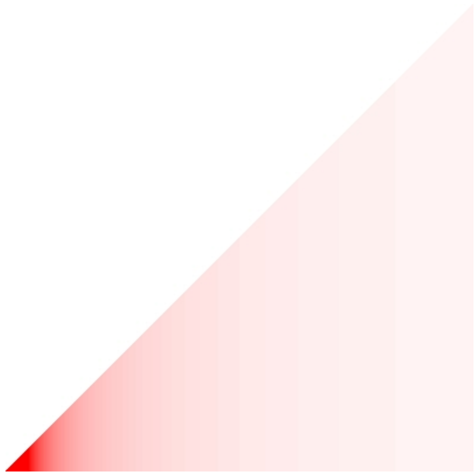
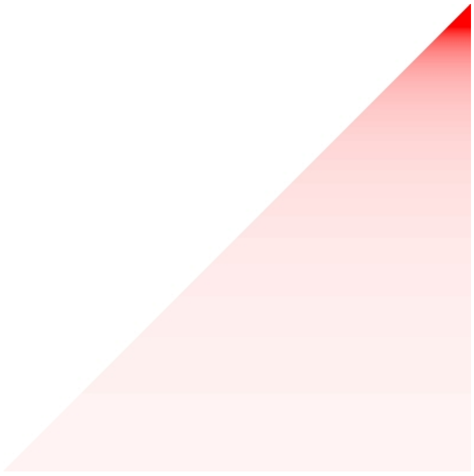
Manual selection of test cases

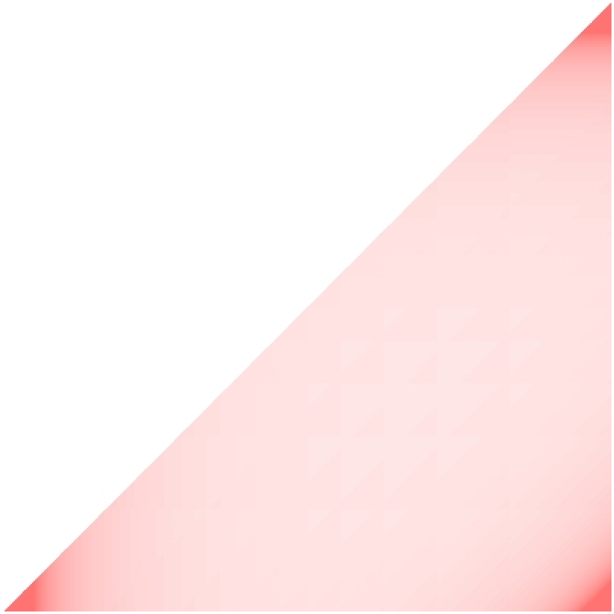
- Test extremal values plus a few in the middle



```
?LET(X,choose(1,10),
        {X,choose(1,X)})
```





Two stage generation

- x <= y?
  - Generate x, then generate y

- Sorted lists?
  - Generate a list, then sort it

- Test cases for prop_delete()?
  - Generate a list, then *choose an element from it*

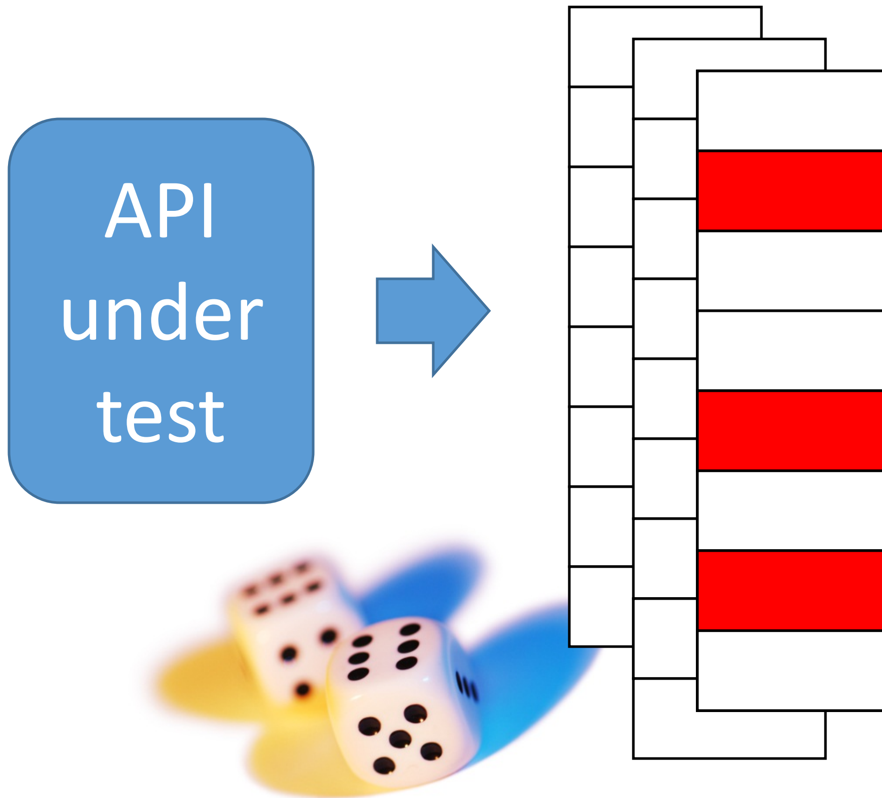
Smartening prop_delete()

```
prop_delete() ->
   ?FORALL({X,Xs},
           {int(),list(int())},
           smart(),
           not member(X,delete(X,Xs))).

smart() ->
  frequency([{10,{int(),list(int())}},
             {90,?LET(Xs,non_empty(list(int())),
                      {elements(Xs),Xs})}]).
```

Fails in **<0.4%** of cases

Fails in **>6%** of cases

Testing stateful APIs



A minimal failing example

The Erlang process registry

Name	Pid
a	…
b	…
c	…

- register(Name,Pid)
  - —add a process to the registry
- whereis(Name)
  - —return corresponding Pid
- unregister(Name)
  - —remove Name from registry

Choose names from a small set to increase collision probabilities

- spawn()
  - —create a process to be registered
- kill(Pid)
  - —kill a process (dead processes are not allowed in the registry)

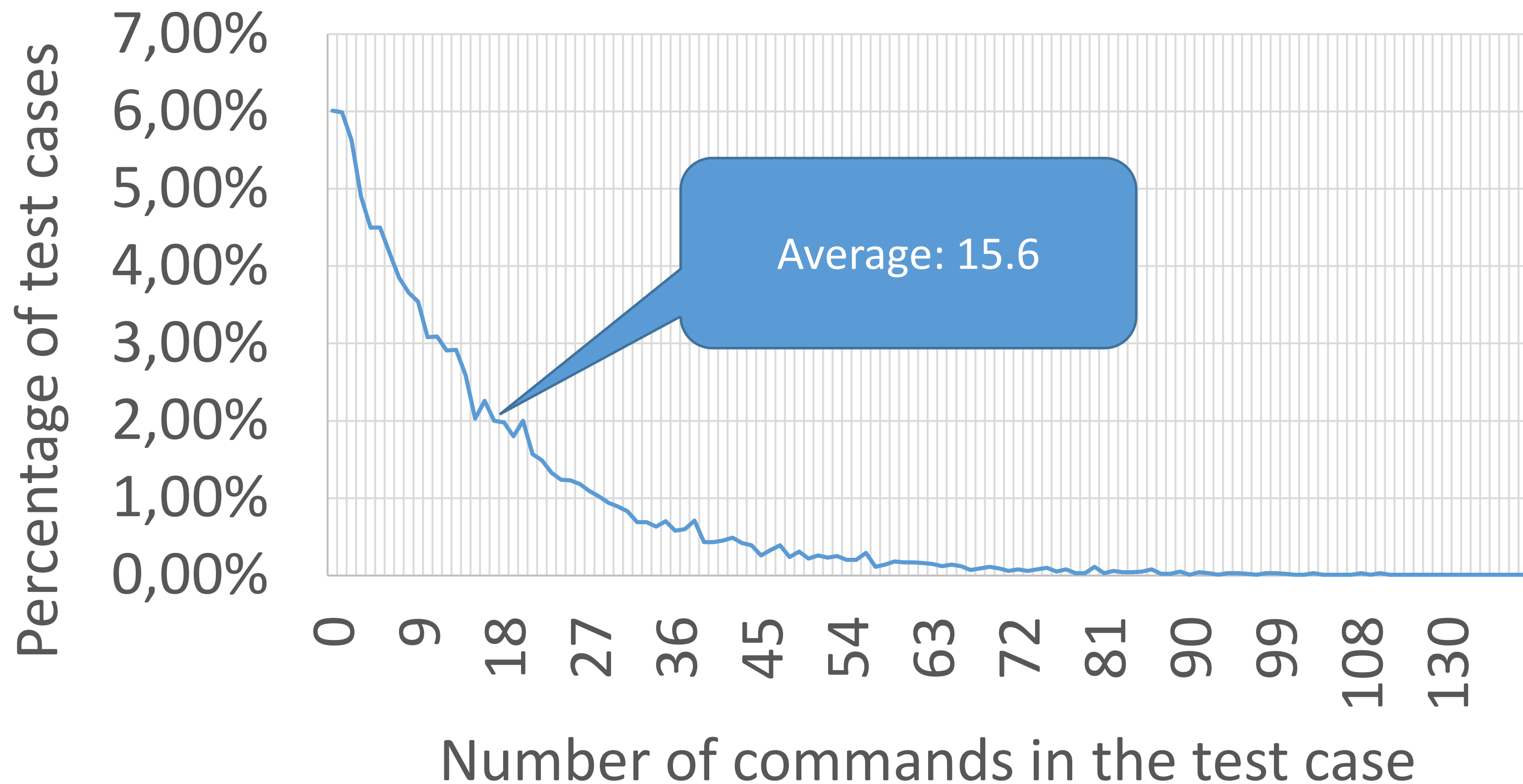
Choose Pids from a small set

A sample ”bug”

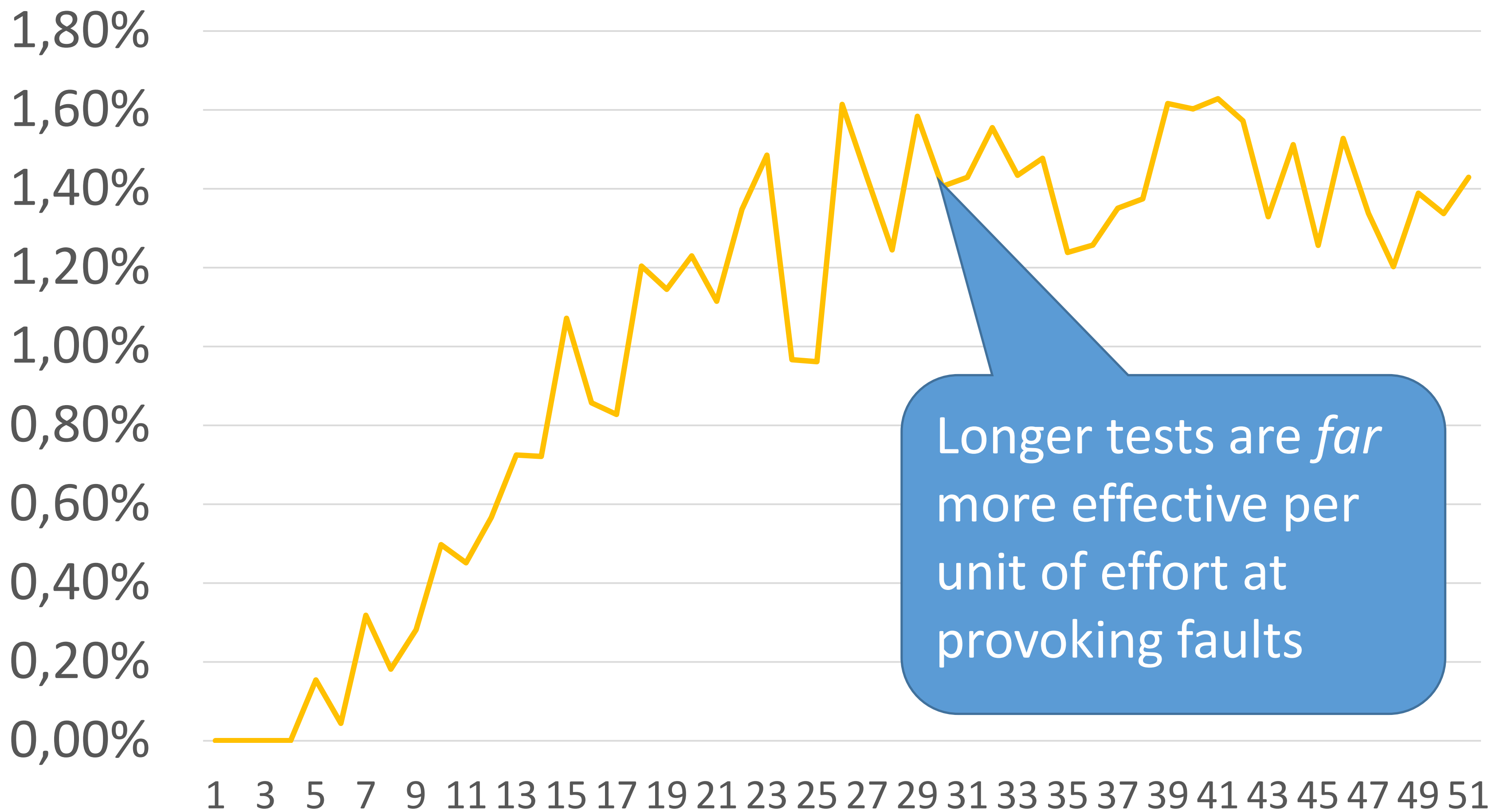
```
spawn()                     -> <0.117.0>
register(a, <0.117.0>)      -> true
kill(<0.117.0>)             -> ok
whereis(a)                  -> undefined

Reason:
  Post-condition failed:
  undefined /= <0.117.0>
```

- A bug in the properties: forgot that *killing* a process must *remove* it from the registry

How long are generated tests?



How often do tests fail?

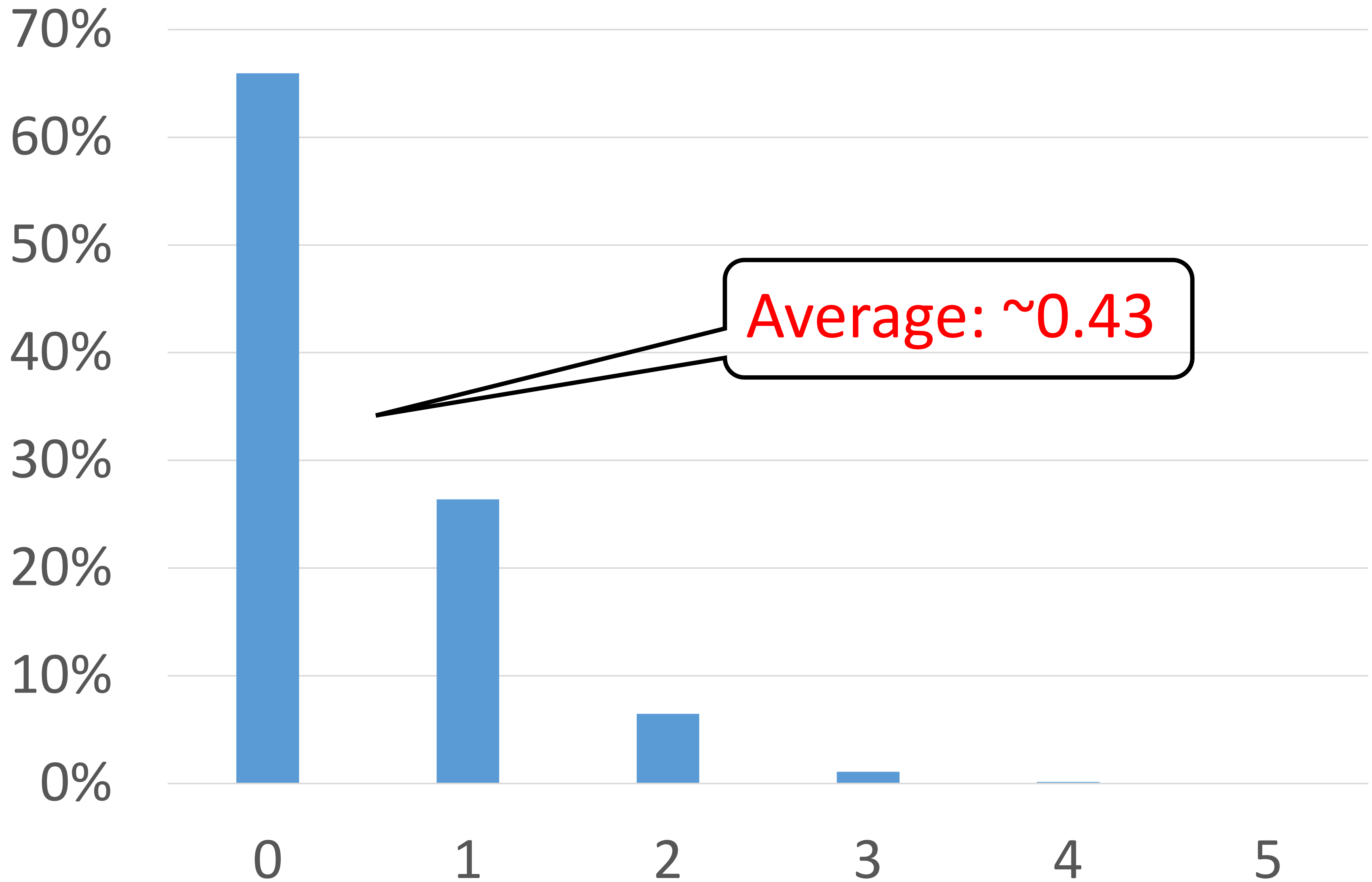

Probability of failure per command
1,80%
1,60%
1,40%
1,20%
1,00%
0,80%
0,60%
0,40%
0,20%
0,00%
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
Longer tests are *far* more effective per unit of effort at provoking faults

The Erlang process registry

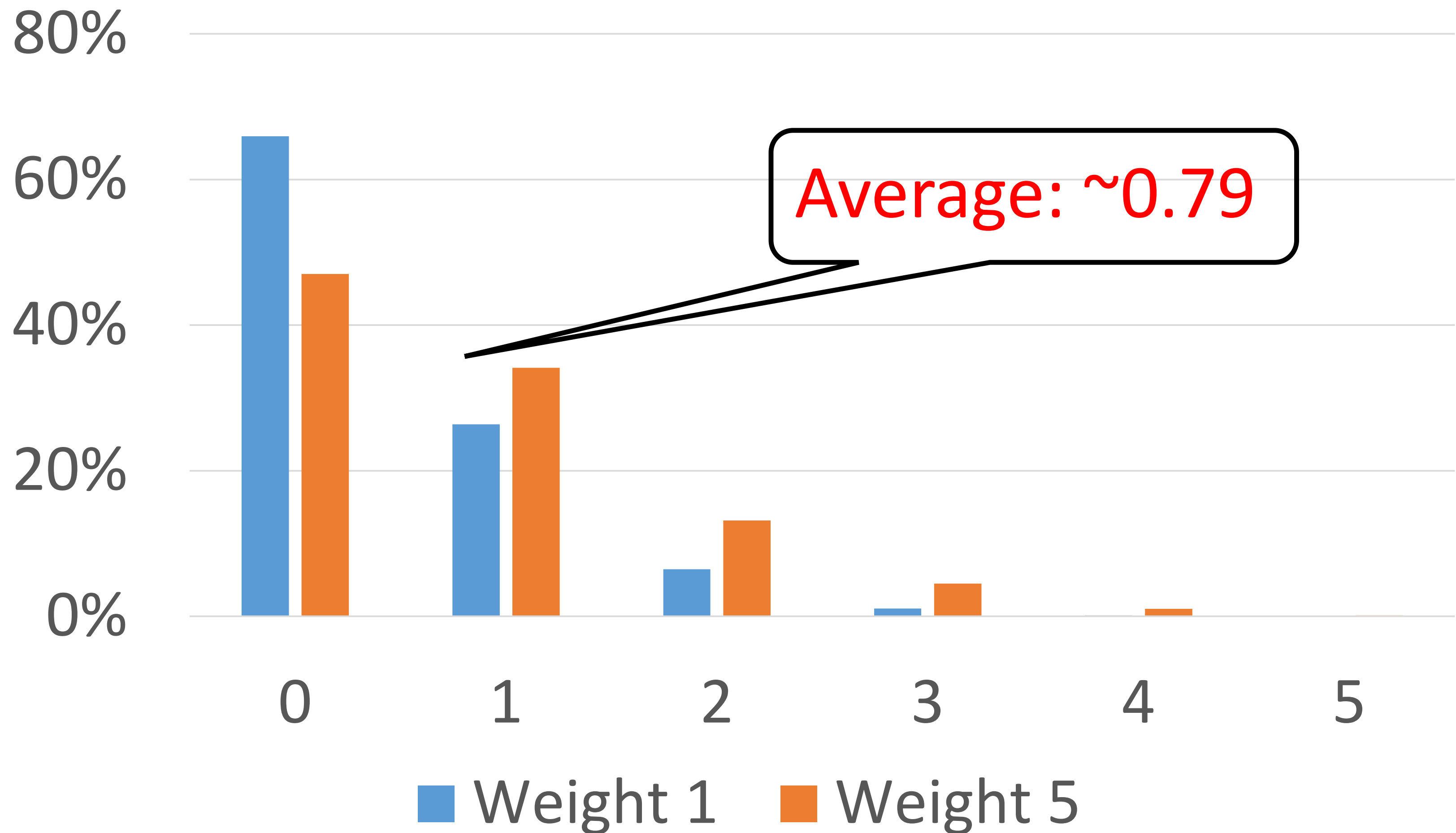
Name	Pid
a	…
b	…
c	…

- register(Name,Pid)
  - —add a process to the registry
- whereis(Name)
  - —return corresponding Pid
- unregister(Name)
  - —remove Name from registry

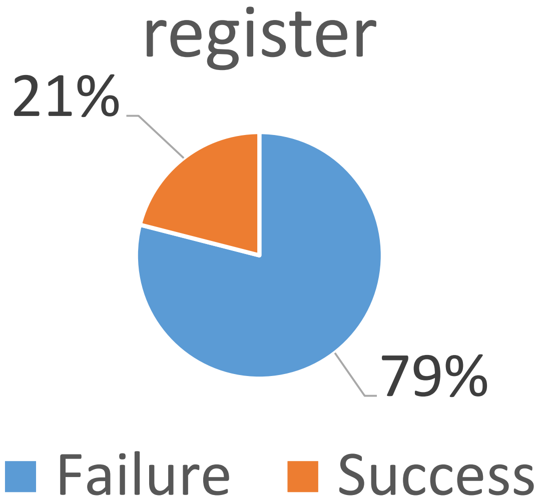
- spawn()
  - —create a process to be registered
- kill(Pid)
  - —kill a process (dead processes cannot be in the registry)

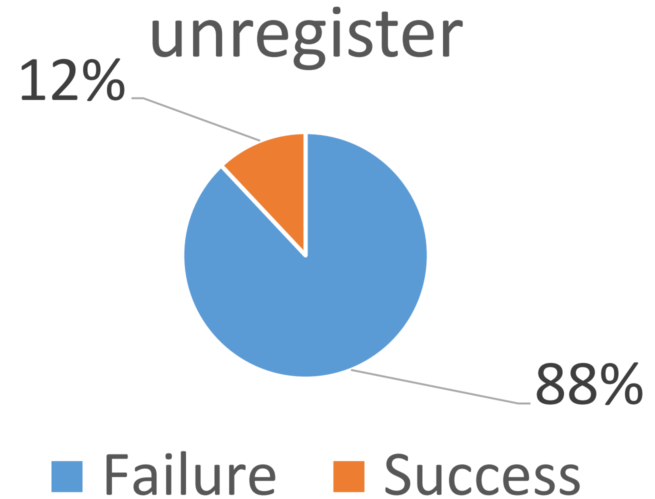

Number of processes in the registry
70%
60%
50%
40%
30%
20%
10%
0%
Average: ~0.43
0
1
2
3
4
5

What if we choose register more often?

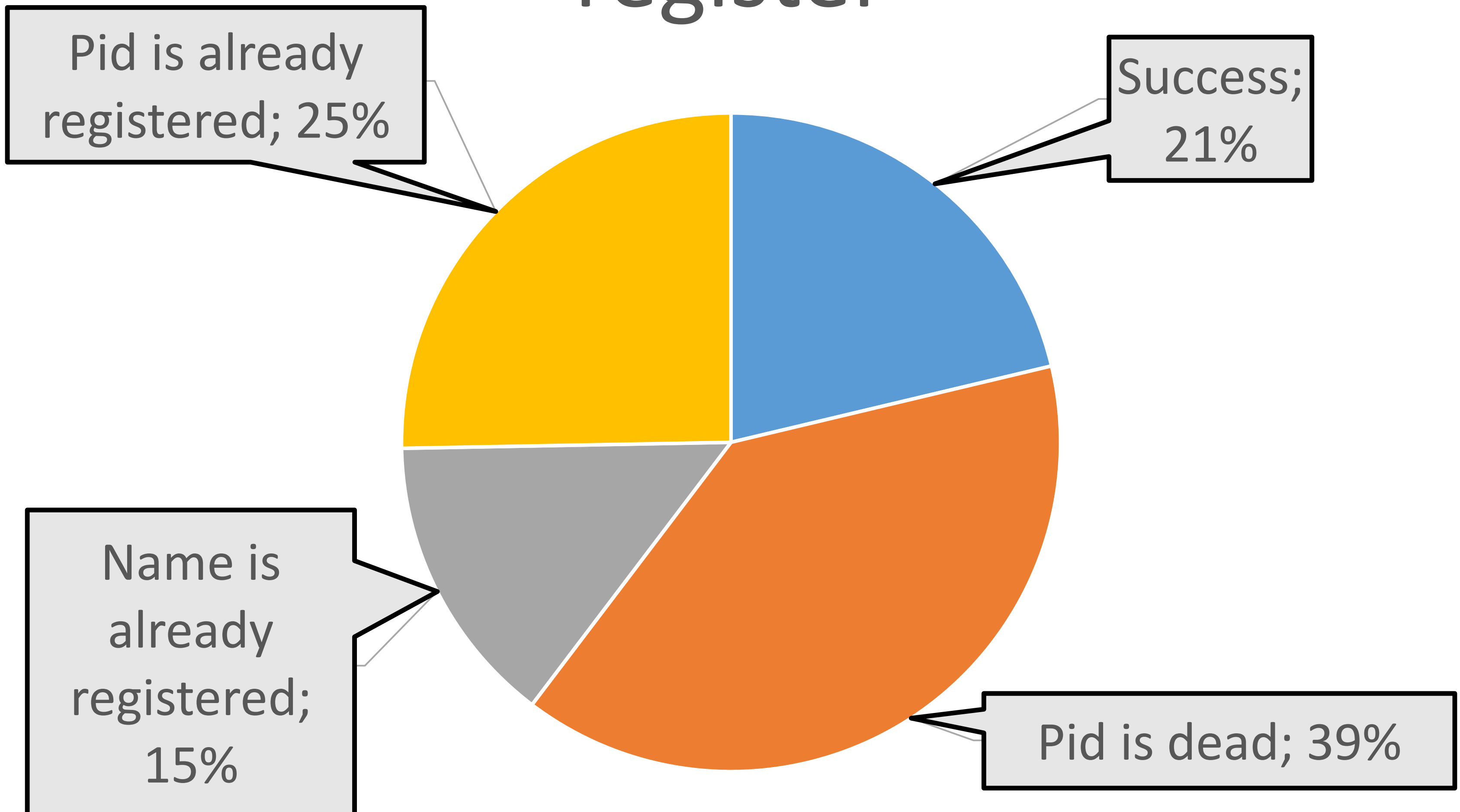


Why aren't we registering more processes?





Outcomes of calls to register



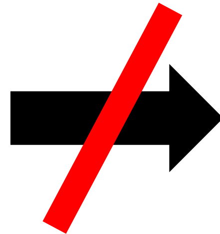
Options

- Choose *arguments* to register/unregister that are likely to succeed?

- Kill processes less often?

- Spawn new processes more often?

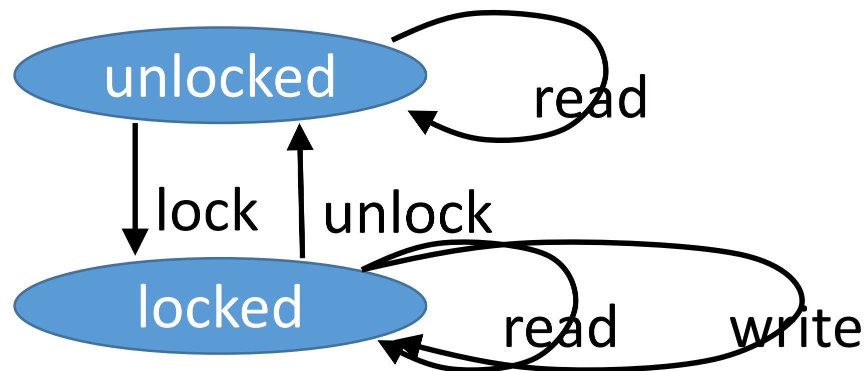
Uniform distribution of inputs



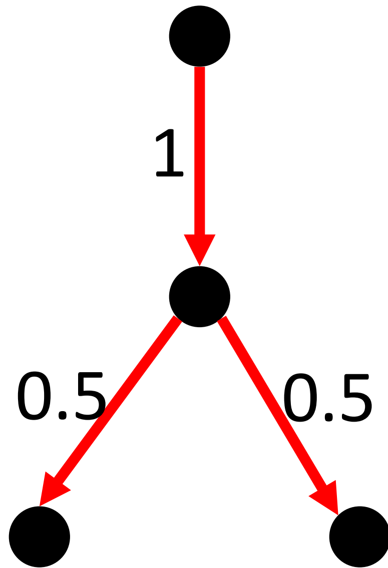
Uniform distribution of outcomes

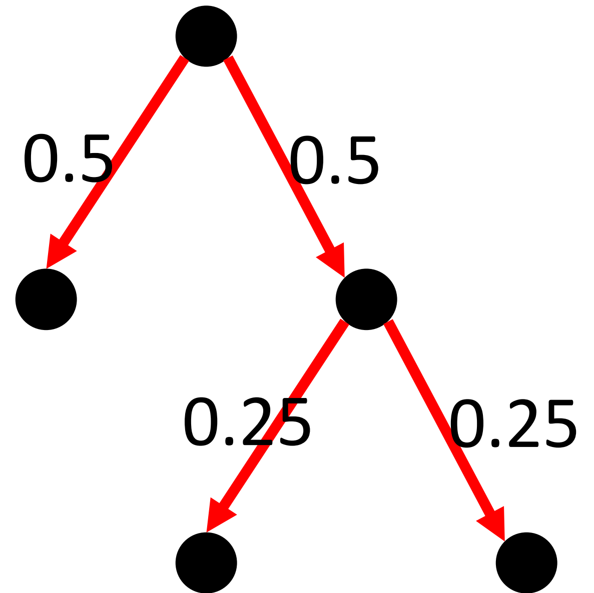
Automating weight assignment?

- # bugs # lines of code
- Test each line of code equally often?
- Follow each transition equally often?



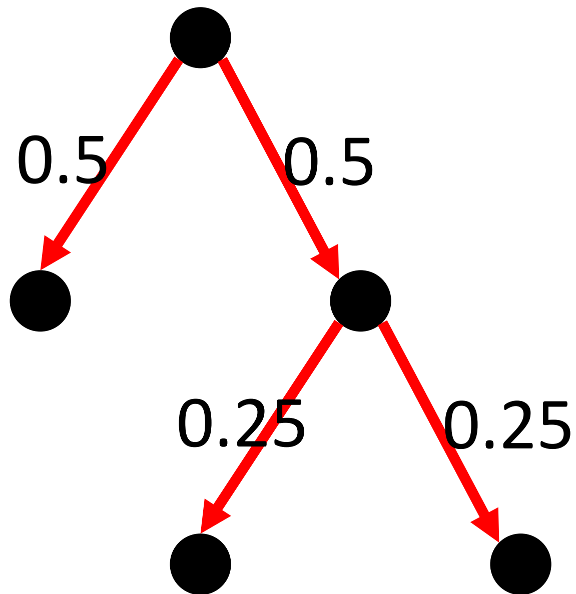
Impossible!

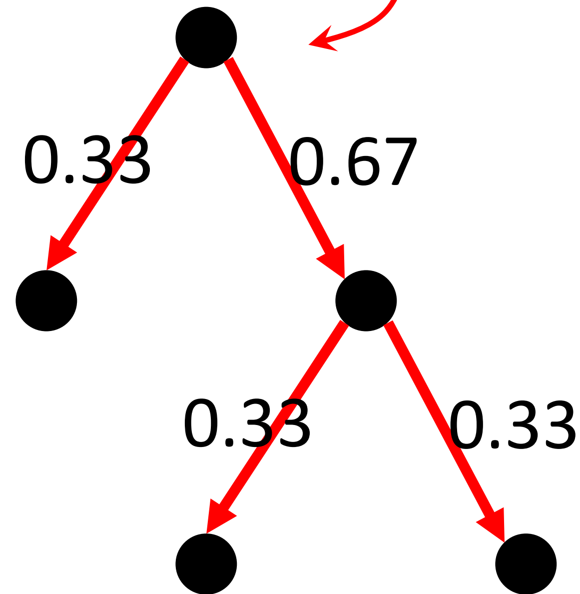




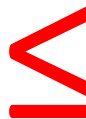
Two weightings

This one is better





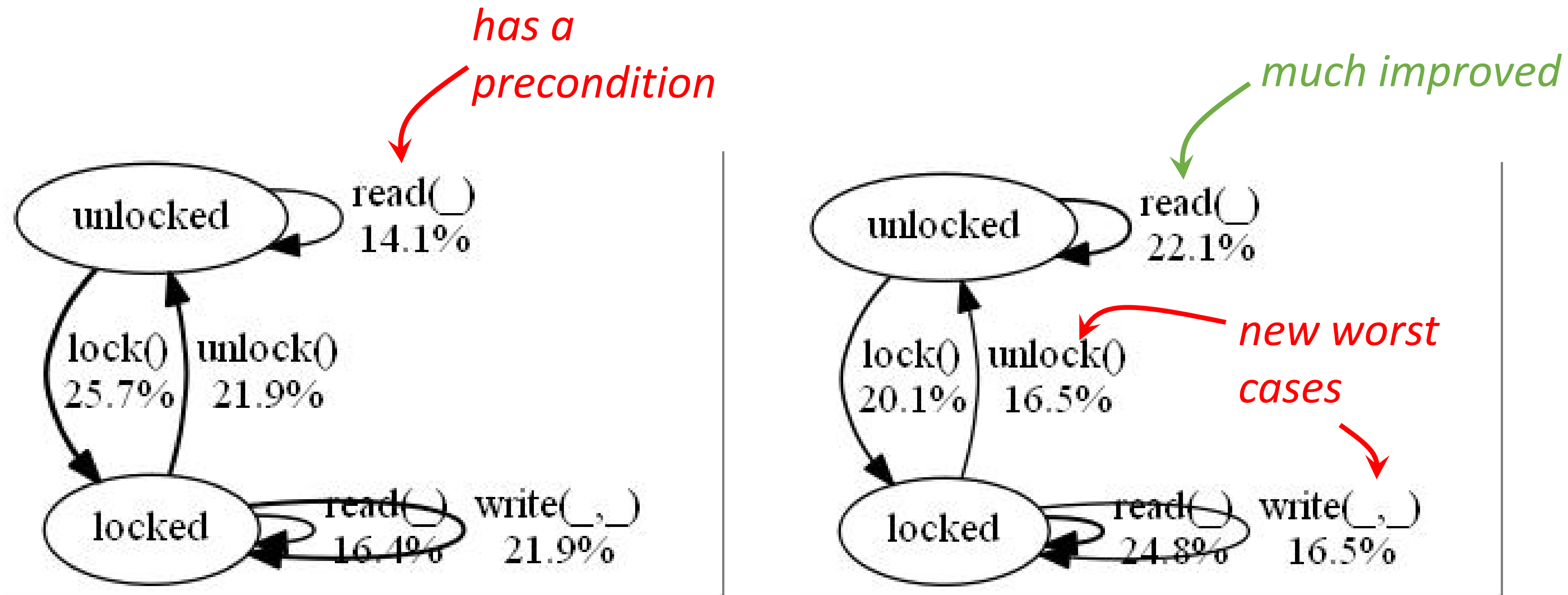
[0.25,0.25,0.5,0.5]



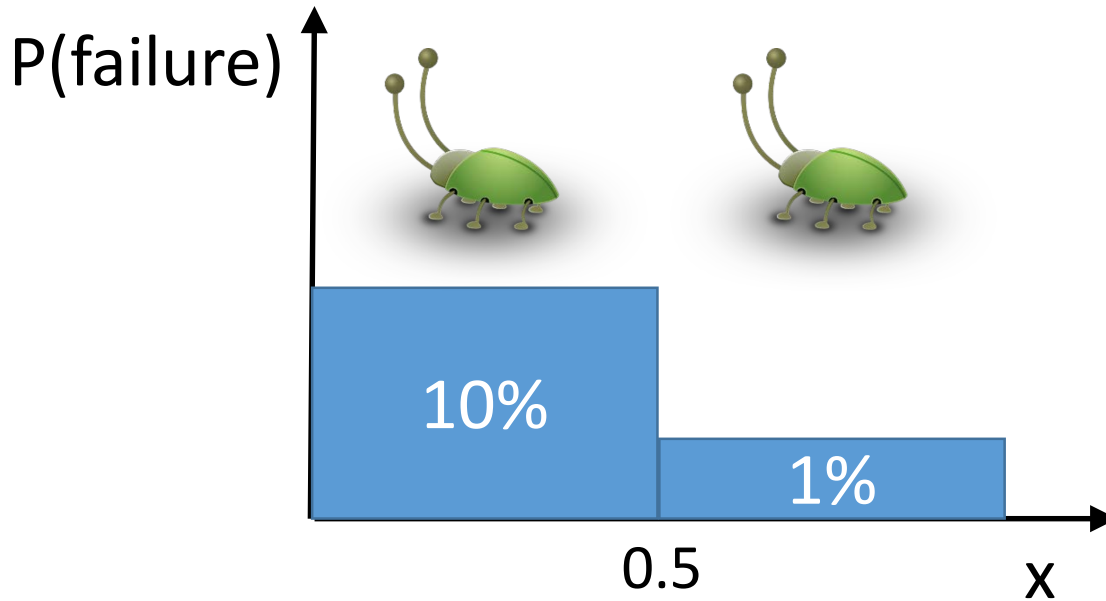
[0.33,0.33,0.33,0.67]

lexicographically

Weighting the locker

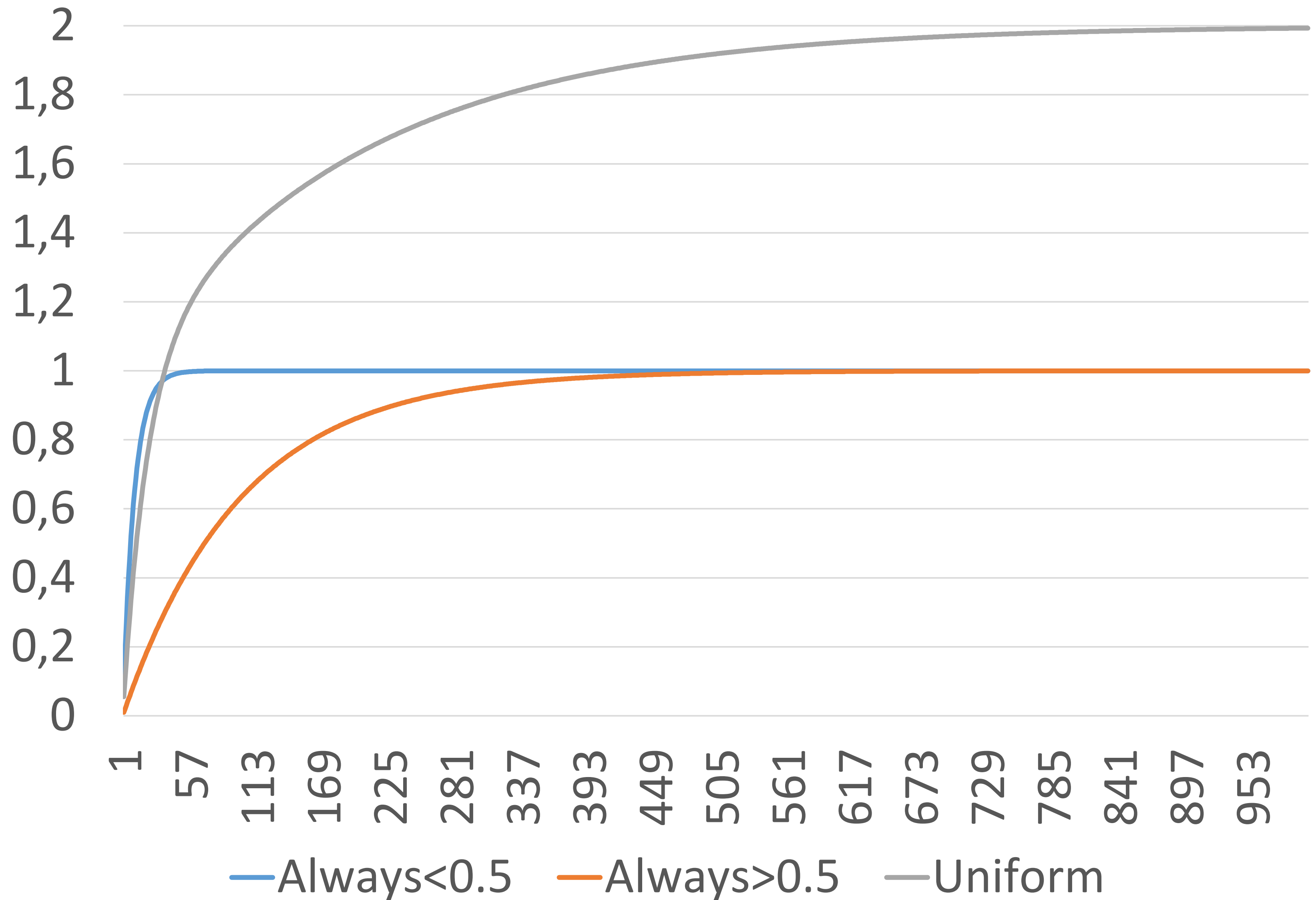



```
weight(locked, locked, read, [_]) -> 2;
weight(unlocked, unlocked, read, [_]) -> 2;
weight(_,_,_,_) -> 1.
```



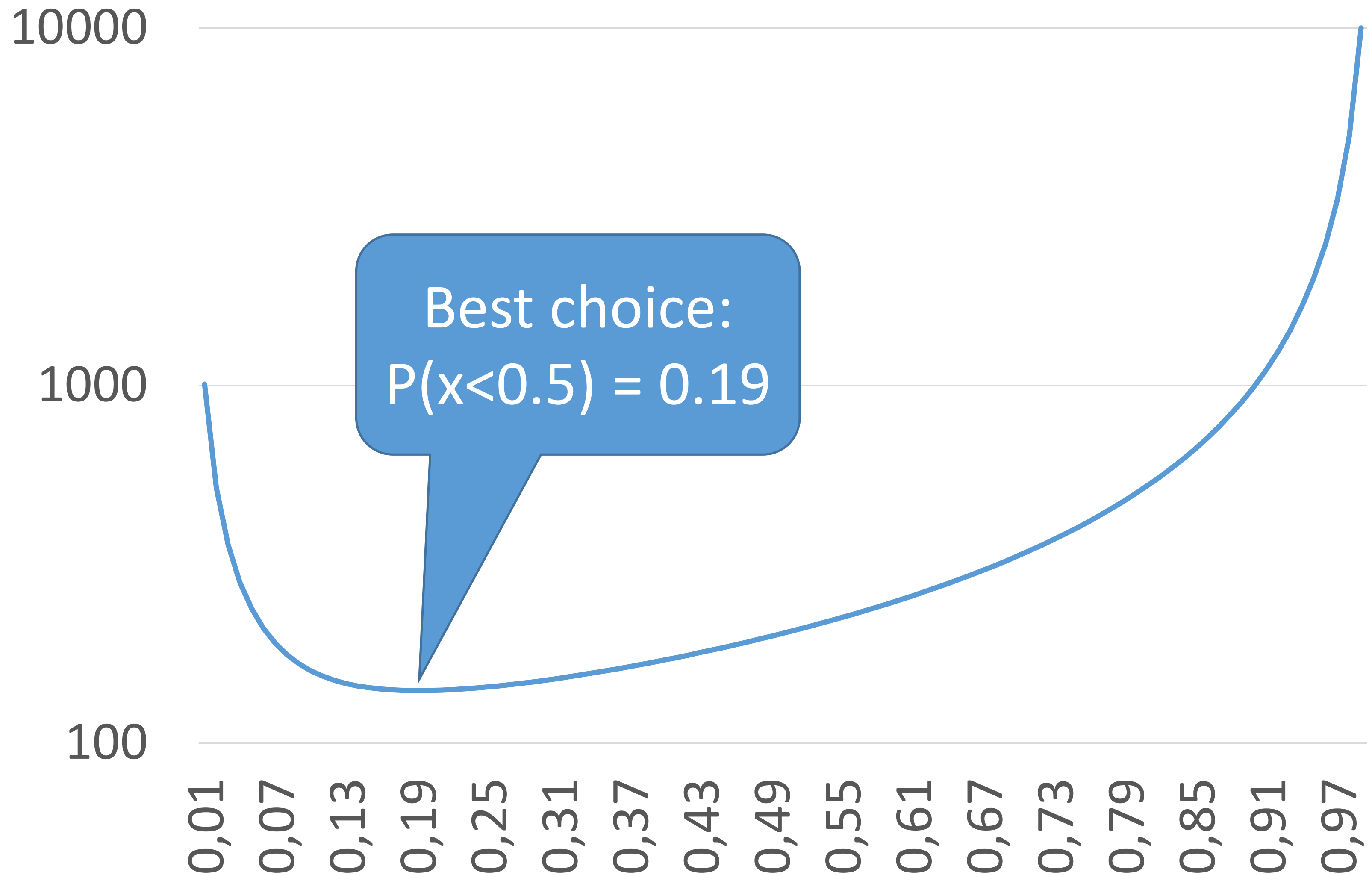
How should we choose x?

- Uniform ➔ P(failure) = 5.5%
- P(x<0.5)==1 ➔ P(failure) = **10%**!
- P(x>0.5)==1 ➔ P(failure) = 1%

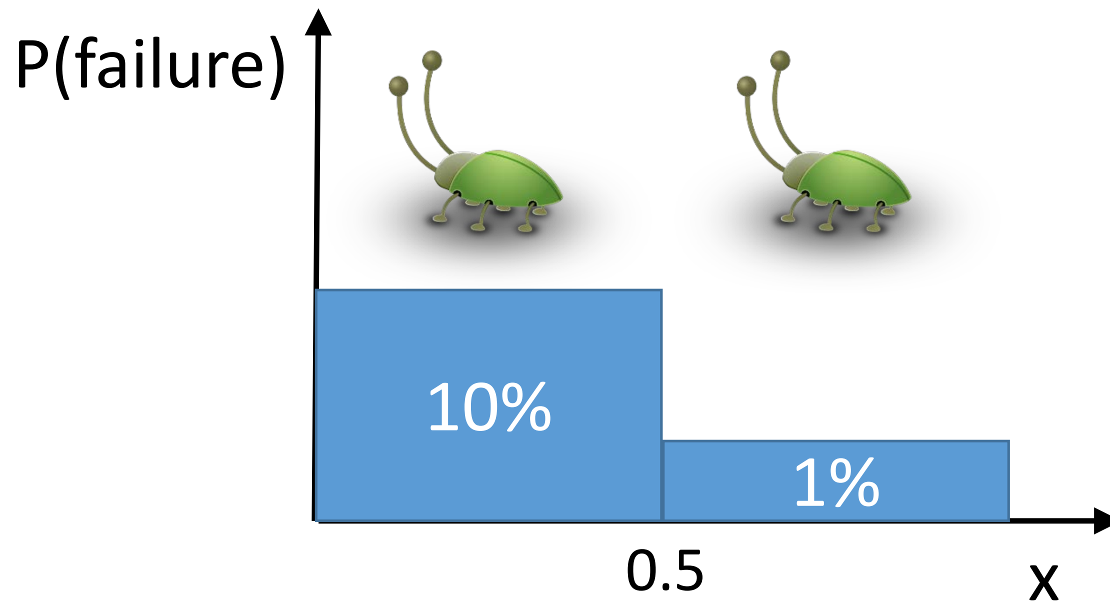


Expected number of bugs found
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1
57
113
169
225
281
337
393
449
505
561
617
673
729
785
841
897
953
Always<0.5
Always>0.5
Uniform

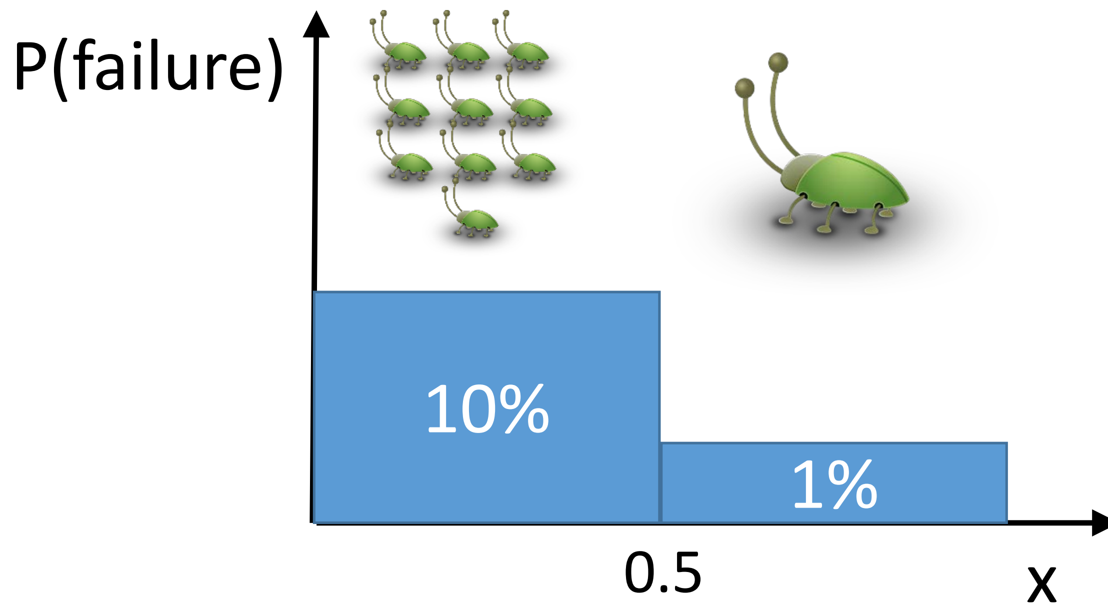
Time to find both bugs



Assumption

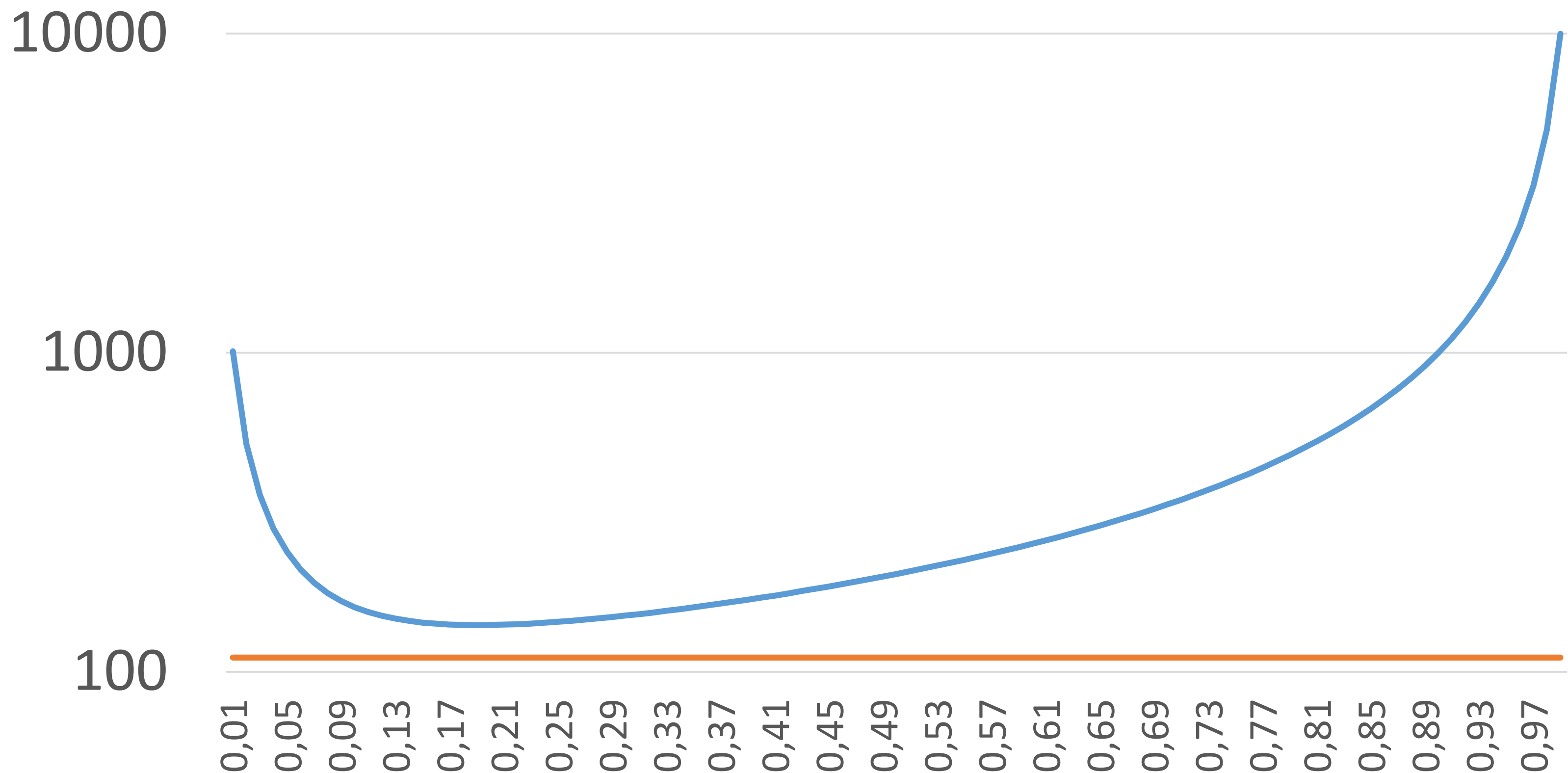


Assumption



Finding the same bug several times is not useful

Why not just stop testing for a bug once it's found?



Testing the registry—do all calls succeed?

unregister(a)

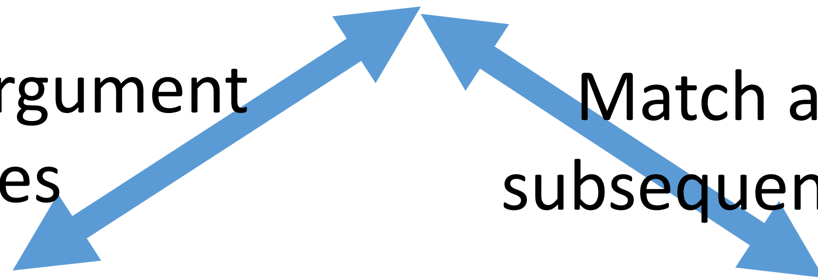
A rarer bug

```
Pid = spawn(),
register(a,Pid),
register(a,Pid)
```

A failed idea

- Avoid the known minimal test case

unregister(a)



Abstract argument values

Match a subsequence

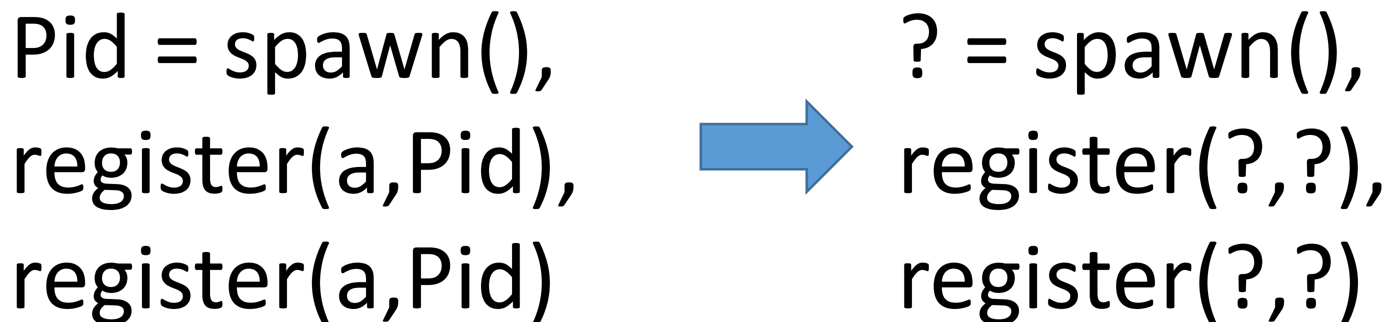
unregister(b)

Pid = spawn(),
unregister(a)

First idea

- Generalize a bug to a *bug pattern* matching any test case with a matching subsequence of *functions*

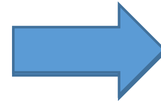
- Don't generate more calls to unregister



Second idea

- *Ignoring* argument values seems a bit crude!

```
Pid = spawn(),
register(a,Pid),
register(a,Pid)
```



```
? = spawn(),
register(?,?),
register(?,?)
```

```
?Pid = spawn(),
register(?A,?Pid),
register(?A,?Pid)
```

- Same ?Name must match the *same* value at each occurrence

A New Bug!

```
Pid1 = spawn(),
Pid2 = spawn(),
register(a,Pid1),
register(a,Pid2)
```

```
Pid1 = spawn(),
register(a,Pid1),
Pid2 = spawn(),
register(a,Pid2)
```

```
Pid1 = spawn(),
Pid2 = spawn(),
register(a,Pid2),
register(a,Pid1)
```

Third idea

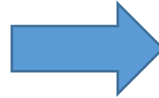
- Generalize *order* using parallel composition

```
?Pid1 = spawn(),
register(?A,?Pid1) || ?Pid2 = spawn(),
                      register(?A,?Pid2)
```

- (This means *any interleaving*, there is no real parallelism)

Another new bug

```
Pid = spawn(),
register(a,Pid),
register(b,Pid)
```



```
?Pid = spawn(),
register(?A,?Pid),
register(?B,?Pid)
```

- Generalizes (and replaces) the second bug pattern found

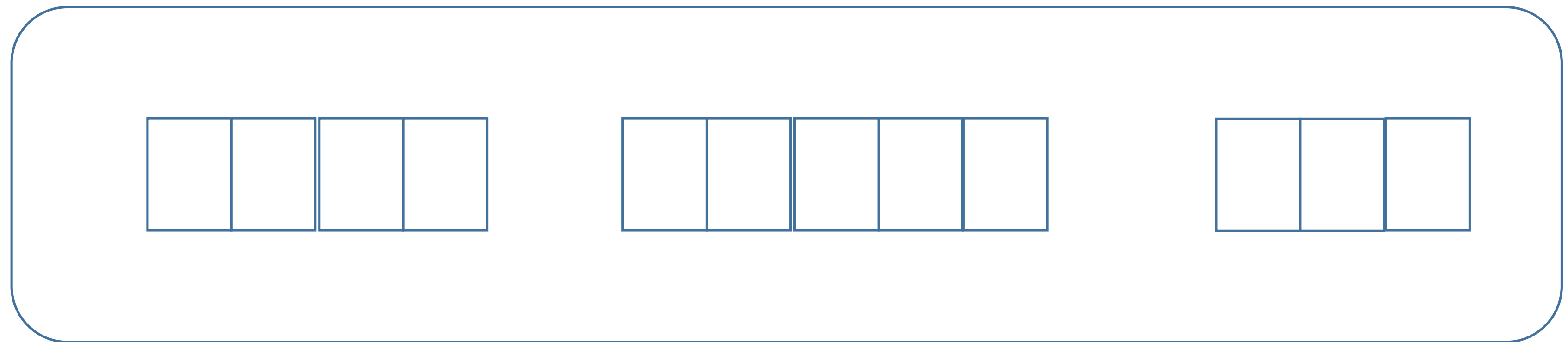
Evaluation of ”MoreBugs”

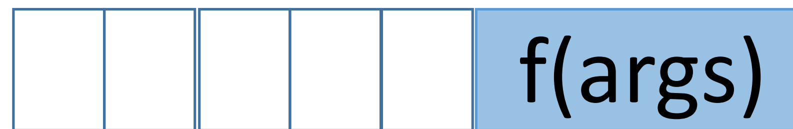
- Comparison with vanilla QuickCheck

- Comparison with RANDOOP
  - which *also* uses ”feedback” from earlier tests to guide the choice of later ones

RANDOOP style testing

- RANDOOP-style test case generation for QuickCheck state machine models





Choose a sequence satisfying f's precondition

Uniform distribution of testing functions

Evaluation: Erlang process registry

Bug	QC	RDS
1	258.1	12.8
2	13.9	3.1
3	18.9	3.1
4	1.1	1.8
5	5.7	5.1

How often each bug is provoked per 1,000 calls to the API

Time to find all five bugs

QC (no shrinking)	QC	RDS	MoreBugs
944	4893	781	713

again, measured in API calls

Evaluation: AUTOSAR CAN stack

The CAN bus is used in vehicles; AUTOSAR is a standard for vehicle software

- 50 functions in the API
- 20 KLOC of C
- QC model is 5 KLOC of Erlang

”Bugs” inserted via incompatible configuration

Time to find bugs

Bug			
1			
2			
3			
4			
5			
6			
7			
8			
9			

Time to find bugs

Bug	QC		
1	5k		
2	1,000k		
3	2,000k		
4	500k		
5	1,000k		
6	2,000k		
7	-		
8	-		
9	1,000k		

Time to find bugs

Bug	QC	MoreBugs	
1	5k	5k	
2	1,000k	1,000k	
3	2,000k	-	
4	500k	2,000k	
5	1,000k	-	
6	2,000k	500k	
7	-	500k	
8	-	1,000k	
9	1,000k	1,000k	

Time to find bugs

Bug	QC	MoreBugs	RDS
1	5k	5k	100k
2	1,000k	1,000k	-
3	2,000k	-	-
4	500k	2,000k	-
5	1,000k	-	-
6	2,000k	500k	-
7	-	500k	-
8	-	1,000k	-
9	1,000k	1,000k	-

Summary

- Optimise *bug-finding rate*, not *test failure rate*
- Generating larger tests and shrinking failures improves bug finding rate
- Tuning input distribution is a creative process
  - —using measurements of outcomes as a guide
  - —multi-stage generation beats generate-and-filter
- Uniform distribution of code execution may help
- Adapting generation to avoid known bugs can speed up bug discovery