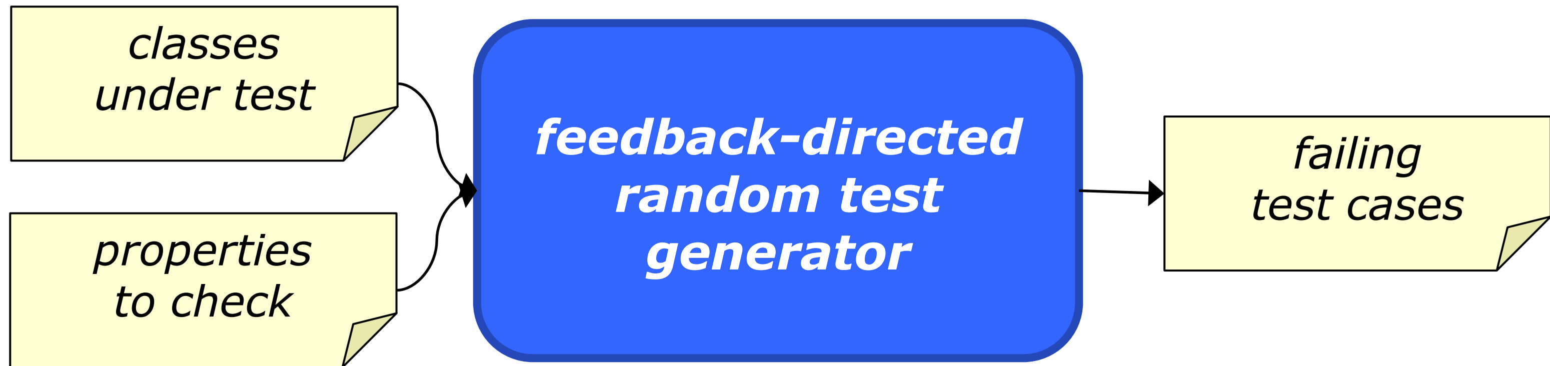
Finding Errors in .NET with Feedback-Directed Random Testing

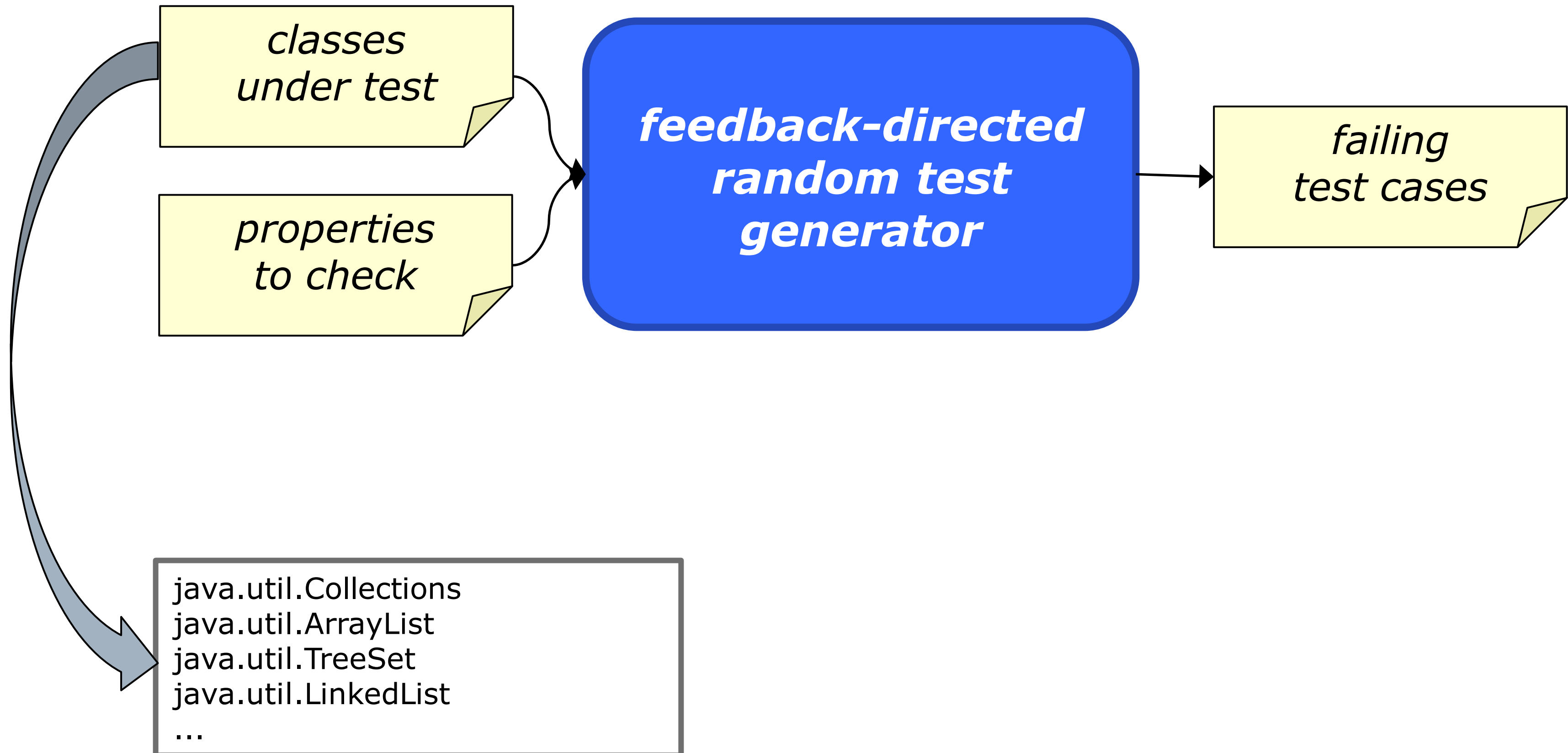
Carlos Pacheco (MIT)
Shuvendu Lahiri (Microsoft)
Thomas Ball (Microsoft)

July 22, 2008

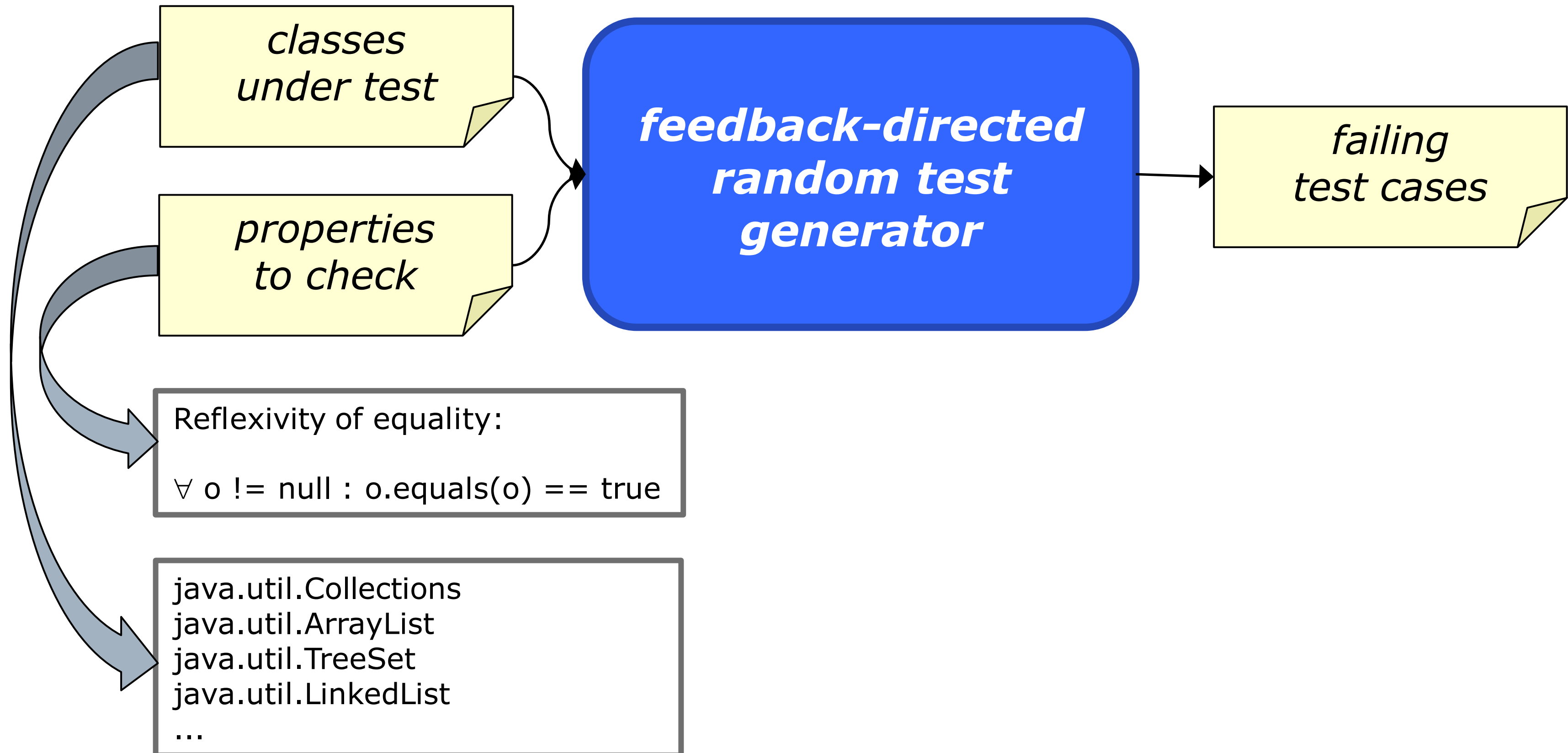
Feedback-directed random testing (FDRT)



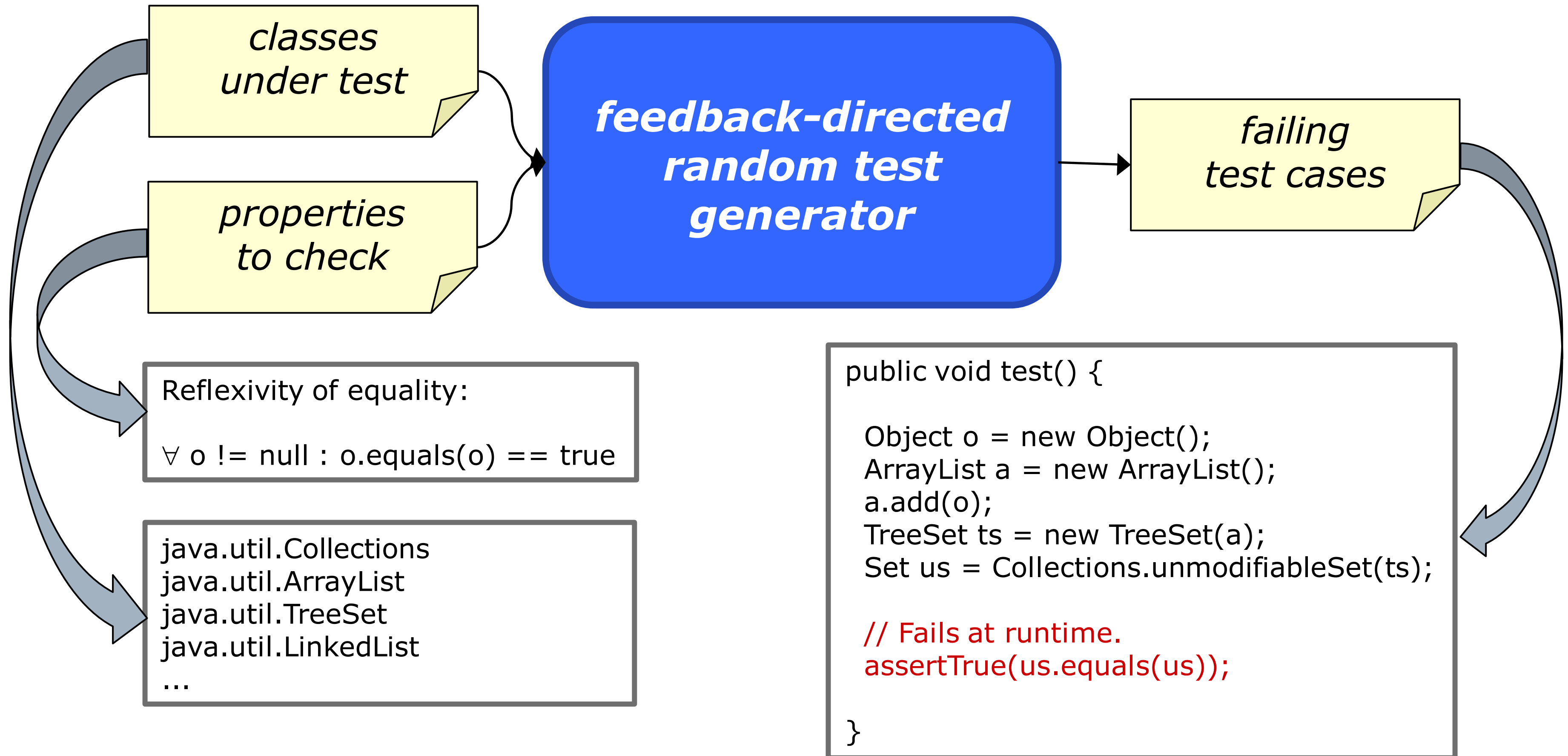
Feedback-directed random testing (FDRT)



Feedback-directed random testing (FDRT)



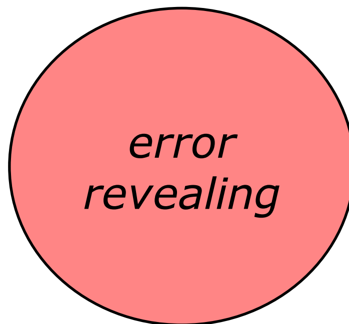
Feedback-directed random testing (FDRT)

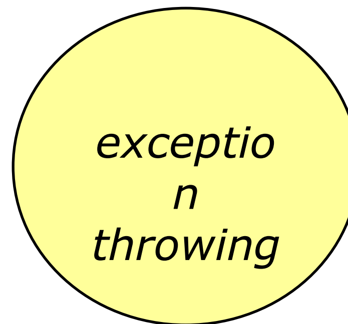


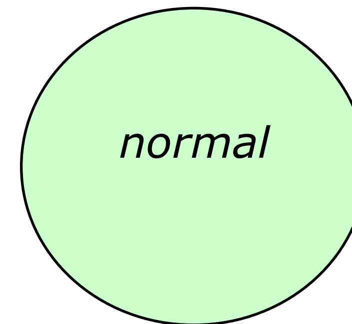
Feedback-Directed Random Test Generation
Pacheco, Lahiri, Ball and Ernst
ICSE 2007

Technique overview

- Creates method sequences ***incrementally***
- Uses ***runtime information*** to guide the generation







output as tests *discarded* *used to create larger sequences*

- Avoids illegal inputs

Prior experimental evaluation (ICSE 2007)

- Compared with other techniques
  - Model checking, symbolic execution, traditional random testing
- On collection classes (lists, sets, maps, etc.)
  - FDRT achieved equal or higher code coverage in less time
- On a large benchmark of programs (750KLOC)
  - FDRT revealed more errors

Goal of the Case Study

- Evaluate FDRT's effectiveness *in an industrial setting*
  - Error-revealing effectiveness
  - Cost effectiveness
  - Usability
- These are important questions to ask about any test generation technique

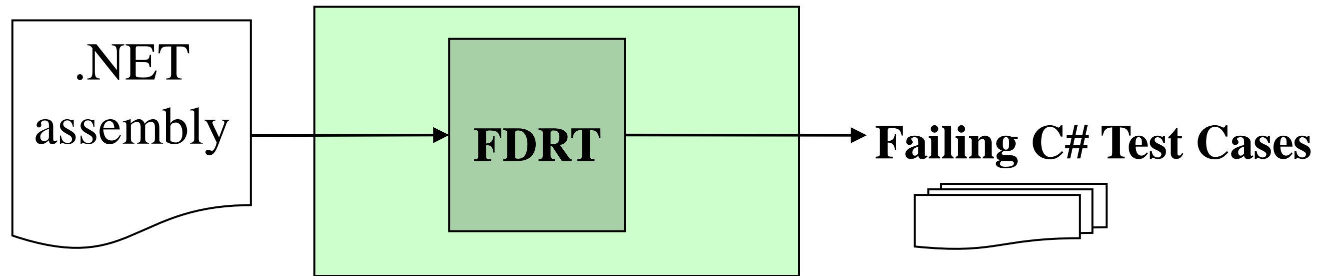
Case study structure

- Asked engineers from a test team at Microsoft to use FDRT on their code base over a period of 2 months.

- We provided
  - A tool implementing FDRT
  - Technical support for the tool (bug fixes bugs, feature requests)

- We met on a regular basis (approx. every 2 weeks)
  - Asked team for experience and results

Randoop



- Properties checked:
  - sequence does not lead to runtime assertion violation
  - sequence does not lead to runtime access violation
  - executing process should not crash

Subject program

- Test team responsible for a critical .NET component

  100KLOC, large API, used by all .NET applications

- Highly stable, heavily tested
  - High reliability particularly important for this component
  - 200 man years of testing effort (40 testers over 5 years)
  - Test engineer finds 20 new errors *per year* on average
  - High bar for any new test generation technique

- Many automatic techniques already applied

Discussion outline

- Results overview
- Error-revealing effectiveness
  - Kinds of errors, examples
  - Comparison with other techniques
- Cost effectiveness
  - Earlier/later stages

Case study results: overview

Human time spent interacting with Randoop	15 hours
CPU time running Randoop	150 hours
Total distinct method sequences	4 million
New errors revealed	30

Error-revealing effectiveness

- Randoop revealed 30 new errors in 15 hours of human effort.

  (i.e. 1 new per 30 minutes)

  This time included:

  interacting with Randoop

  inspecting the resulting tests

  discarding redundant failures

- A test engineer discovers on average 1 new error per 100 hours of effort.

Example error 1: memory management

- Component includes memory-managed and native code
- If native call manipulates references, must inform garbage collector of changes
- Previously untested path in native code reported a new reference to an invalid address
- This error was in code for which existing tests achieved 100% branch coverage

Example error 2: missing resource string

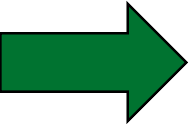
- When exception is raised, component finds message in resource file
- Rarely-used exception was missing message in file
- Attempting lookup led to assertion violation

- Two errors:
  - Missing message in resource file
  - Error in tool that verified state of resource file

Errors revealed by expanding Randoop's scope

- Test team also used Randoop's tests as ***input*** to other tools
- Used test inputs to drive other tools
- Expanded the scope of the exploration and the types of errors revealed beyond those that Randoop could find.
  - For example, team discovered concurrency errors this way

Discussion outline

- Results overview
- Error-revealing effectiveness
  - Kinds of errors, examples
  
  - Comparison with other techniques
- Cost effectiveness
  - Earlier/later stages

Traditional random testing

- Randoop found errors not caught by fuzz testing
- Fuzz testing’s domain is files, stream, protocols
- Randoop’s domain is method sequences
- Think of Randoop as a ***smart*** fuzzer for APIs

Symbolic execution

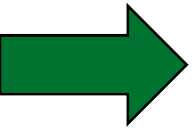
- Concurrently with Randoop, test team used a method sequence generator based on symbolic execution
  - Conceptually more powerful than FDRT

- Symbolic tool found no errors over the same period of time, on the same subject program

- Symbolic approach achieved higher coverage on classes that
  - Can be tested in isolation
  - Do not go beyond managed code realm

Discussion outline

- Results overview
- Error-revealing effectiveness
  - Kinds of errors, examples
  - Comparison with other techniques



Cost effectiveness

  - Earlier/later stages

The Plateau Effect

- Randoop was cost effective during the span of the study
- After this initial period of effectiveness, Randoop ceased to reveal errors
- After the study, test team made a parallel run of Randoop
  - Dozens of machines, hundreds of machine hours
  - Each machine with a different random seed
  - Found fewer errors than it first 2 hours of use on a single machine

Overcoming the plateau

- Reasons for the plateau
  - Spends majority of time on subset classes
  - Cannot cover some branches
- Work remains to be done on new random strategies
- Hybrid techniques show promise
  - Random/symbolic
  - Random/enumerative

Conclusion

- Feedback-directed random testing
  - Effective in an industrial setting
- Randoop used internally at Microsoft
  - Added to list of recommended tools for other product groups
  - Has revealed dozens more errors in other products
- Random testing techniques are effective in industry
  - Find deep and critical errors
  - Scalability yields impact

Randoop for Java

- Google “randoop”
- Has been used in research projects and courses
- Version 1.2 just released