Analysis is necessary – but far from sufficient

Jon Pincus
Reliability Group (PPRC)
Microsoft Research

Why are so few successful real-world development and testing tools influenced by academic research?

My definition of "real world":
- Commercial or quasi-commercial
- Software and net services

(Excluding IT or Mil-Aero – I have no background there)

Outline

- What makes a tool successful?
- Characteristics of successful tools
- Analysis – in context
- Implications for analysis
- Summary

Success!

- Cute diagram

Success: a real-world view

- A tool is successful if people use it
  - Not if people buy it but don't use it ("Shelfware")
  - Not if people try it but don't use it

Some examples of success

(drawn from *defect detection* space, because that's my background)

- Purify
- BoundsChecker
- PREfix (2.X and later)
  - Especially interesting because 1.0 was unsuccessful

Why do people use a tool? If

- it helps them get their work done …
- … more efficiently than they would otherwise
- … without making them look bad.

- Think in terms of
  - Goals
  - Value proposition

Goals

- *Organizational* goals
  - "compensate for not being able to find enough good developers/QA engineers"
  - "get higher-quality software to market"
  - "get high-quality software to market more quickly"
  - "avoid the memory leaks that plagued our last release"
  - "stop breaking the build"
- *Personal* goals
  - "stop having to waste my time on others' mistakes"
  - "find that killer bug"
  - "stop getting blamed for breaking the build"

Three definite non-goals

- Looking stupid
- Not being able to use techniques I already know
- Creating additional work

For more on goals: Alan Cooper's *About Face*
http://www.cooper.com/books/01_goal_directed_design.html

An example

- People want to fix the key defects as easily as possible
- PREfix 1.0: "detects defects"
  - Too far from goal to be generally useful
  - Not successful [although it found a lot of real defects]
- PREfix 2.X and later: focus on *prioritizing* and *understanding* defects
  - Successful (although fixing them would be even better)
  - Techniques: focus on user interaction, data storage, repository, understandability, prioritization, …

Look at the *value proposition*

- (Value – Cost) must be
  - Positive
  - More positive than any alternatives
- Initially, cost will exceed value; how long until payback?
- Value: the benefit of the tool
  - E.g., higher quality
- Cost: more complex

Cost

- Time investment
  - Initial use
  - Steady-state use
  - Training
    - Existing employees
    - New employees
- Process changes
- Licensing cost
  - Zero for "free" software
  - Typically much smaller than the others

An example

- Purify 1.0:
  - Virtually zero initial cost on most code bases
  - Immediate value
  - Companies invested (substantially!) to leverage
    - E.g., changing memory allocators to better match Purify's

An example

- PREfix 1.0:
  - Major initial cost
  - Usability issues increase ongoing costs
  - There's value; but nobody got to sustained value
- PREfix 2.X:
  - Lowered initial cost
  - Improved usability decreased steady-state and training cost
  - Value – to people who care a lot about reliability
- Ongoing work: decrease costs further

Outline

- What makes a tool successful?
- Characteristics of successful tools
- Analysis – in context
- Implications for analysis
- Summary

Characteristics of successful tools

- Successful tools
  - address significant problems,
  - on real code bases,
  - give something for (almost) nothing,
  - and are easy to use.

Significant problems

- Nobody fixes all the bugs. What are the key ones?
  - Often based on most recent scars
  - Often based on development or business goals

Examples: significant problems

- Purify: memory leaks
- BoundsChecker: bounds violations
- PREfix: defects not found by existing tests
- Lint (back in K&R days): portability issues

Example: insufficiently significant problems

- Vanilla lint (today): ???

  (Note: PC-Lint/FlexeLint extend Lint to attack today's problems)
- Pointer analyses for its own sake

  (although it may be useful for solving another problem)
- C/C++ metrics tools

Real code bases

- Usually large (1M+LOC) or very large (10M+ LOC).
  - In some areas, "reality" is smaller – e.g., 100s of lines of DHTML/JSscript
- In nasty languages (e.g., C/C++),
  - using nasty features (e.g., casts between pointers and ints, unions, bit fields, gotos, …)
  - with nasty extensions (GCC, MS)
  - and non-ANSI-compliant code (GCC, Sun, MS)

Examples: insufficiently real code bases

- "For a subset of C, excluding pointers, structs, and unions …"
- "Assuming the whole program text is available (i.e., there are no calls to system libraries) …"
- "We have tested on our approach on programs up to several thousand lines of Scheme …"

A "fully general solution" includes

(Please assume appropriate trademark/copyright symbols)

- Perl
- C [K&R, ANSI], C++ [Cfront, GCC extensions, MSVC extensions], Java, C#
- VB, TCL
- ECMAScript [JScript, JavaScript], VBScript, Python
- HTML [HTML3.2, Netscape/MS variants], DHTML
- SQL
- XML, XSL, XSL-T, XML Schemas
- FORTRAN, COBOL for legacy code
- Make, sh, InstallShield, IDL, Excel macros, …

Something for (almost) nothing

- Engineering time is the single most critical resources at most (successful) companies
- Engineers need to be convinced before investing non-trivial amounts of time
- So don't even *think* about *requiring* significant up-front investment
  - code modifications
  - process changes

Examples: something for (almost) nothing

- Purify for UNIX: just relink
- BoundsChecker: you don't even need to relink!
- PREfix 3.5: just type "prefix"
  - (Uh, most of the time, anyhow)
- A non-technology solution: "we'll do it for you"
  - Commercial variant: an initial benchmark for $X, money back if it doesn't work
  - In many cases, money may be cheaper than engineering time …

Examples: too far from nothing

- “Once the programmer modifies the code to include calls to the appropriate functions …”
- “The programmer simply inserts the annotations to be checked as conventional comments …”
- PREfix 1.0: “the following changes to your build process may prove necessary …”

Ease of use

- Admittedly, the bar is low here …

[not sure exactly what to say; it seems so self-evident …]

Examples: ease of use

- [use an example from the CAD space. Place and route is highly algorithmic; but these days, it’s ease of use which gets the best results – because engineers can make one more iteration in the time allotted for a benchmark]

Outline

- What makes a tool successful?
- Characteristics of successful tools
- Analysis – in context
- Implications for analysis
- Summary

Counterintuitively …

Actual analysis is only a small part of any program analysis tool.

In PREfix, < 10% of the “code mass”

PREfix architecture

- Pretty picture here

PREfix' key operations

- Parsing
- Calculating function dependencies
- Walking paths through functions
- Tracking memory during simulation
- Generating and storing models
- Generating and storing defect information
- Viewing/sorting/filtering sets of defects
- Viewing paths through source code
- Build integration

3 key non-analysis issues

- User interaction
  - Information presentation
  - Navigation
  - Control
- Integration
  - Build process
  - Defect tracking system
  - SCM system
- Parsing

User interaction

- Engineers must be able to use the results of the analysis
  - Understanding individual defects
  - Prioritizing, sorting, and filteng sets of defects
  - Interacting with other engineers
  - Controlling the analysis (because analyses aren't perfect)
- Today, the bar is ridiculously low
  - A good place to make progress!

Example

- [single-line Dereferencing NULL Pointer message]

A better example

- [Complex code path failing to check new]

A still better example

- Message describing the problem

Noise

- [definition of noise]
- [deal with it at analysis level, or at rest of system level?]

Some interesting questions …

- How to summarize information usefully?
- How to visualize (sets of) (partial) paths through code?
- Can analysis refine presentation?

Integration

- A tool is useless if people can't use it
  - Implied: "use it *in their existing environment*"
- "Environment" includes
  - Configuration management (SCM)
  - A build process (makefiles, scripts, …)
  - Policies
  - A defect tracking system
- People have invested *a lot* in their environment
  - They probably won't change it just for one tool

Approaches to integration

- Special-case methods seem the norm
  - E.g., "intercepting" build commands; special purpose scripts
  - PREfix' latest attempt:
    - treat build information as first-class data – tracked in database, used in analyses, ...
    - move to more general "repository"
- Can this be generalized?
- Is there a better way?

Parsing

- You can't parse better than anybody else …

  … but you *can* parse worse
- Complexities:
  - Incompatibilities
  - Extensions
  - Full language complexity
  - Language evolution

Approaches to Parsing

- Don't
  - Alternatives: EDG, GCC, Jikes, …

Outline

- What makes a tool successful?
- Characteristics of successful tools
- Analysis – in context
- Implications for analysis
- Summary

Characteristics of useful analyses

- Scalable to large system
  - Typically implies *incomplete, unsound, incremental,* and/or *very simple*
- Produce information usable by typical engineer
  - If there's a violation, where? How?
  - Post-processing output can be useful
  - Remember: half the engineers are below average
- "Accurate enough" for the particular task
- Handle full language complexity
  - Or can compensate for unhandled constructs

Different tradeoffs from compilers

- Focus on information, not just results
  - Compilers don't have to explain what they did and why
- Incompleteness and unsoundness may be okay
- Intra-procedural analysis often not enough

A spectrum of analyses

[make this a chart?]

- Flow- and context-insensitive:
  - Typically scales well.
  - Not particularly accurate (but clearly accurate enough for some tasks)
  - Often hard to understand or prioritize the output [no path information; no callstack]
- Flow- and context-sensitive
  - Scaling problems.
  - More accurate; still issues with path-sensitivity
  - Info may be more understandable
- Path-sensitive
  - Non-PREfix Examples?

Examples of analysis tradeoffs

- Purify/BoundsChecker: very simple
- PREfix: incomplete, somewhat unsound, incremental
- Lint (without post-processing): not "accurate enough", often not usable by typical engineer
- PREfast: consciously tradeoff completeness for performance

Some interesting questions …

- Which analyses are right for which problems?
  - No such thing as the right pointer analysis – it depends what you want to do with the results
- Are there opportunities to combine analyses?
  - Can we use a cheap flow-insensitive algorithm to focus a more expensive algorithm on juicy places?
  - Can we use expensive local path-sensitive algorithms to improve global flow-insensitive algorithms?

Outline

- What makes a tool successful?
- Characteristics of successful tools
- Analysis – in context
- Implications for analysis
- Summary

Summary

- People use tools to accomplish their tasks
- Successful tools must
  - address real problems,
  - on real code bases,
  - give something for (almost) nothing,
  - and be easy to use
- Analysis is only one piece of a tool
- Information is useless if it's not presented well

Why are there so few successful real-world programming and testing tools based on academic research?

These are not where research has focused.

Can – and should –that change?

Questions?

Analysis is necessary – but far from sufficient

Jon Pincus
Reliability Group (PPRC)
Microsoft Research