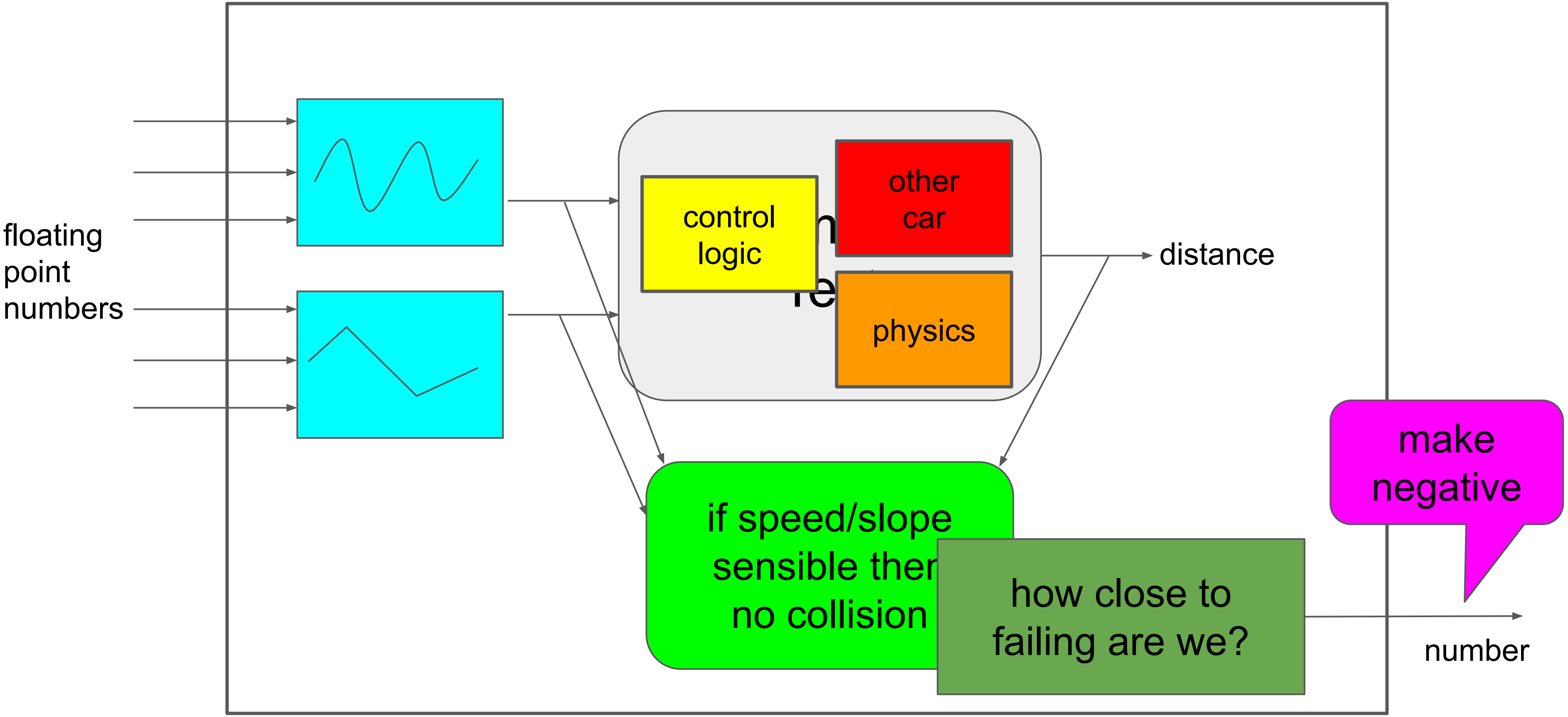
Numerical Optimization for Generating Test Data

Koen Claessen
Chalmers University of Technology
Sweden

Example: Adaptive cruise control




floating point numbers
use numerical minimization!
make negative
number
“falsification”

needs gradient

Numerical Optimization Metho

complicated

- Gradient descent
- Nelder-Mead
- Swarm optimization
- ...

gravitate towards local minima

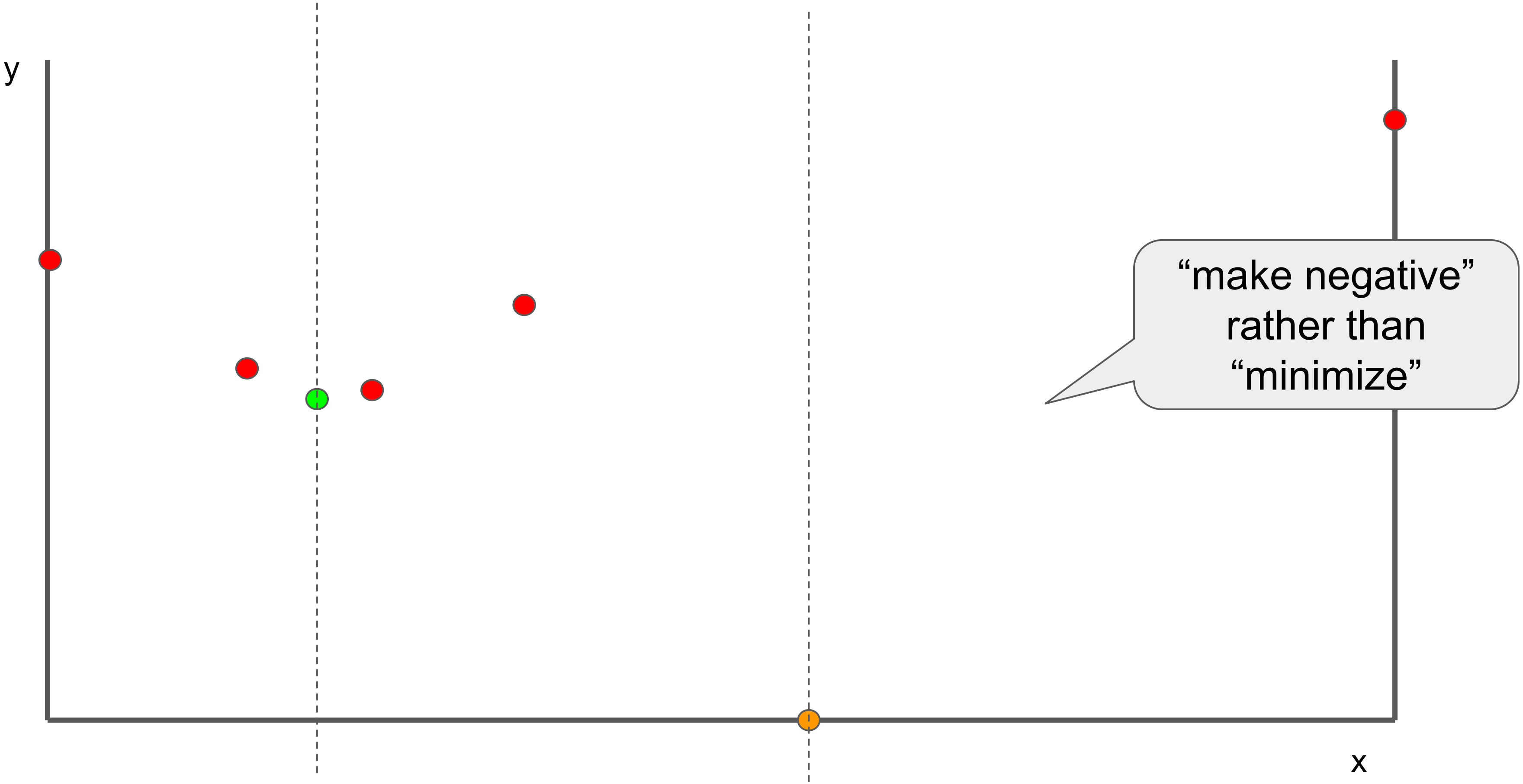
- SNOBFIT
- Bayesian optimization
- ...

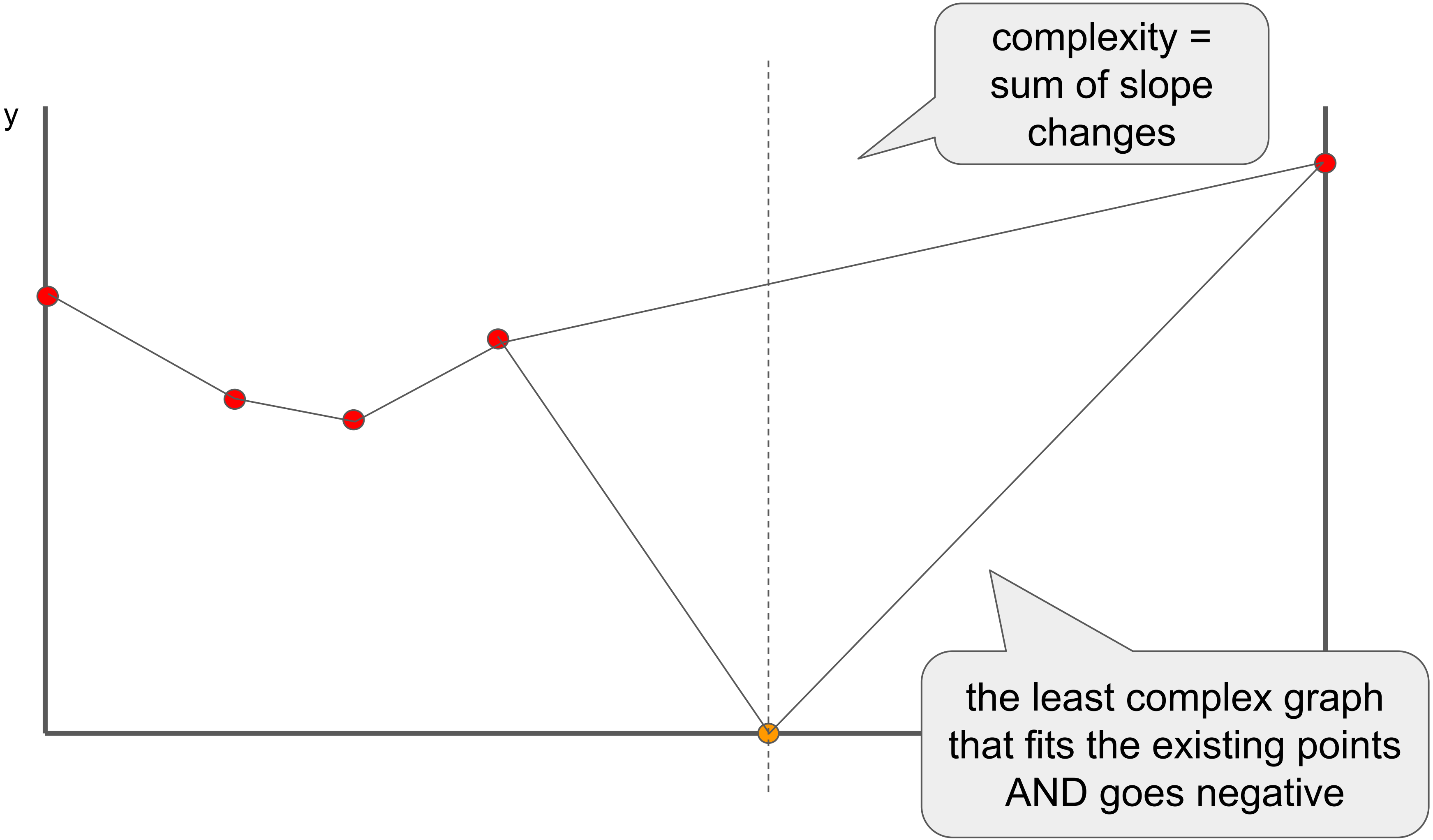
expensive

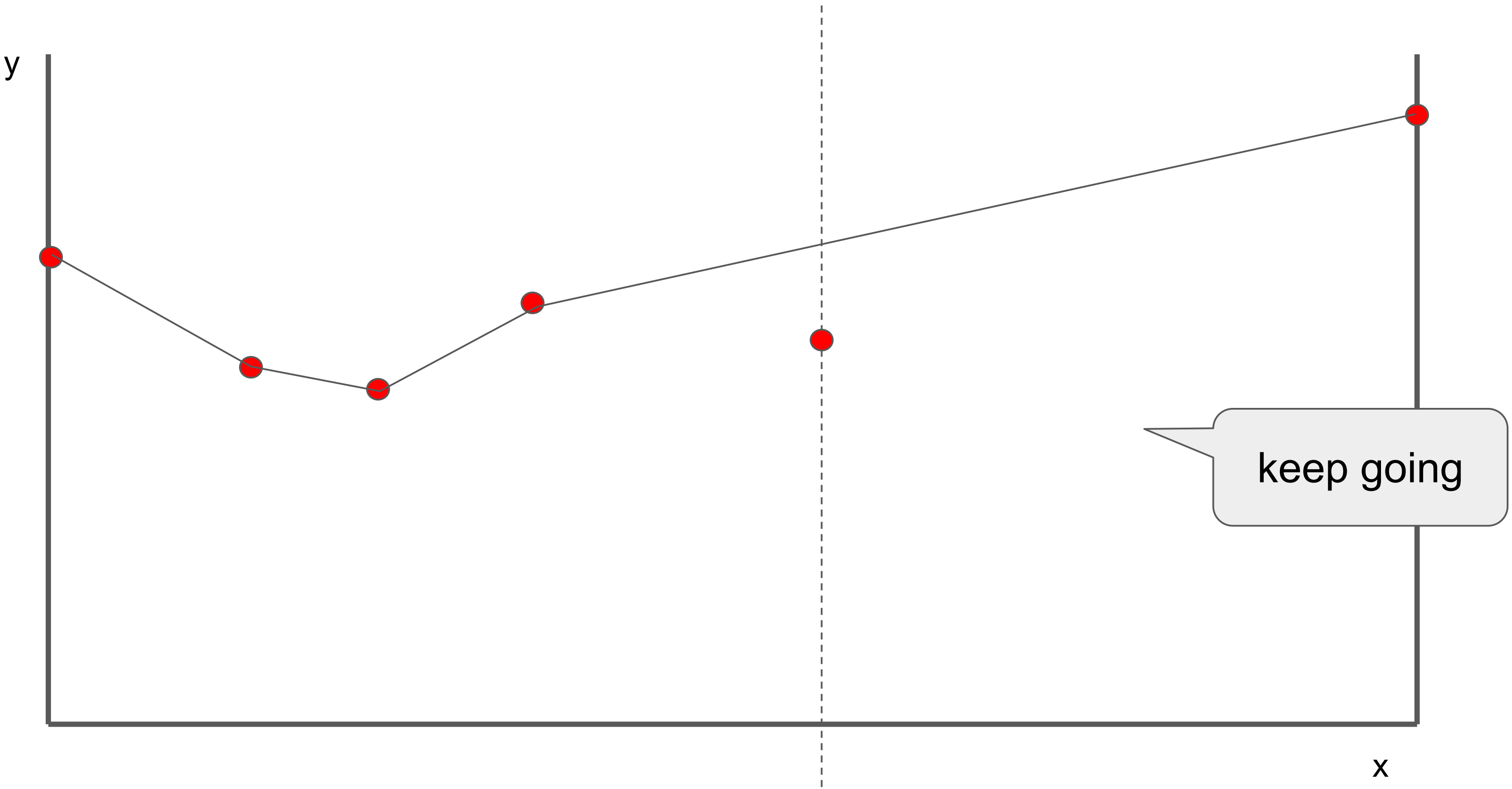
Which aspects are important for falsification

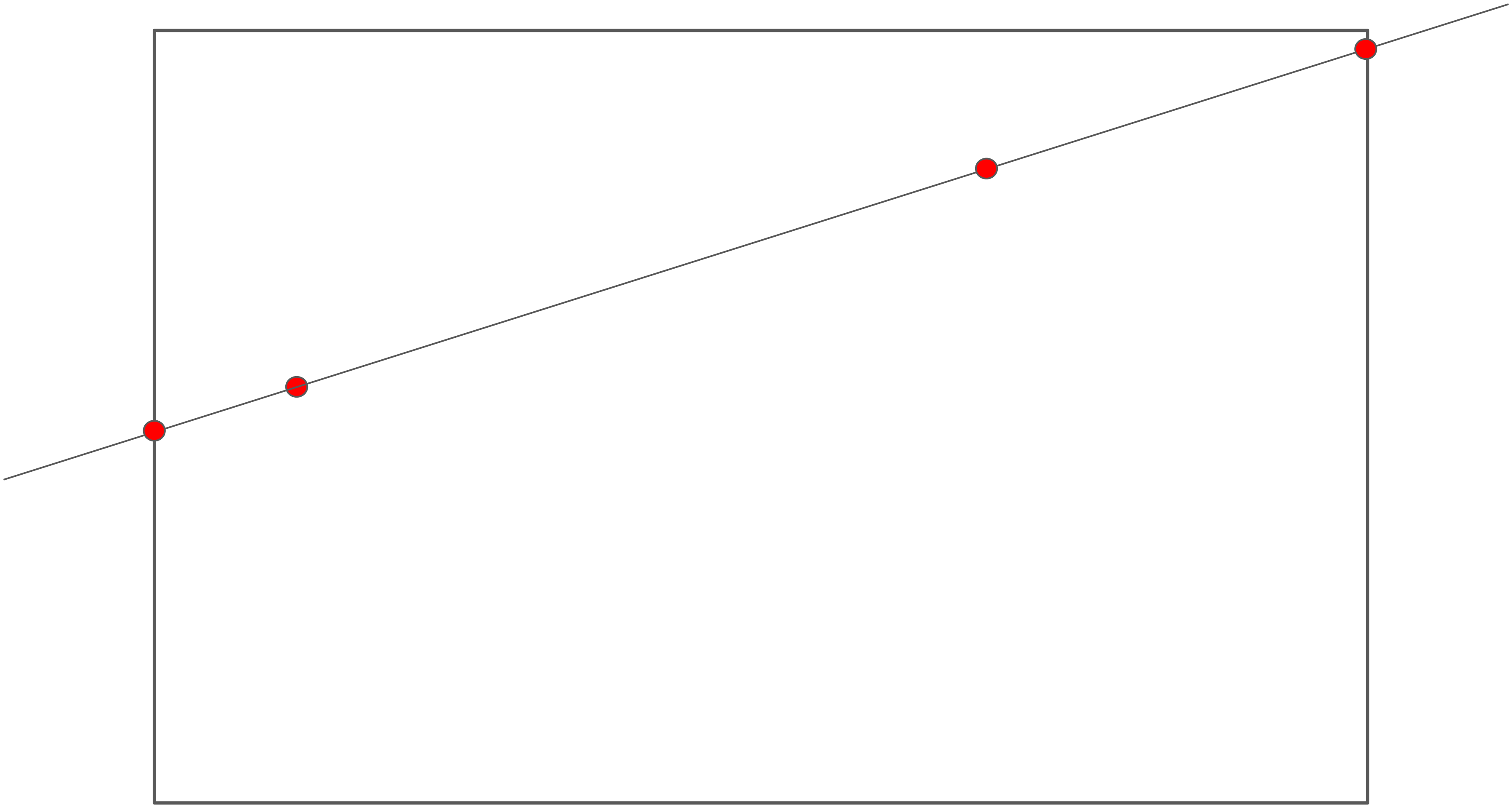
- No gradient
- Not getting stuck in local minima
  - Do not gravitate to local minima
- Having a defined “area” to search in
  - Extreme values often lead to bugs

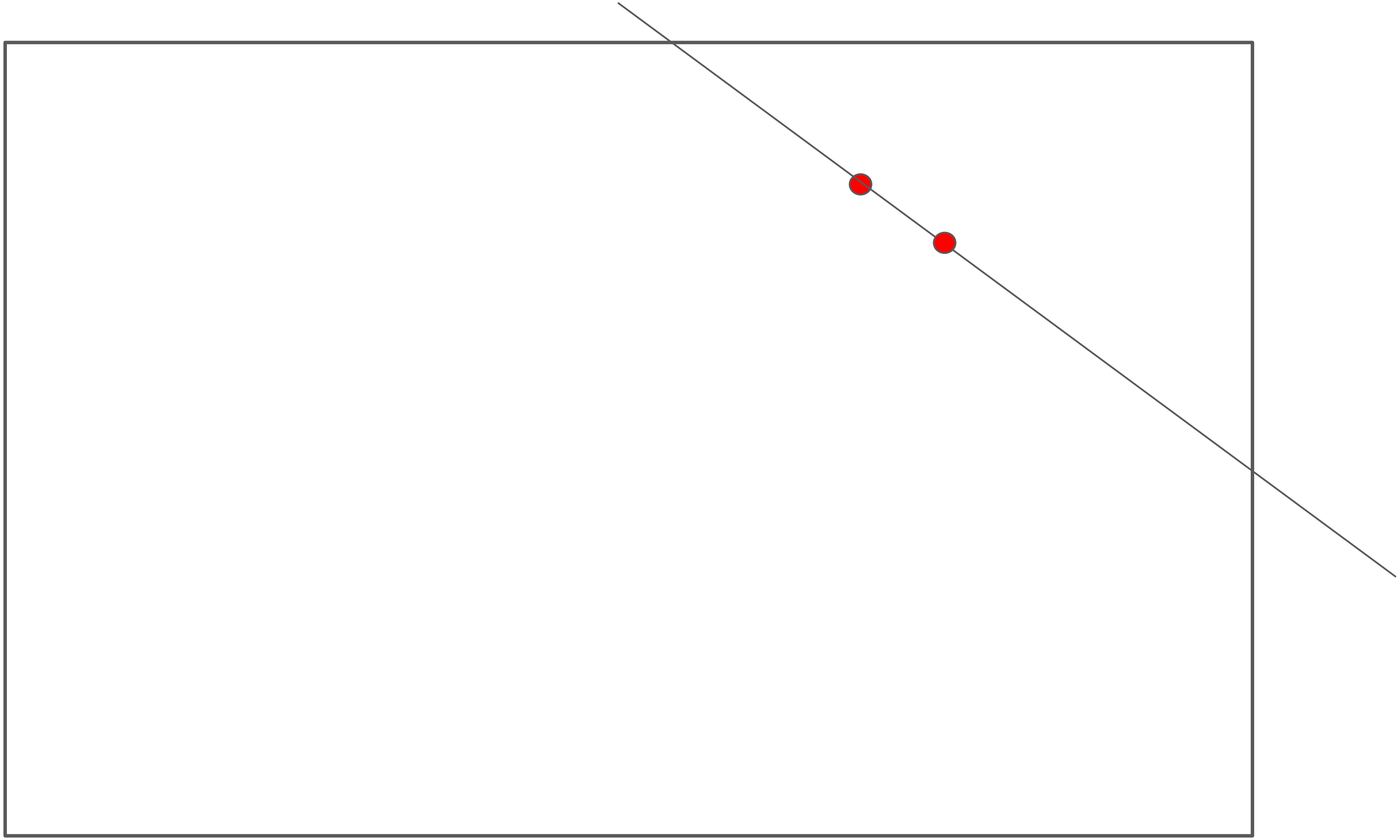
find **negative** results

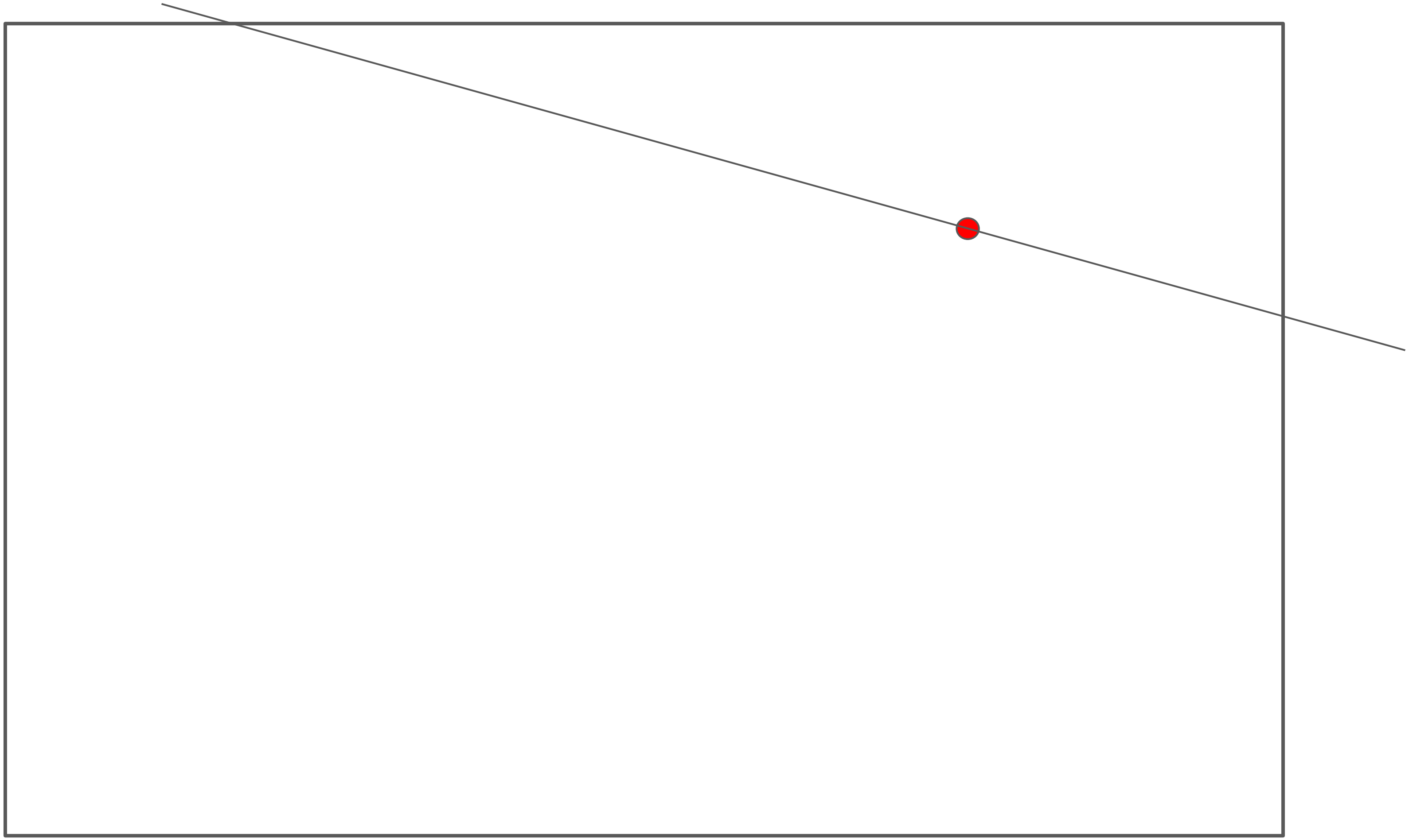

y
x
“make negative”
rather than
“minimize”


complexity = sum of slope changes
y
the least complex graph that fits the existing points AND goes negative


y
x
keep going

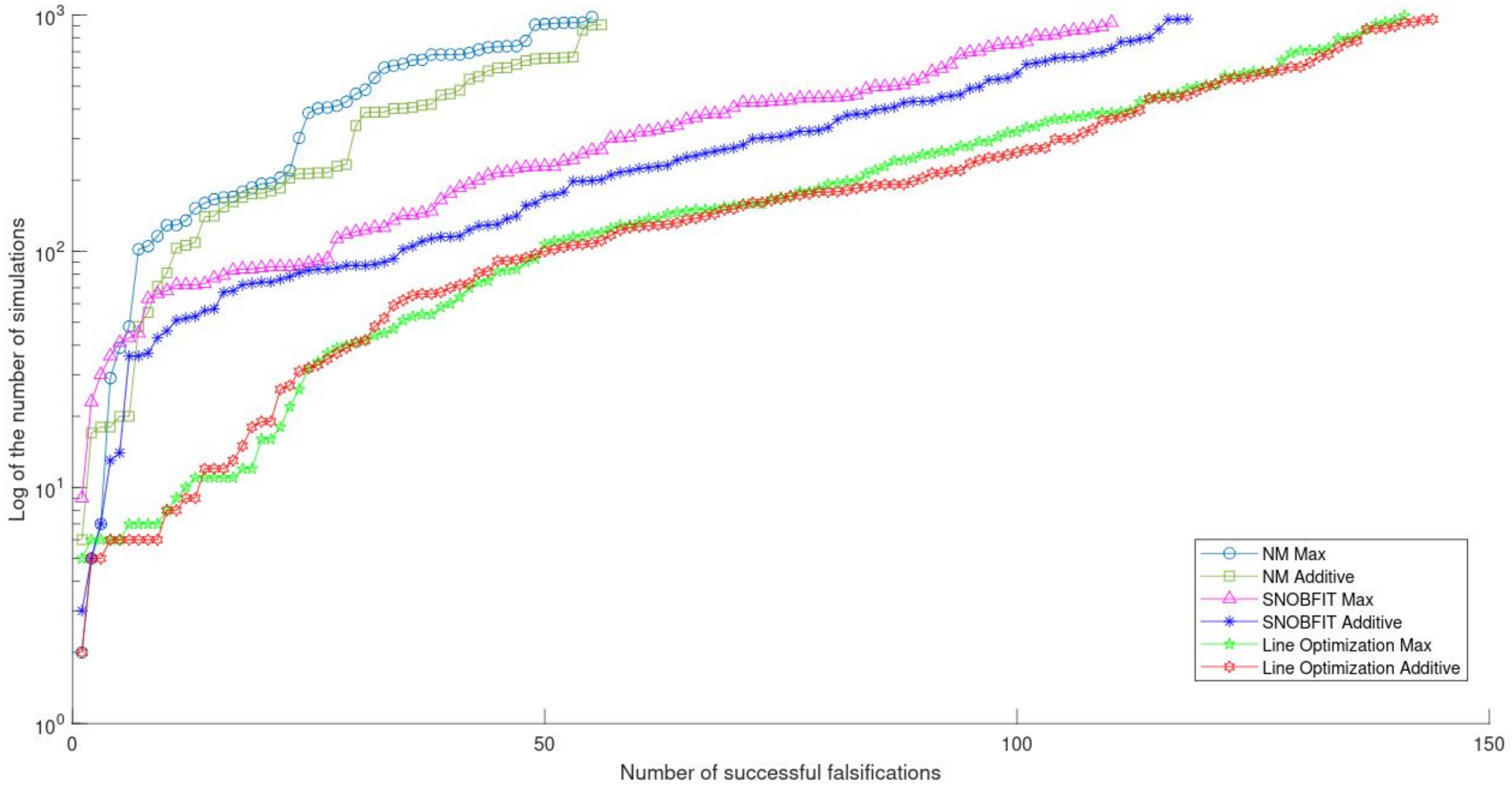






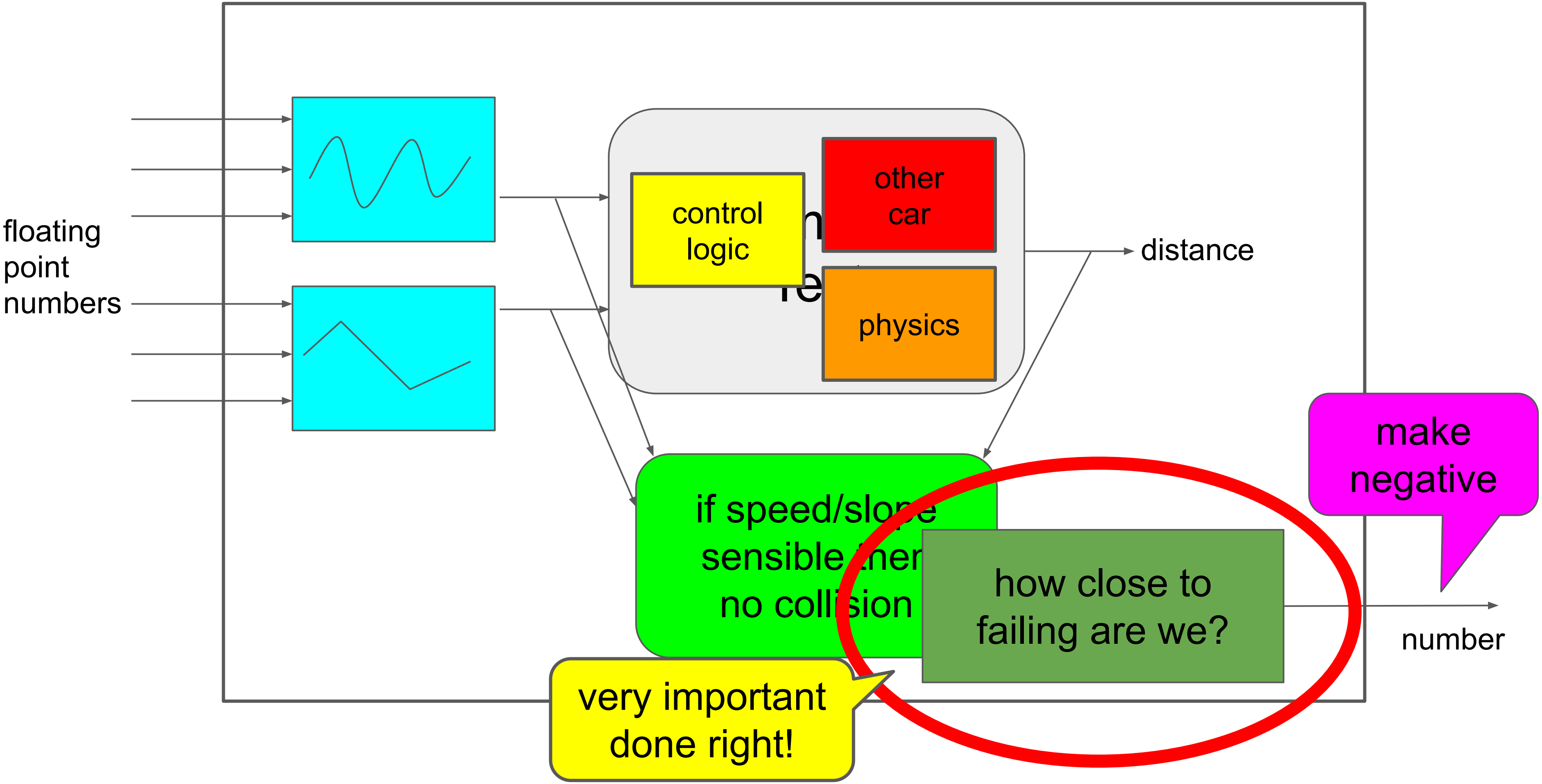
“Line falsification”

- Simple, fast
- Exercises “extreme values” in the box
- Is not sensitive to local minima
- Is not very sensitive to #dimensions
  - Can ignore dimensions that don’t seem to matter
  - 100s of inputs



Log of the number of simulations
Number of successful falsifications
10^0
10^1
10^2
10^3
0
50
100
150
NM Max
NM Additive
SNOBFIT Max
SNOBFIT Additive
Line Optimization Max
Line Optimization Additive

Example: Adaptive cruise control



```haskell
type DBool = Double {- >=0 -}
```

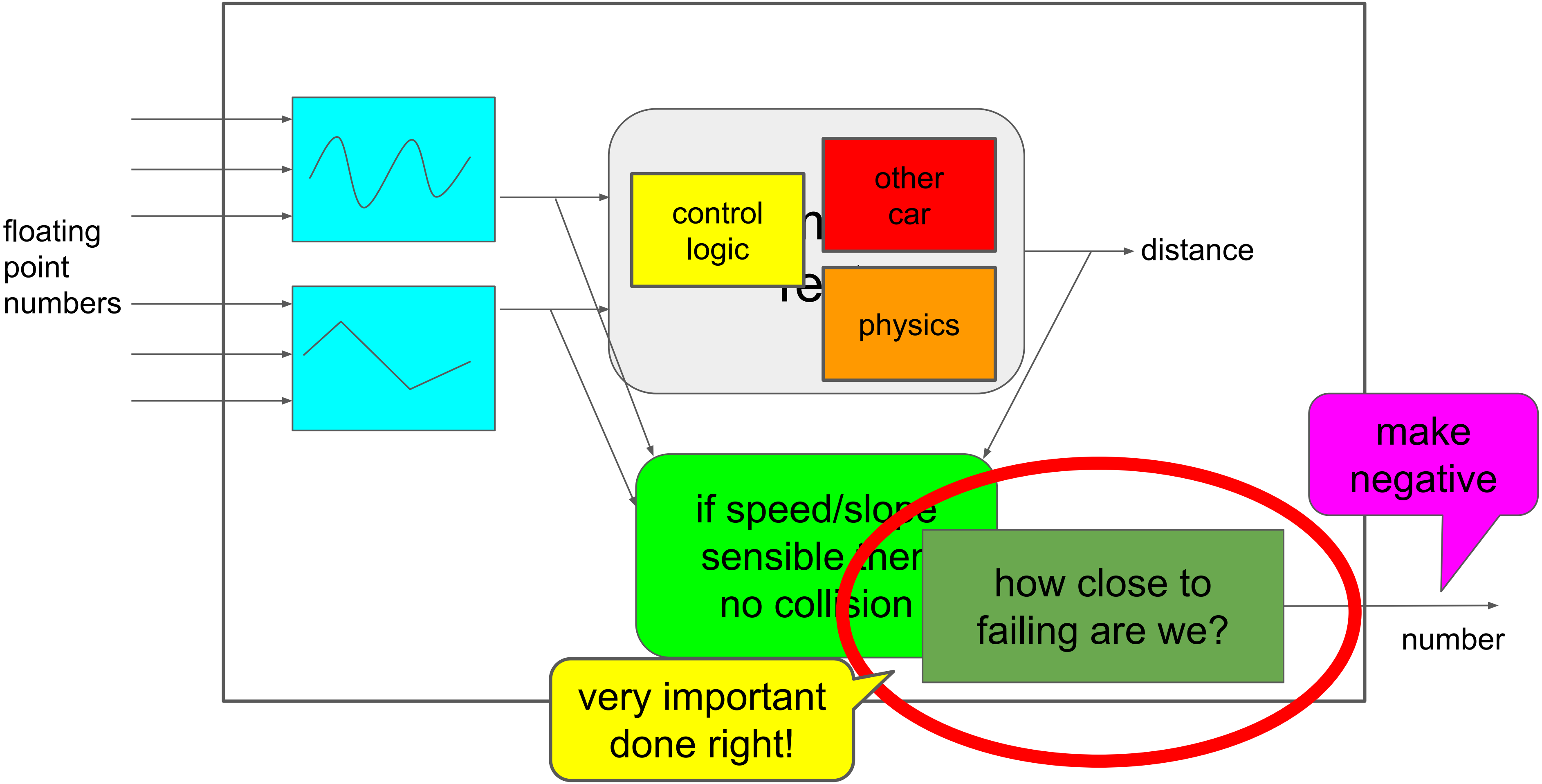
```haskell
false, true :: DBool
false = inf
true  = 0

(&&), (||) :: DBool -> DBool -> DBool
x && y = x + y
x || y = x `min` y
```

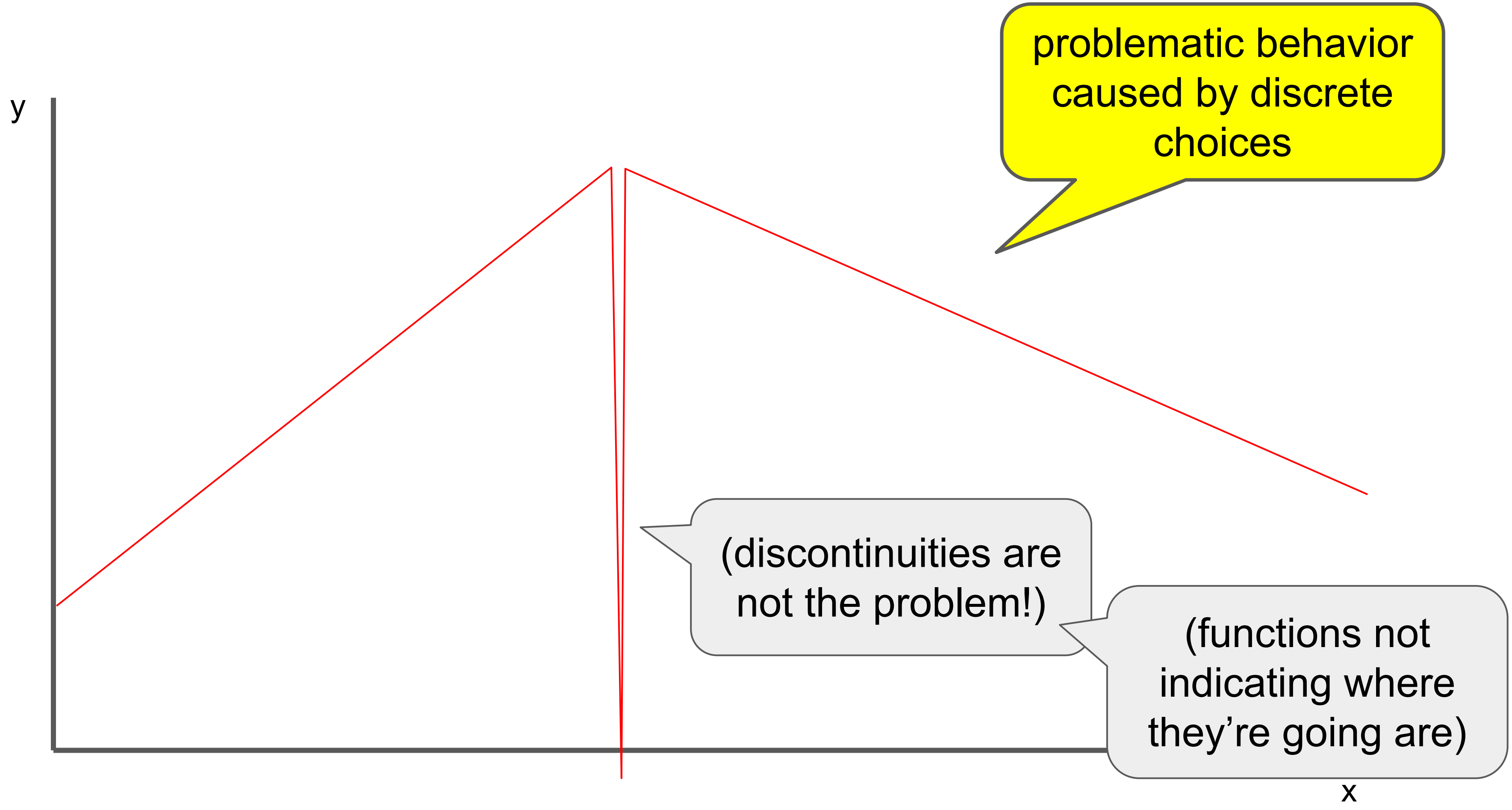
```
type DBool = Double {- >=0 -}
```

```
(>=?) :: Double -> Double -> DBool
x >=? y
  | x >= y    = true
  | otherwise = y - x
```

a specification logic
of “valued booleans”

Example: Adaptive cruise control




problematic behavior caused by discrete choices
y
(discontinuities are not the problem!)
(functions not indicating where they're going are)
x

type for representing simulation values

a table of *alternative* values, and *distance* to them

```
data Val a = Val a [(a,DBool)]
```

the actual value

```
instance Applicative Val
```

```
pure :: a -> Val a
```

```
pure x = Val x []
```

```haskell
instance Applicative Val
```

```haskell
lift2 :: (a -> b -> c) ->
         Val a -> Val b -> Val c
```

```haskell
lift2 f (Val x xds) (Val y yds) = Val (f x y) zds
 where
  zds = table (
        [ (f x y',  d) | (y',d) <- yds ] ++
        [ (f x' y,  d) | (x',d) <- xds ] ++
        [ (f x' y', d1+d2) | (x',d) <- xds
                           , (y',d) <- yds ])
```

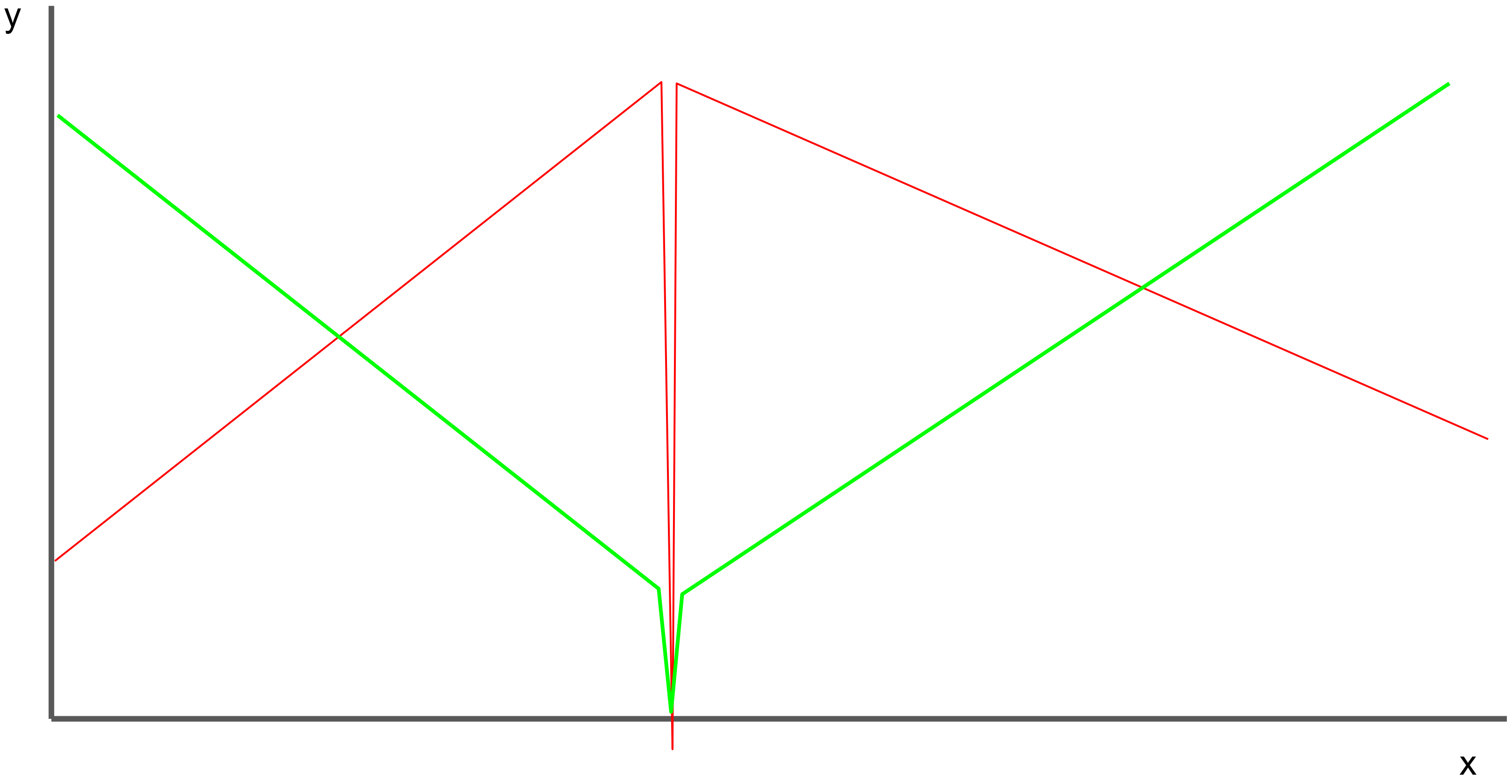
table chooses the minimum distance

```
ifThenElse :: Val Bool ->
              Val a -> Val a -> Val a
```

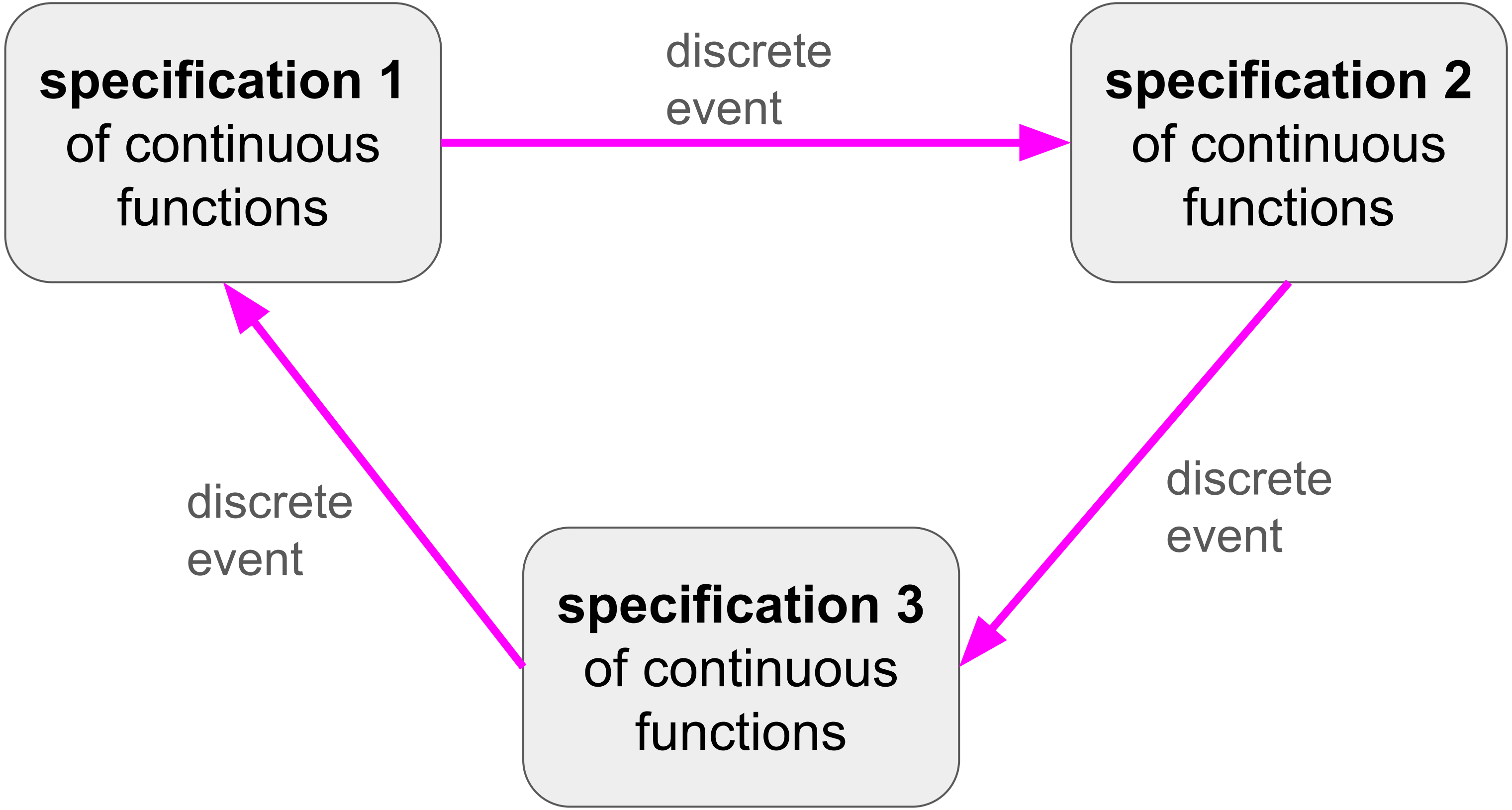
plot the distance to a negative result

```
f :: Val Double -> Val Double
f x = ifThenElse (x <=? 10.0)
        x
        (ifThenElse (x >=? 10.1)
           (20-x)
           (-1))
```

want to automate instrumentation



y
x


specification 1 of continuous functions
discrete event
specification 2 of continuous functions
discrete event
specification 3 of continuous functions
discrete event

```
data State = Spec1 | Spec2 | Spec3
```

current state +
alternative states

continuous inputs

property ok?

```
system :: Val State -> [Double] -> Val Bool
system st xs = ... system st’ xs’ ...
```

Summary

- Very effective method for finding bugs in hybrid systems
  - flexible (e.g. math / black-box

compares well in typical benchmarks

- “Additive” valued booleans

used in physics! (lasers)

- Our own numerical optimization method
  - good at finding negative values (vs. minimization)

- How about fully discrete systems?

```
parse :: String -> Maybe T
show  :: T -> String
```

```
prop_parse_show (x :: T) =
  parse (show x) == Just x
```

(show should really be non-deterministic)

```
prop_show_parse (s :: String) =
  let pres = parse s in
    isJust pres ==>
      s == show (fromJust pres)
```

implies

```
prop_show_parse' (x :: T) =
  let s = show x in
    let pres = parse s in
      isJust pres ==>
        s == show (fromJust pres)
```

need to check that we do not parse too much!

```
parse q d (c:s) =
  case c of
    '(' | q == 0 ->
      parse 0 (d+1) s

    '+' | q == 1 || q == 2 ->
      parse 0 d s

    ')' | q == 1 || q == 2 ->
      parse 2 (d-1) s

    _ | '0' <= c && c <= '9' && (q == 0 || q == 1) ->
      parse 1 d s

    _ ->
      False

parse q d [] =
   d == 0 && (q == 1 || q == 2)
```

```haskell
parse q0 q1 q2 d (c:s) =
  case c of
    '(' | q0 ->
      parse True False False (d+1) s

    '+' | q1 || q2 ->
      parse True False False d s

    ')' | q1 || q2 ->
      parse False False True (d-1) s

    _ | '0' <= c && c <= '9' && (q0 || q1) ->
      parse False True False d s

    _ ->
      False

parse q0 q1 q2 d [] =
   d == 0 && (q1 || q2)
```

```haskell
parse q0 q1 q2 d (c:s) =
  parse q0' q1' q2' d' s
 where
  q0' = (c == '(' && q0) || (c == '+' && (q1 || q2))
  q1' = '0' <= c && c <= '9' && (q0 || q1)
  q2' = (c == ')' && (q1 || q2))
  d'  = d + one (c == '(' && q0) - one (c == ')' && (q1 || q2))

  one False = 0
  one True  = 1

parse q0 q1 q2 d [] =
   d == 0 && (q1 || q2)
```

```
parse :: Val Bool -> Val Bool -> Val Bool -> Val Int -> [Val Char]
      -> Val Bool
```

```
parse q0 q1 q2 d (c:s) =
  parse q0' q1' q2' d' s
 where
  q0' = ((c ==. '(') &&. q0) ||. ((c ==. '+') &&. (q1 ||. q2))
  q1' = foldr1 (||.) [ c ==. w | w <- ['0'..'9'] ] &&. (q0 ||. q1)
  q2' = ((c ==. ')') &&. (q1 ||. q2))
  d'  = d + one ((c ==. '(') &&. q0) - one ((c ==. ')') &&. (q1 ||. q2))

  one q | q > 0.1 = 0
  one _           = 1

parse q0 q1 q2 d [] =
  zero d &&. (q1 ||. q2)
```

done manually but systematically

```
1: 16246.0 "\128\128\128\128\128\128\128\128\128\128\128\128\128\128\128\128\128\128\128\12
2: 14159.613405876566 "\133\&8w\128\147\134|\166/\\\128\145v\130\135Q{h\133\130"
3: 14000.797581007737 "\132*v\131\142\136y\166&\\t\145a\152\136B~o\138\129"
4: 13889.697684456827 "\145+n\130\DEL\134\129\178\DC3]\134\138:\135\156?\133>\130}"
5: 13262.59733297698 "C+t\133dz\223\133\RS\183\159\138\&37\182=\144=\129E"
6: 13242.948921948733 ";(v\134iz\221\133#\182\164\137\&47\185@\140<\130H"
7: 11484.400249769613 "((f\133\EM{\178\155#y\165\158\&87\181:\140\CAN~T"
9: 11463.775253154907 "(%_\142\ESCy\180\152\GS{\166\160\&36\170;\133)\DELN"
10: 11420.19252220173 "($_\143\CANy\178\150\RS|\166\154;6\158G\134)~P"
13: 8735.83252235814 "#\USb\133\SUBw\155;!x\GS\156\&4+\182:\135!\DEL6"
14: 8703.19067710843 "#\CANL\138\vr\135E!v\FS\146)*\190K\134\SI|7"
15: 8218.419410501145 "\ESC\SUBE\142\CANc\137D&k$\145\&9\DC3\182J\139\NULe\CAN"
27: 7956.26821813288 "\NUL3G\134\SYNa\152E k!e@\DC4\179JR\GSe\NUL"
29: 7374.530957174507 "\EOTAF]\NULN\148*\RSk0g\RS\b\179Mz\SUBd\b"
33: 6774.30408363287 "\aEBX)C\147*Ok\EMS\CAN\v\174\132F\ETB\156\NUL"
34: 5801.790239481378 "0F\nZ4FT(Oa\RSP\NUL\RS\229\142B\SUBlR"
35: 5653.209328854889 "7=\ETXX9<U(8c\NULT\SOH\RS\227\142:\SYNwV"
36: 4721.329118696091 "6\US+Z43P?@^\b?\a\ETB\231\151<!wC"
37: 4593.049769795101 "7\US8W2\USI?O^\SUB9\a$\229\152<'w;"
38: 4228.367829788856 ": +X)$I-PV\DC43\SOH:\229s@\SUBx1"
39: 3867.5196853961315 "2#\SUB8 $O.@Q\DC4-\ENQ@\231h@\SUB\132\US"
40: 3811.67592083331 "2&\GS3\RS#T.:L\EM\FS\STX>\230hA\SUBz\US"
...
```

```
208: 618.606774397607 "(7,79*5)+(9)'*|<08P/"
209: 615.6372863875167 "(7+8:*5)+)9))*|<09P/"
225: 613.581652239311З "(119:12,2*5)(0vF08J-"
226: 611.7831431512103 "(119:12+3*5))0vE08J-"
237: 590.0426580236012 "(106834+2,6''0v@4;H*"
244: 563.5745831260235 "(903930,3+4)+-x:<>D+"
245: 555.7315838498512 "(903930,3+5)+.x:<>D*"
246: 538.8363549593371 "(901730-2+5)+-u:=>D+"
247: 536.3424754714836 "(40+862-1+3)(-s:<>C+"
250: 531.1760065113687 "(50,643+304+),s9=?A,"
253: 509.40062523136993 "(82+2750311-*,s;9@A+"
254: 497.4706664838121 "(52+2750100.*+s:8@B+"
257: 497.23938144030933 "(33,1715330)++s:8?@*"
258: 495.7613994906361 "(33,1715331)++t:8>B*"
259: 487.44509852135127 "(33,2704421))+t:7?A*"
273: 471.5657432105902 "(57+792784/(+.p9:@;)"
275: 463.08959029546634 "(58+6927840(+-p8:@;)"
278: 457.02702628397947 "(58+692686.)+.p9:?;)"
280: 418.8725115002297 "(98+5726961)+,n8>73("
303: 416.5640775970895 "(99+6418960*(+o8<43-"
309: 410.9369027686399 "(98+8565850+()n9>50."
310: 408.6283272354027 "(97+7564861+()n9>50."
...
```

```
...
1176: 2.7057640843563036 "7+6773+420+(93497+9,"
1180: 2.645749153303157 "7+6773+420+(93497+9,"
1183: 2.5125838864933314 "7+6773+420+(93497+9,"
1184: 2.4164023100685768 "7+6773+420+(93497+9,"
1185: 2.3587784555340363 "7+6773+420+(93497+9,"
1188: 2.239741301366365 "7+6773+420+(93497+9,"
1189: 2.226287007742286 "7+6773+420+(93497+9,"
1190: 2.1885096509571014 "7+6773+420+(93497+9,"
1191: 1.924813066798606 "7+6773+420+(93497+9,"
1193: 1.8706175204849131 "7+6773+420+(93497+9,"
1194: 1.867332592042274 "7+6773+420+(93497+9,"
1195: 1.8038501729502485 "7+6773+420+(93497+9,"
1196: 1.5728463911464843 "7+6773+420+(93497+9,"
1203: 1.5092923931476605 "7+6773+420+(93497+9,"
1205: 1.132590624093993 "7+6773+420+(93497+9,"
1208: 1.1229937642522572 "7+6773+420+(93497+9,"
1209: 1.119287494716147 "7+6773+420+(93497+9,"
1215: 1.0790638124029215 "7+6773+420+(93497+9,"
1216: 0.0 "7+6773+420+(93497+9+"
```

How about symbolic evaluation (using SAT/SMT)?

- We can handle more complicated arithmetic
- We can abstract away completely over unknown functions
    - black-box
- Every run is an actual run
- State space exploration can explode
- Clearly a complement to random testing

finding bugs
in type systems

(demo)

https://ifc-challenge.appspot.com/

IFC CHALLENGE

The Information Flow Control Challenge consists of 10 challenges to leak the secret in the face of increasingly hardened information flow control mechanisms.

[Start the IFC Challenge]

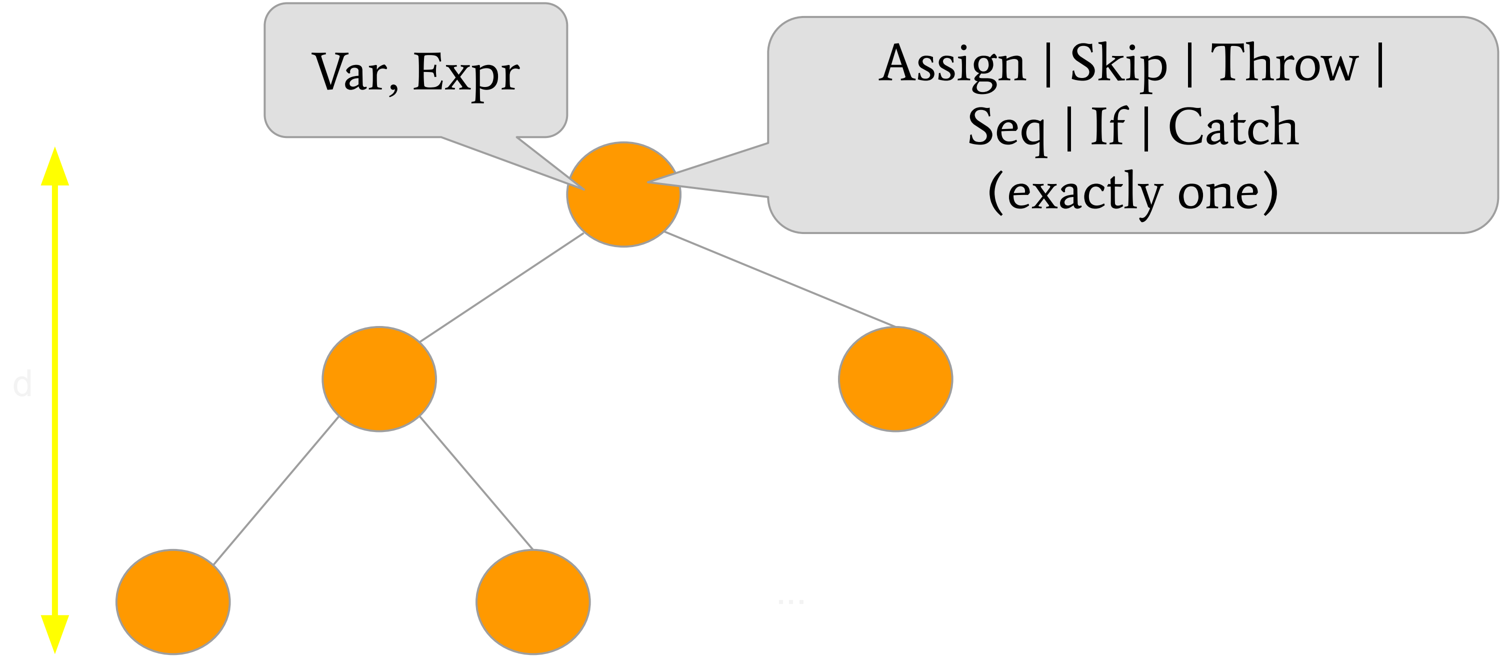
scalable

constraints & variables over finite domains

SAT

other theories via SMT

IFC Challenge in SAT (take 1)

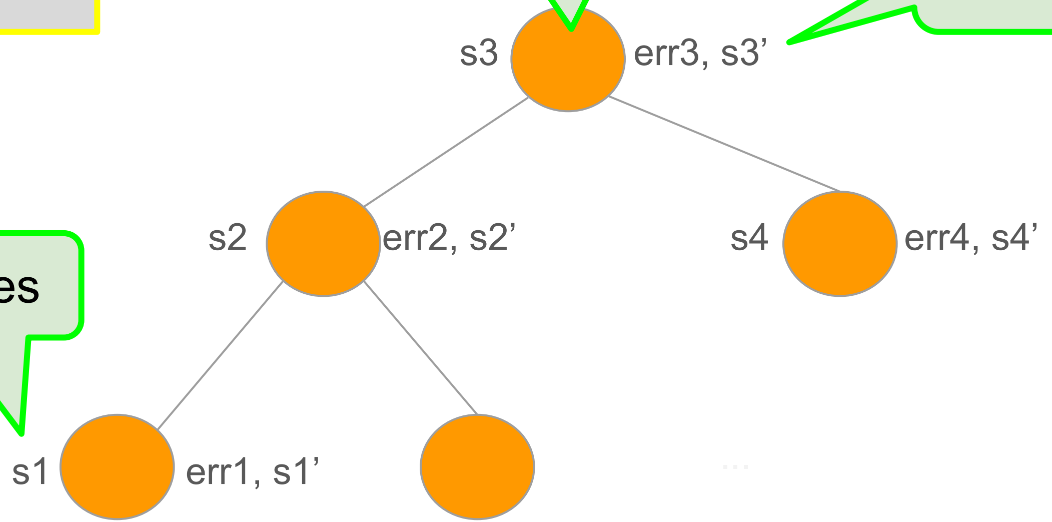


IFC Chal (take 1)

add constraints about s2,err2,s3,...,err4,s4’

1 run

say that s3’[l] = s3[h]



all variables

add for all runs

a few milliseconds

solve

IFC Challenge in SAT (take 1)

solver knows
about types

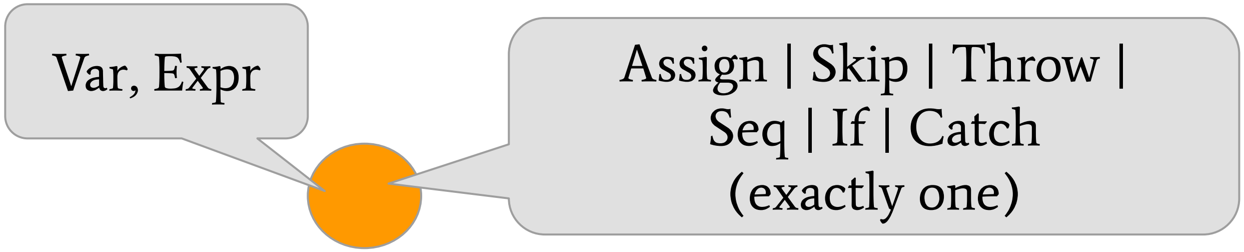
solver knows
about semantics

trees are
wasteful

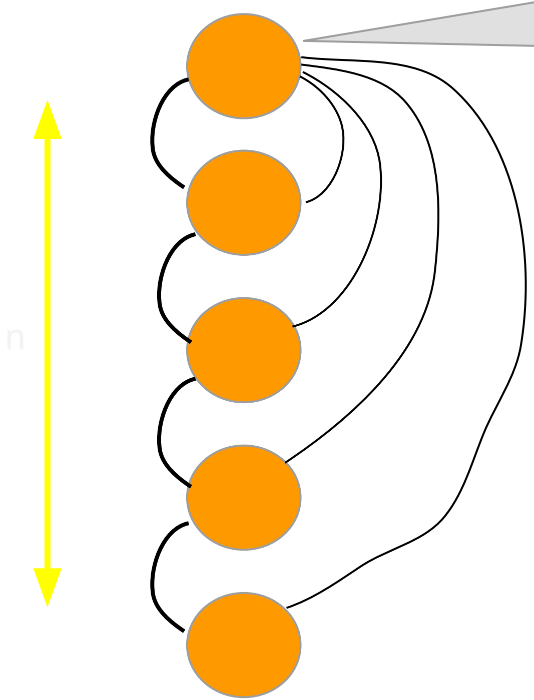
what about
loops /
recursion?

what about
non-finite
types?

Size-based tree encoding



Size-based tree encoding



extra variable "pointer"

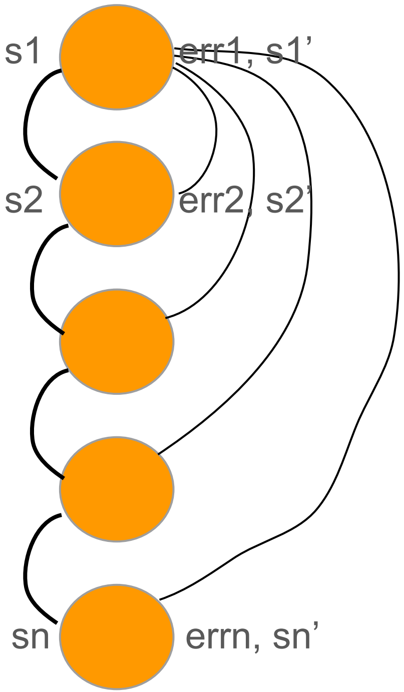
constraints based on **arities of constructors**

works well for arities <= 2

other constraints are a bit more complicated

SAT problem MUCH smaller, **more scalable**

Size-based tree encoding



constraints have to pick the right si'

Non-termination / loops

have a **maximum number of loop takes** for a semantic run

size of the program

number of variables

size of expressions

enumerate these in some way

maximum number of function calls

dependent on the input

Summary

- Works OK to find bugs in **type systems**

- hand-coded

- Problems with:
  - unbounded program execution
  - more complicated data types (e.g. lists)
  - scalability

Experiment - use Val + numerical optimiza

want to find bugs in
type checkers
(implementation)

- Similar coding of **type systems**
    - But, don't care about many details

- Still hand-coded

- Could find the same bugs as SAT, in similar running times
    - No problems with coding types, bounded running times. etc.

speculative part

Idea: **instrument a type checker with Val**

- Still, hand-coded
  - but systematic
- Have it compute the “distance” to when things go wrong
- Use numerical optimization

Idea: **instrument a type checker with Val**

- Still, hand-coded
  - but systematic

- Have it compute the “distance” to when things go wrong

- Use numerical optimization

new project starting 2026

NULL-pointer dereferencing

Numerical optimization methods for automated bug finding in software

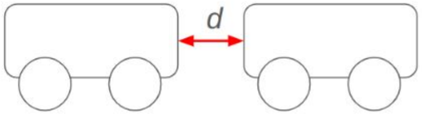
1. Purpose and aims

non-interference

...roject is about investigating how **numerical optimization methods** can be used to ... **automatic generation of test cases** in order to search for **difficult to find bugs** ...**tware**.

Earlier work. The inspiration for this project comes from our earlier work on specification and testing of hybrid systems [1]. In a hybrid system, *discrete* software interacts with ...*inuous* physics. An example is the adaptive cruise controller in a vehicle; the discrete ...ware reads inputs from the sensors (radar and speed) and provides control signals to the ...e; the engine and the cars on the road are described by continuous physics. A possible ...ty property is: the distance between our car and the car in front of ours should always be more than, say, 10 meters. How can we generate test data to test a property like that?

real-time constraints



Here is what we did: (1) Rather than expressing the safety property as a boolean statement that is merely true or false for any run of the system, we instead express the property as a real[1] number p that indicates *how close to being false* the property is. For example, for the cruise controller, we would simply use $p=d-10$ where d is the distance between the cars in meters. If $p>0$, the safety

1 PhD student

working on bug finding using numerical optimization

1 post-doc

working on programming language semantics