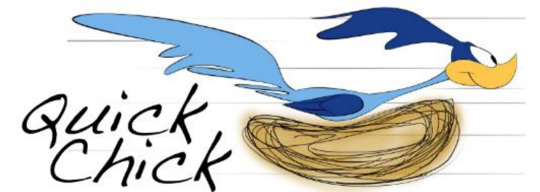
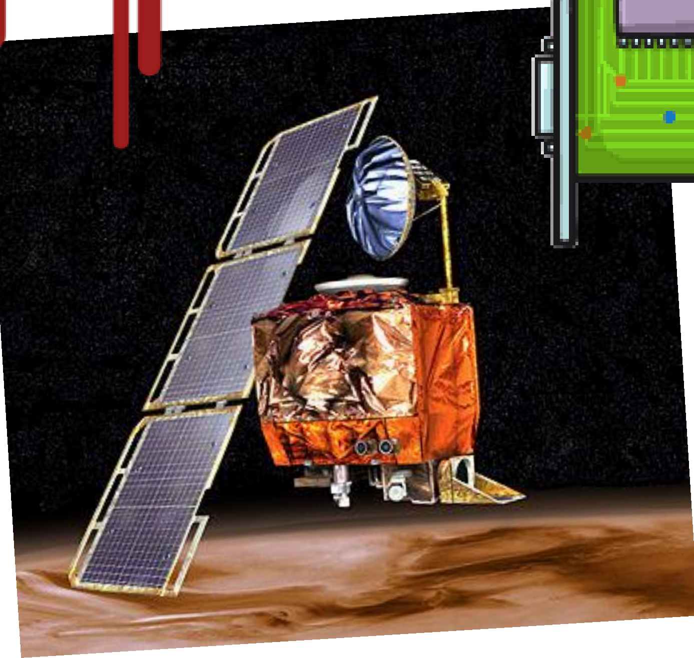
Software Correctness at Scale through Testing and Verification

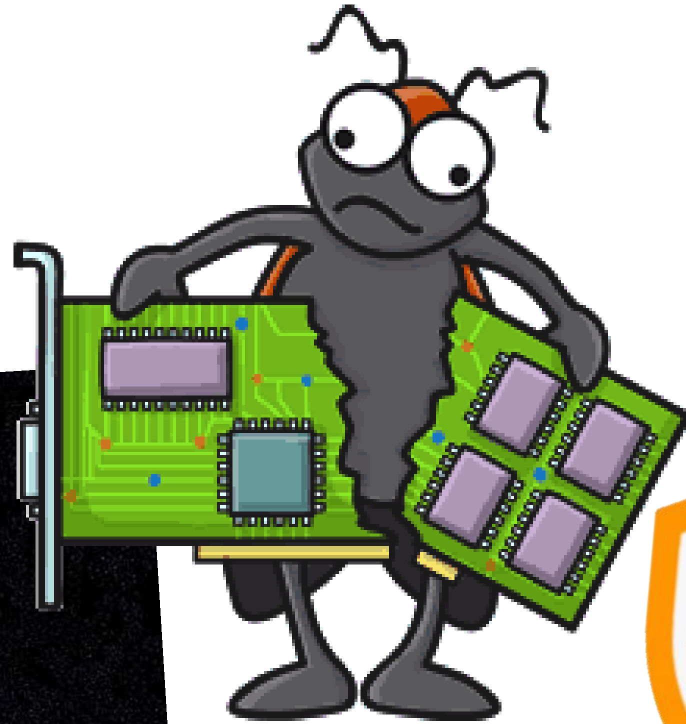


Leonidas Lampropoulos

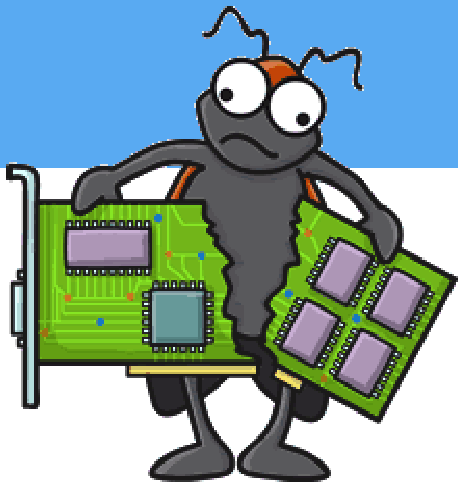
Research Overview

Bugs are everywhere





Bugs are everywhere



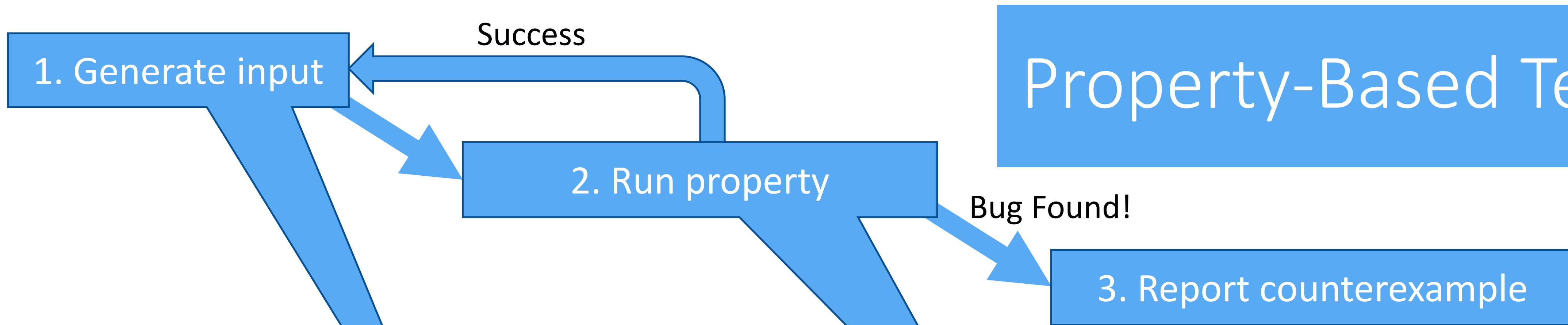
- How do we find such errors?
- How do we guarantee they don't exist?
- What constitutes an error in the first place?

Specifications!

```
Theorem sort_correct :
   forall (l : list int), is_sorted (sort l).
```

Specifications…
…state the intended behavior of programs
…can be *tested*
…can be *verified*

Property-Based Testing




```
Theorem sort_correct :
    forall (l : list int), is_sorted (sort l).
```

Specification

- ✓ Discover bugs quickly!
- × Does not guarantee correctness.

- Where do properties come from?
- How to generate inputs?
- How to minimize counterexamples?

Formal Verification

From handwritten proofs…

…to machine checked ones!

Q.E.D.

```
Theorem sort_correct :
    forall (l : list int), is_sorted (sort l).
```

Specification

✓ Formal guarantees!

× Expensive and time-consuming.

- How to discover invariants?
- How to structure proofs?
- Are the assumptions sound?

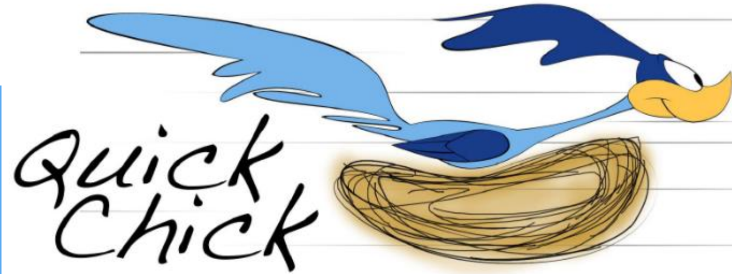
Writing, Debugging and Reasoning about Specifications

Property-Based Testing ✚ Formal Verification

Tightly integrated in a single framework!

- ✓Debug specifications quickly
- ✓Specifications provide the properties “for free”
- ✓Discover program invariants
- ✓Gain confidence in assumptions.

My prior work:



Property-based testing tool for Coq

- Proved correct [***ITP 2015***]
- Efficient and correct generation for constrained data [***POPL 2018***]
- Coverage-guided fuzzing [***OOPSLA 2019***]

… used in practice to facilitate verification
([***JFP 2016***], DeepWeb, Vellvm)

… taught in courses and summer schools
(UMD 631, DeepSpec 2017 and 2018)

SOFTWARE FOUNDATIONS
VOLUME 4

QuickChick

Property-Based Testing in Coq

Leonidas Lampropoulos

Benjamin C. Pierce

PHOTO: Benjamin C. Pierce

Prior and *Future* Work:

Verification

Testing

Checked C Core [POST 19]

Checked C at Scale

Testing Noninterference [ICFP 13, JFP 16]

Smarter Generation

Noninterference for Risc-V

FuzzChick [OOPSLA 19]

Deriving Executable Specifications

Generating Generators [POPL 18]

QuickChick [Software Foundations]

Testing Laziness [ICFP 18]

Coq vs Liquid Haskell [Haskell 17]

Urns [Haskell 17]

Verifying QuickChick [ITP 2015]

Luck [POPL 17]

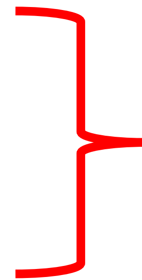
A Theory of Shrinking

Probabilistic Programming for Input Generation

Verify Neural Network Robustness [NIPS 16]

Security Application: Checked C

- Prior Work: Checked C [POST 19]
  - A safety proof for a core calculus, a subset of C
  - Medium sized (3500 lines).
  - Out of reach for current automated testing techniques:
    cyclic structures, complicated typing invariants
- Goal: Checked C at scale
  - Layer on top of CompCert, the verified C compiler
  - Type system rules out (spatial) memory errors
- Challenge: Much larger scale
  - Almost impossible to debug manually
  - Tricky invariants



Needs better automation!

Solution: Testing!

Security Application: IFC for RISC-V

- Prior Work: Testing Noninterference, Quickly [ICFP 13, JFP 16]
  - A testing framework for simple *information flow control* monitors
  - Required elaborate handwritten data generation strategies.
- Goal: Noninterference for RISC-V
  - Co-Processor spec on top of FORVIS, a RISC-V Haskell spec
  - Programmable security policies
- Challenge: Much larger scale
  - Much slower test throughput
  - Tricky invariants

Secret inputs don't influence public outputs!

Requirement: Improve the state of automated testing!

Automating Input Generation

Problem: (Sparse) Preconditions

$$\forall x. q(x) \rightarrow p(x)$$

1) Generate x

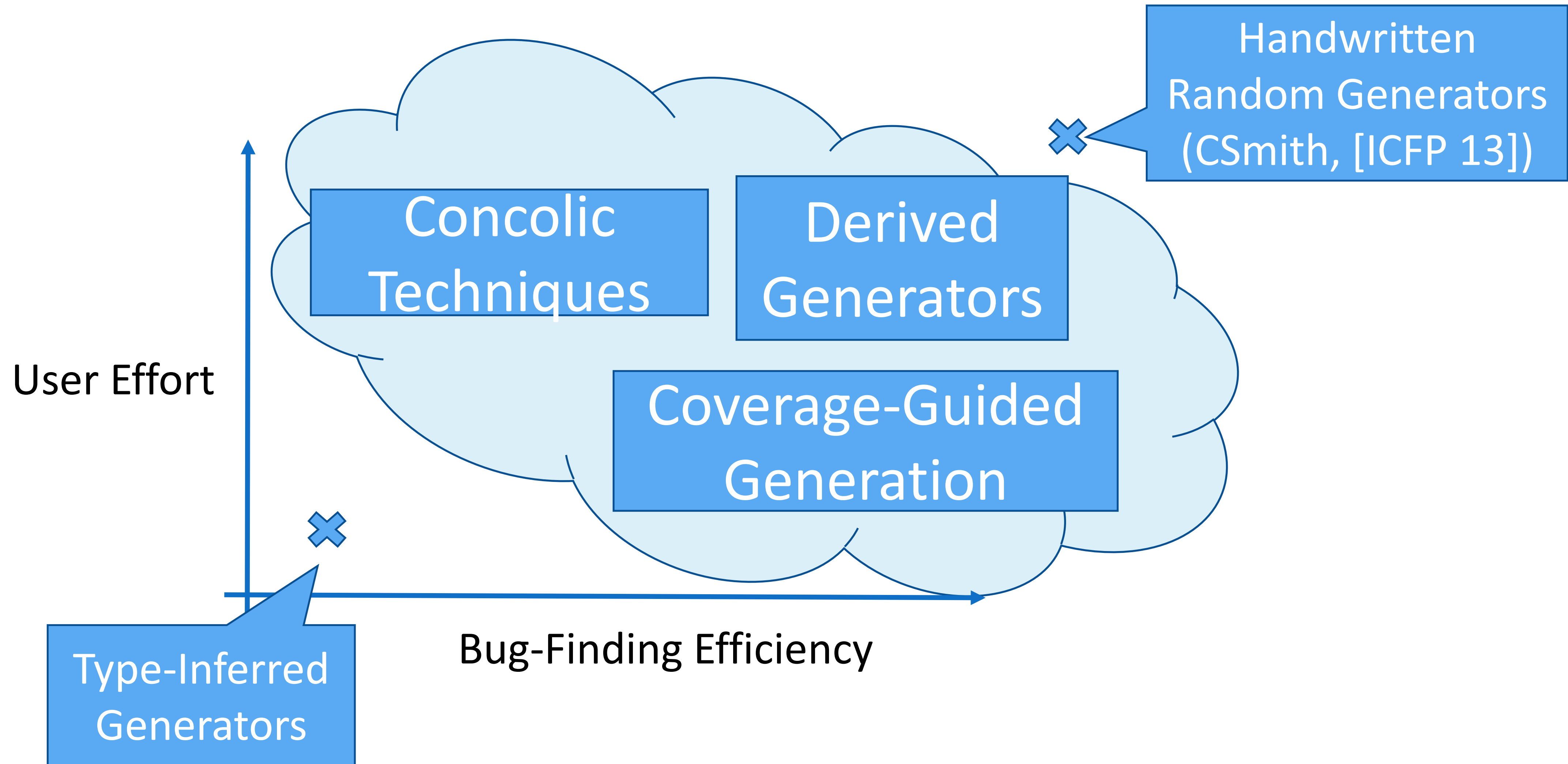
2a. Check $q(x)$

2b. Only if q holds, test $p(x)$

3. Minimize x, s.t. $q(x)$ && $\sim p(x)$

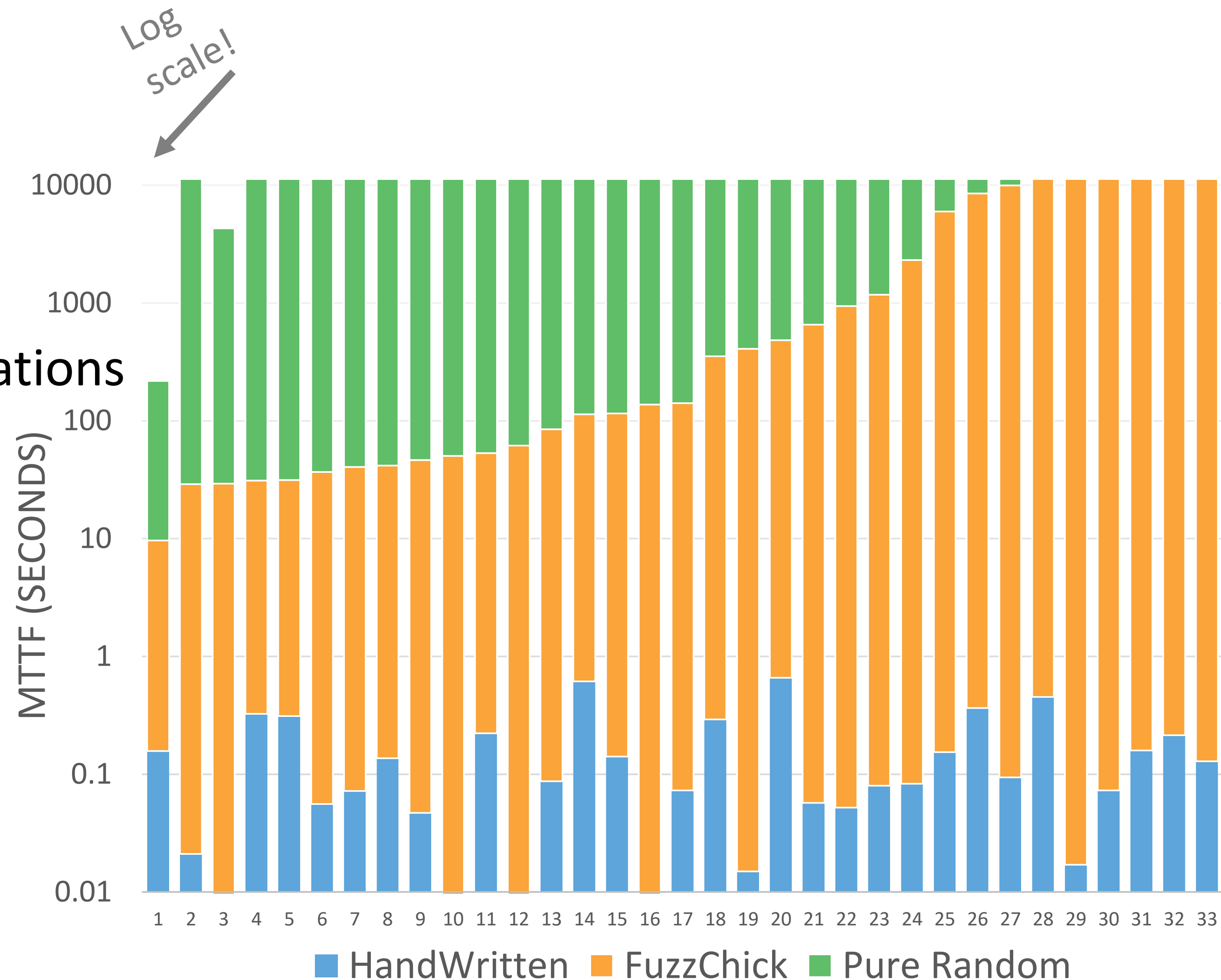
Solution: Automatic generation of data *satisfying q!*

Smarter Test Data Generation



Smarter Test Data Generation

- Prior Work: Derived Generators
  - Luck [POPL 17]:
  DSL for writing generators
  - Generating Generators [POPL 18]:
  Derive generators from inductive relations
- Ongoing: Coverage-Guided PBT
  - FuzzChick [OOPSLA 19]:
  Semantic mutations using coverage
  - Lots of promise and room for improvement!
- Future: Concolic PBT
- Future: Adaptive Choice Strategy



Automating Executable Specifications

Problem: Specifications in non-executable form

- Theorems expressed inductively
- Non-functional behaviors

  (nondeterminism, nontermination)
- Maintaining two versions of specifications is tedious and error prone

Solution: Extract computational content from inductive

- Translate inductive definitions to a *backtracking* decidability procedure
- Correct-by-construction with a proof certificate

Automating Counterexample Minimization

Problem: Report small counterexamples to users

- Current approach:

  Greedily try to minimize a counterexample to an immediately smaller failing one
- Challenges:

  Bug slippage – Transform a potentially critical bug to a simpler uninteresting one

  Deduplication – When multiple bugs are found, how many do we report?

Solution: Whitebox minimization techniques

- Concolic Minimization
- Deduplication by Clustering

More Future Threads

- Machine Learning Techniques for Testing:
  - Generation via sampling probabilistic programs
  - Reinforcement learning for weighing mutations
- Improving Correctness for Python:
  - Slow test execution
  - Shift balance to more symbolic techniques