Why Isn't Verification Standard Practice?

Steve Crocker, Shinkuro, Inc.

steve@shinkuro.com

Motivation

- Bad software is very expensive
  - Testing, patches, exploitation of flaws, etc.
- Lots of money being spent on bandages
- Culture assumes bad software is inescapable
  - **Let's change the ethic!**
- Programs should not leave the programmer's desk with elementary errors
- We should build tools that take care of most of the work. And it's doable.

Competing Approaches

- Post hoc analysis tools
- Better languages, advanced type theory
- Testing
- Tighter training and management
- Formal methods (program verification)

Competing Approaches

- Post hoc analysis tools
- Testing
- Tighter training and management
- ✓ Formal methods (program verification)

The "Easy" Stuff

- Buffer overflow and related errors
- Not handling all possible inputs
- Code injection
- Endless loops

There are many other properties of concern. This is just the underbrush

Why not Formal Methods?

- Simple bugs abound and are expensive.
- Big advances in formal methods
- Significant positive experiences

BUT

- Not part of the standard tool kit
- Not part of standard expectation

WHY?

Vision

- Easy Properties
- Ubiquitous
- Mature Annotation Language(s)
- Mature Checking Tools
- Algorithm Libraries

"Easy" Properties

- Data structure integrity
  - No buffer overflow, invariants for list structures, no memory leakage, etc.
- Clean Termination (no aborts)
  - Includes not running out of stack space, etc.
- Tight semantics
  - No dependency on compiler variations
  - No referencing of uninitialized variables or freed storage, etc.

(Information flow, correctness and performance come in increments later)

Ubiquitous

- Included with ordinary languages
  - C, Python, Javascript, etc., etc.
- Standard part of the tool chain
- Standard part of the language
- Standard part of the education
- Standard part of the software cycle
- Considered bad form if low-level bugs leave the programmer's desk

Annotations

- Annotation language must be easy to learn and use
- Annotation language must be powerful enough to express the necessary concepts
- Annotations should increase text <100%.

Annotation Language(s)

- Big learning curve for languages
  - Cf programming languages
- Details matter, e.g. "=" vs ":="; loop design
- What concepts?
  - Pre and post conditions? Invariants?
  - Sequences of values? Abstract data structures?
  - Sets? Bags? Etc?
- Experiment! Iterate!

Checking Tools

- 100% predictable and automatic
  - Not necessarily complete
- Fast
- Incremental
- Embedded in IDE and source code control system
- Might also include counter examples

Algorithm Libraries

- Separate algorithm proofs from code proofs
- Capture the literature
  - Math proof systems, e.g Coq, PVS, ACL2
  - Visualizations
- Connect with "stitching"
  - Import algorithm into code
  - Match up points in code with way points in algorithm

Summary of Attributes

Properties	"Easy properties" • Conservative language semantics • Integrity of data structures • Clean termination
Programming Languages	Every language(!) – C, Python, Javascript, etc.
Assertion Languages and Assertions	• Easy to learn and use • Past, present, future states • Between 10% and 100% additional text • Enjoyable(!), useful
Position in the tool chain	IDE and Source Code Control System
Algorithms	• Separate system • Library of algorithms and proofs • Import into source code
Solvers	• Predictable even if not complete • Fast

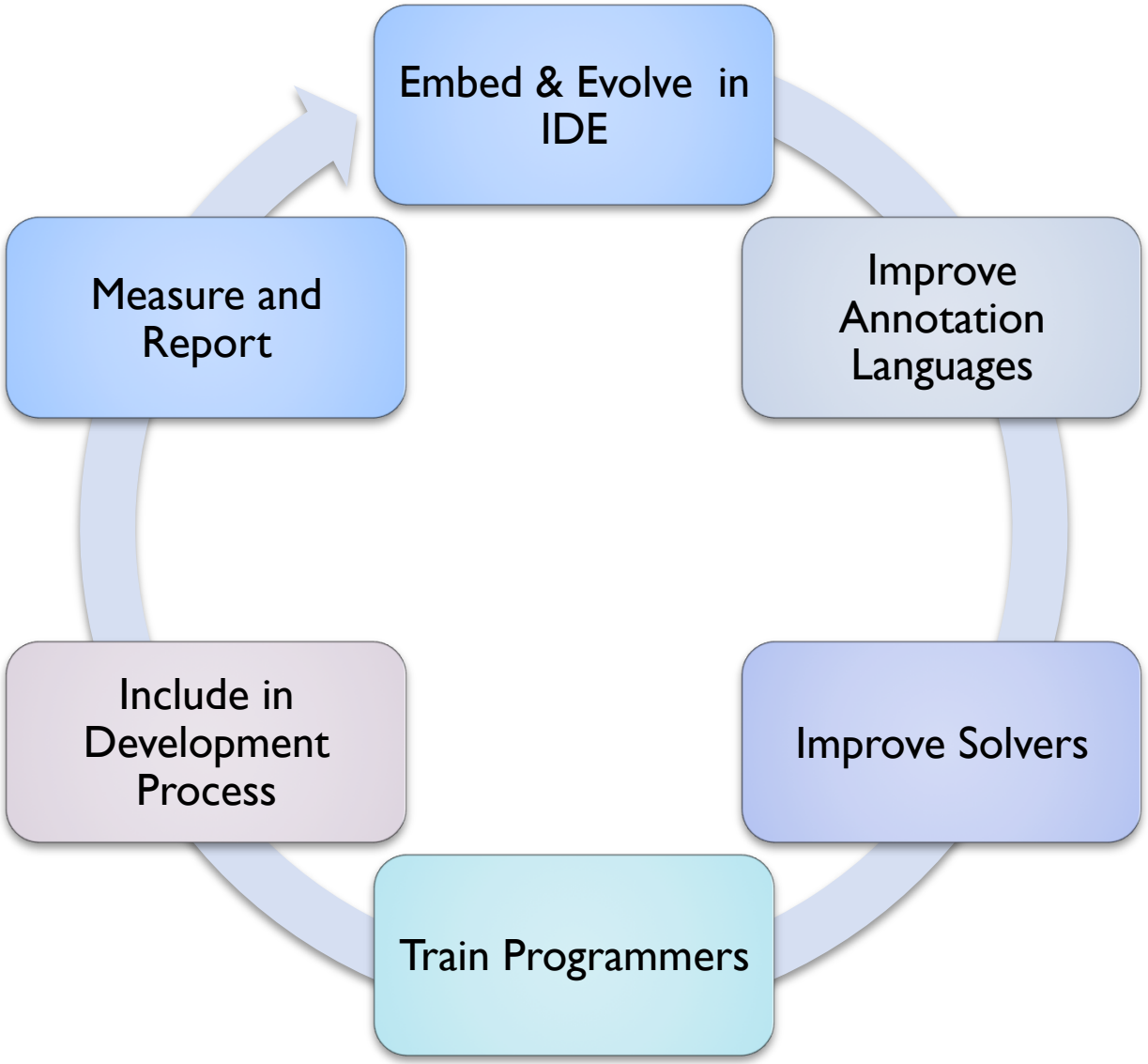
The Hurdles

- Automatic is preferred
  - No annotations; nothing to learn
- Languages for annotations, assertions, proofs still in early stages.
- Culture doesn't believe in bug-free software
- Proof tools too hard to use
- Belief we need cleaner languages, Ada vs C
- Too time-consuming or costly

The bigger context

- In practice, little control over tools
  - Delivered code
- Need to add meta info – provenance, properties, etc.
- GIT?

Building a Virtuous Cycle



Thanks!