Evaluating Program Analysis and Testing Tools with the RUGRAT Random Program Benchmark Application Generator

Ishtiaque Hussain,
Christoph Csallner
University of Texas at Arlington
Arlington, TX 76019, USA

Mark Grechanik
Accenture Technology Labs
and University of Illinois
Chicago, IL 60601, USA

Chen Fu, Qing Xie
Accenture Technology Labs
Chicago, IL 60601, USA

Sangmin Park
Georgia Institute of
Technology
Atlanta, Georgia 30332, USA

Kunal Taneja
Accenture Technology Labs
and North Carolina State
University
Raleigh, NC 27606, USA

B. M. Mainul Hossain
University of Illinois at Chicago
Chicago, IL 60607, USA

15 July, 2012

Motivation

- **Benchmark** is a point of reference from which measurements can be made to evaluate the performance of hardware or software or both[1]
- It is important as organizations and companies use different benchmarks to evaluate and choose mission-critical software for business operation, e.g., US-Dept. of Defense acquisition process[2]
- Companies spend 3.4% - 10.5% of their revenue in technology; biased or poor benchmark that leads to wrong decision

[1] G. McDaniel. *IBM Dictionary of Computing. McGraw-Hill, Dec.*1994

[2] Role of application benchmarks in the DoD HPC acquisition process. U.S. Army Engineer R&D Center, ERDC MSRC Resource, 2005

[3] K. S. Nash. Information technology budgets: Which industry spends the most? Nov 2007

Motivation

- Benchmarks are used to evaluate p**R**ogram **A**nalysis and **T**esting (**RAT**) tools
  - How scalable RAT tools are
  - How fast RAT tools get coverage
  - How through RAT tools evaluate different language features
- Such benchmarks are difficult to find
  - Not many benchmarks match all different constraints
  - Custom built benchmarks are often biased and reproducibility of results are difficult
  - Existing third-party benchmarks are often hard to install because of their external library dependencies

Motivation

- Benchmarks must be:
  - Neither too simple nor too complex to work with
  - Publicly available, reproducibility of results should not be an issue
  - Not laborious to build and should be cost effective

Solution: RUGRAT

Random **U**tility **G**enerator for p**R**ogram **A**nalysis and **T**esting

- **RUGRAT** automatically creates random applications that match your criteria
  - Developers configure what properties they want in benchmark applications
  - RUGRAT is scalable to generate very large benchmark apps (e.g., 10MLOC)

Overview

- Configuration Options
- Implementation
- Case study
- Experiment
- Related Work
- List of things available in the tool website

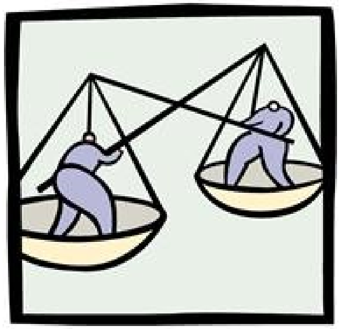
Configuration Options

- Many configuration options or parameters to RUGRAT
- Example values for these parameters or config. files used in the experiments are available at: **www.rugrat.ws**
- Many parameters are inter-dependent (e.g., total # classes ≥ # classes to populate an inheritance tree of a desired depth)
- Many parameters have maximum and minimum values
- Once these limits are defined, RUGRAT randomly chooses values from each range

List of Major Parameters for Current Prototype That Generates Java Apps

1. total LOC
2. # classes
3. class name prefix
4. # fields/class
5. # meth./ class
6. # param./meth.
7. # interfaces
8. # methods/interface
9. # interface a class explicitly implements

10Inheritance depth
11# inheritance chain
12allowed #meth. calls from a meth
13allow array?
14upper limit of array size
15iteration (for loop) upper limit
16allow indirect recursion?
17allow recursion?
18recursion depth
19nested if depth
20max. int value
21meth. call type (local or across-class calls)
22allowed types: int, float …
23etc.

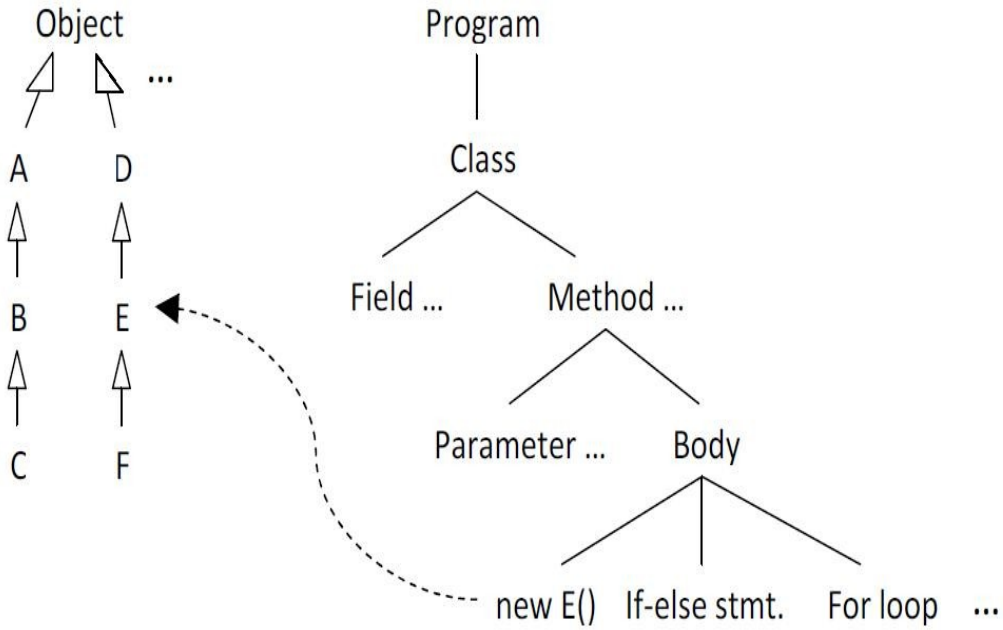


Standard Values

- Default range is defined based on empirical data from observed averages in Java projects:
  - Zhang et al. [APSEC’07]: #classes = LOC/ 114
  - Collberg et al. [SP&E’07]: each package has 12 classes, 1 interface per package: # interface = #classes/ 12
  - Collberg et al.: 96% programs have: <20 class fields 99% programs have: <60 #meth/class
  - Grechanik et al. [ESEM’10]: max. #meth/interface = 3.4

Implementation

- View language definition rules as production rules
- Start from the AST root node and keep instantiating production rules
- Randomly choose from multiple non-terminals
- Ignore non-terminals when limit is reached
- Choose only terminals when target LOC is reached



Example snapshot of RUGRAT’s program generation process

Random Program Generation is Simple but Not Enough

- Many features in modern OOP languages impose additional well-formedness rules. For example:
  - a method can only be called if it’s visible from the call site
  - for Java, no multiple-inheritance is allowed
  - a *final* field must be initialized, directly or by constructor
  - no *non-static* field should be referenced in a *static* method without initializing an object.
  - generated non-abstract class must

Desired Properties in Generated Programs

- No compiler errors can still lead to runtime errors:
  - generated expressions should not have runtime exceptions, e.g., divide-by-zero
  - recursion should be controlled to avoid heap exhaustion
  - indirect recursion can lead to heap exhaustion, too, e.g., methA --> methB --> methA

Solution: Internal tables, maps are used to maintain well-formedness rules

Limitations of the current RUGRAT Prototype Implementation

- Available prototype implemented in Java 6, creates only Java programs
- Only primitive types are used as fields in a class
- No Java library method calls are made
- Java generics are not supported
- No ‘do-while’ or ‘switch’ statement is generated
- Only single threaded programs generated

✓Future work (**RUGRAT4Load**) will implement other features, e.g., network

Case Study on a Loader

- **Problem**
  - A loader fetches code into the main memory from a secondary storage; this loader by a Fortune 100 company was written in C++ back in ‘70s
  - Bug in fetching > 3MLOC C++ code, took too long to fetch
  - Client code exposing the bug could not be shared for privacy issues

Case Study on a Loader…

- **Attempts to find/fix the bug**
  - 5 engineers spent 3 weeks to find the bug (600 man hour ≈ $35K)
  - Repeated attempts in different subject applications failed to reproduce the bug
  - Bug was in Hash function that takes 128 character prefix of the access path of identifiers as input
  - Overtime same 128 char. prefix of tens of thousands of identifiers put into the same bucket reduced the Hash table to a linked list
- **RUGRAT-C++**
  - A separate RUGRAT prototype (**RUGRAT-C++**) that generates C++ programs reproduced the bug in < 4 hours !

Experiments

- **RQ1:** How similar are RUGRAT-generated applications to third-party applications?

- **RQ2:** How do RAT-tools behave while analyzing RUGRAT-generated applications?

- **RQ3:** Can RUGRAT-generated applications find defects in RAT-tools?

RAT = p**R**ogram **A**nalysis and **T**esting

Experiment Setup

- We ran all experiments on a HotSpot 1.6.0_24 JVM on Windows XP on a 2.33GHz 64-bit Xeon processor with 32GB RAM
- 77 RUGRAT-generated Java programs
- 33 open-source projects from SourceForge
- 4 RAT-tools: FindBugs, PMD, JLint and Randoop

RQ1: Similarity in Subject Applications

- Calculated 78 different software metrics for each application (both generated and downloaded)
- Used ANOVA to determine if significant difference occurs w.r.t metrics

Answer: Statistically impossible to tell whether an app is generated or written by humans

Few Examples of Metrics

- We used Eclipse plug-in, Metrics 1.3.6 to calculate different software metrics. E.g.,
  - NSM – Number of Static Methods
  - TLOC – Total Lines of Code
  - NOC – Number of Classes
  - NOF – Number of Attributes
  - DIT – Depth of Inheritance Tree
  - NOM – Number of Methods
  - VG – McCabe Cyclomatic Complexity and more

RQ2: Comparing RAT Tools

- Used 4 RAT tools: 3 static, 1 dynamic
  - FindBugs, PMD, JLint and Randoop
- 2 configurations for each tool: min. and max.
  - Min-config: each tool’s default features/bug patterns are enabled
  - Max-config: each tool’s all the features are enabled
- (**77** generated app) * (**4** RAT tools) * (**2** config/tool) = **616** exprs.

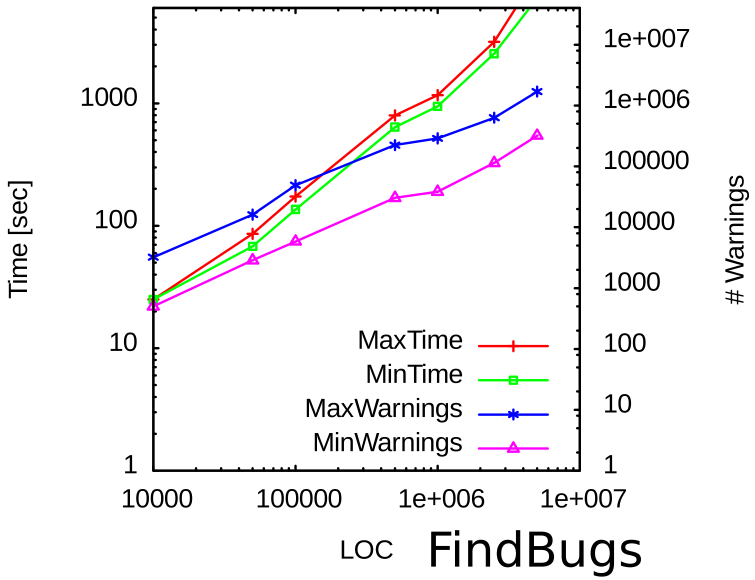
Min. and Max. Configuration of RAT-tools

- FindBugs
  - Min-config: ‘effort’ = minimum; ‘reportLevel’ = high
  - Max-config: ‘effort’ = maximum; ‘reportLevel’ = low
- PMD
  - Min-config: only enabled ruleset is: basic
  - Max-config: enabled rulesets are: braces, clone, codesize, controversial, coupling, design, imports, naming, strictexception, strings, typersolution and unusedcode

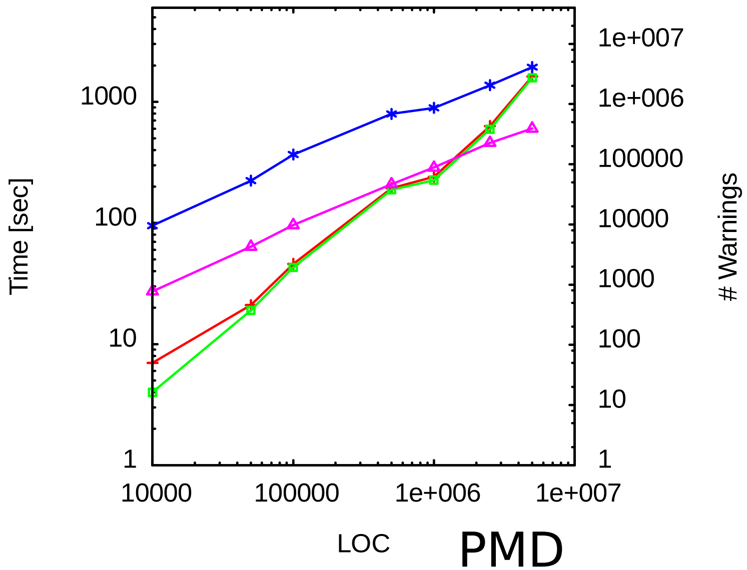
Min. and Max. Configuration of RAT-tools

- JLint
  - Min-config: disabled thread synchronization bug pattern
  - Max-config: enabled all bug patterns
- Randoop (no flags or config. options available)
  - Min-config: time limit = 100 second (default)
  - Max-config: time limit = 2400 second (= 40 min.)

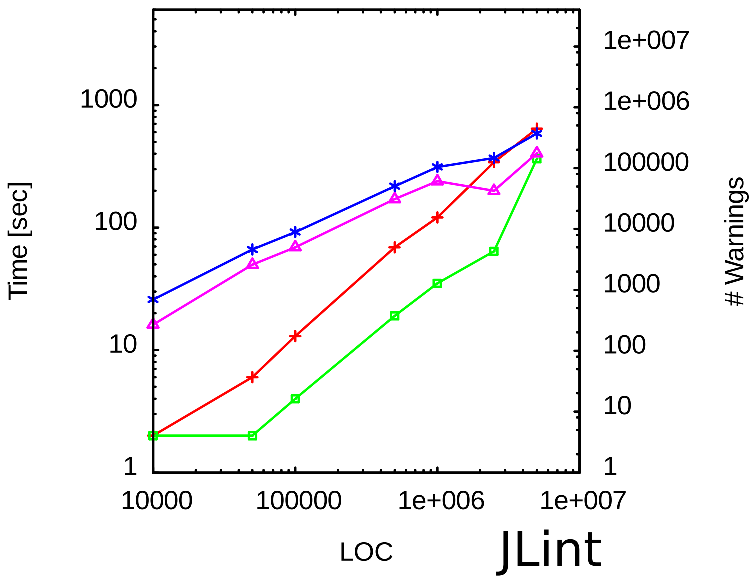
RQ2: Comparing RAT Tools



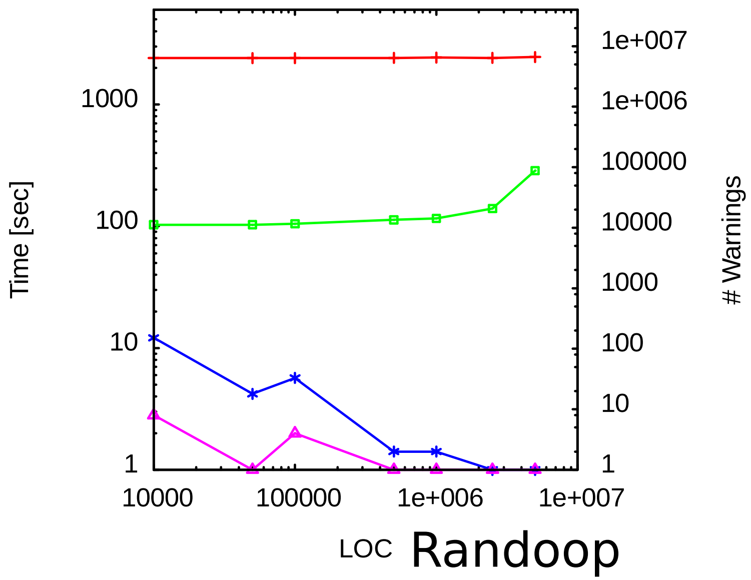
FindBugs



PMD



JLint



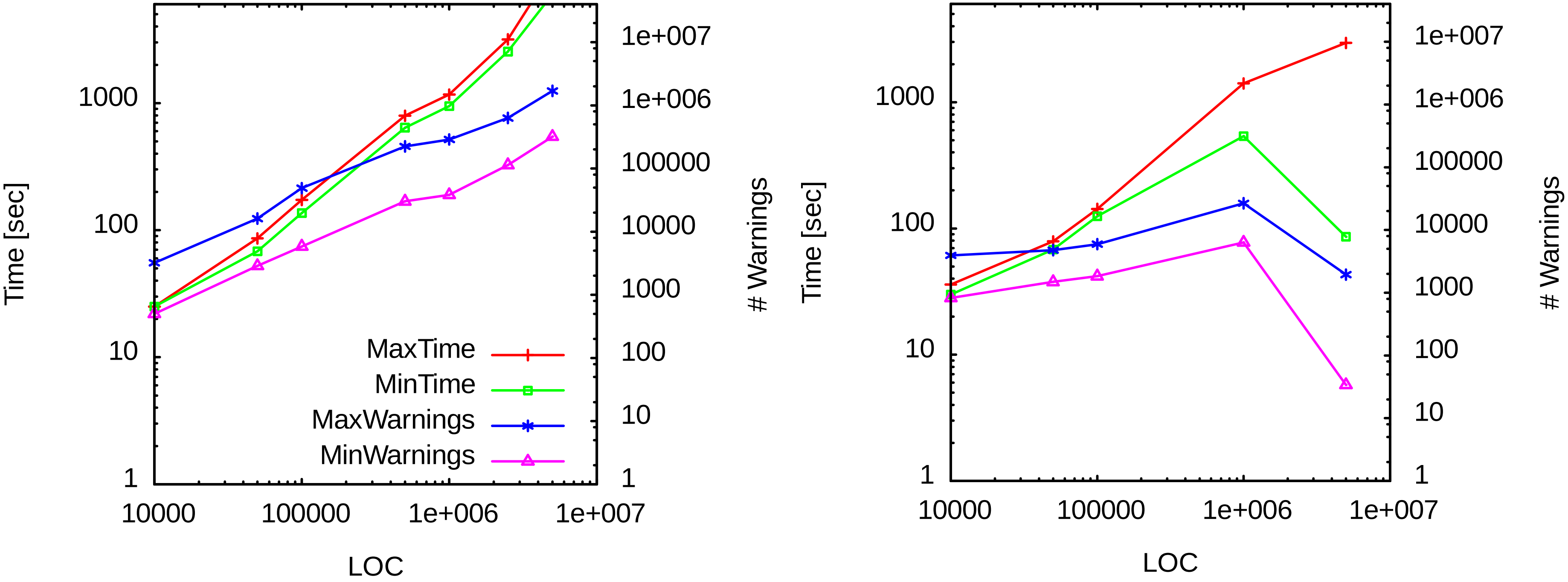
Randoop

RQ2: Comparing RAT Tools

- Overall Results
  - Exec time: JLint < PMD < FindBugs
  - # Warnings: JLint < FindBugs < PMD
- Observations
  - PMD: exec. time approx. equal in both max. and min. config
  - Static tools: (#warnings or Exec. time) α (LOC)
  - Randoop: Exec. time α 1/(LOC)
  - Randoop does not terminate after 100 sec. for larger programs

α ≈
proportional

RQ3: RUGRAT Found RAT Bugs/Issues



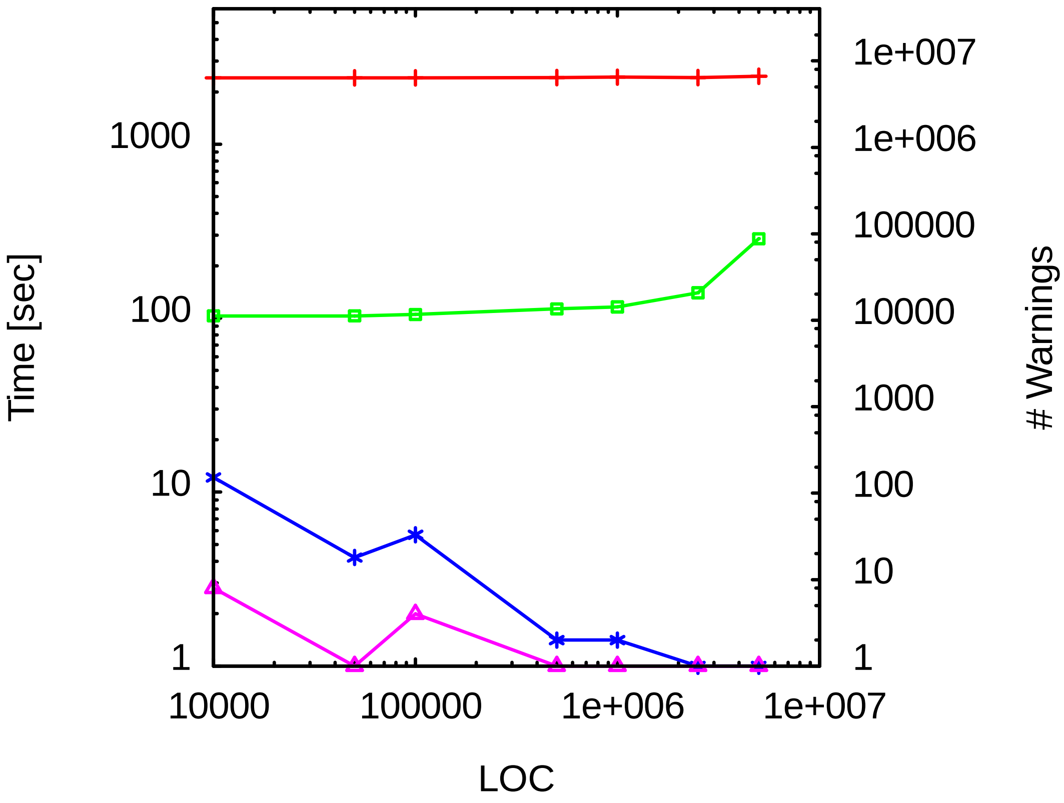
(a) FindBugs (b) Find Bugs skipping some classes

(a) FindBugs’ result of generated programs with default values for the parameters. (b) FindBugs’ result with wider range for the parameter values

RQ3: RUGRAT Found RAT Bugs/Issues

- FindBugs may skip classes and miss bugs
  - if #methods in a class > 1000
  - if size of classfile > 1MB
  - FindBugs author, Bill Pugh confirmed that no configurable options in FindBugs to prevent this
  - He also recommended source code modification as a fix
- Not only generated programs, real programs (manually written/generated then manually modified) may suffer, too
  - Apache Derby, DoctorJ, Drools, and OpenJDK have more than 1000 methods in any class
  - Split classes with less methods makes FindBugs report warnings

RQ3: RUGRAT Found RAT Bugs/Issues



Randoop’s result on generated programs with default value range used for the parameters.

RQ3: RUGRAT Found RAT Bugs/Issues

- We independently discovered a reported Randoop bug known as Issue 14:
  - Randoop terminates without creating any test cases if no test is generated after 10 seconds of the last generated one
- Randoop does not terminate after 100 sec. for larger programs
- Exec. time $\alpha_{\text{proportional}}$ $\overset{\alpha\ \approx}{1/(\text{LOC})}$

Related Work

- Grammar based test input generator: pioneered by Hanford and Purdom in ‘70s
- Slutz [VLDB’98] used random SQL stmt. generator
- Yang et. al. [PLDI’11] in their *Csmith* creates random C programs (no OOP) to test compilers
- Yoshikawa et al. [QSIC’03] used Java random program generator to create small programs (≤ 10 classes) to test JIT compiler
- *ASTGen* by Daniel et. al. [FSE’07]

Things Available at the Tool Website: www.rugrat.ws

- Current prototype tool that works for Java
- Tool source code and executable jar file
- Sample RUGRAT-generated benchmark programs used in the experiments (since full size > 90GB)
- All configuration files with different parameter values that were used in the experiments
- Ant scripts to generate and run the experiments
- List of 33 programs from the SourceForge repository that were downloaded for the experiments (RQ1)
- Links to the RAT tools and supporting libraries used in the experiments

Questions?

The collaborators:

UNIVERSITY OF TEXAS ARLINGTON

accenture

UIC UNIVERSITY OF ILLINOIS AT CHICAGO

Georgia Tech

NORTH CAROLINA STATE UNIVERSITY NC STATE UNIVERSITY

Thank you.

References

- [APSEC’07]: H. Zhang and H. B. K. Tan. An empirical study of class sizes for large Java systems. In *Proc. 14th Asia-Pacific Software Engineering Conference (APSEC), pages* 230–237. IEEE, Dec. 2007
- [SP&E’07]: C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software—Practice & Experience,* 37(6):581–641, May 2007
- [ESEM’10]: M. Grechanik et al. An empirical investigation into a large-scale Java open source code repository. In *Proc. 4th International Symposium on Empirical Software Engineering and Measurement (ESEM).* ACM, Sept. 2010
- [VLDB’98]: D. R. Slutz. Massive stochastic testing of SQL. In *Proc. 24rd International Conference on Very Large Data Bases (VLDB), pages* 618–622. Morgan Kaufmann, Aug. 1998
- [PLDI’11]: X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages* 283–294. ACM, June 2011
- [QSIC’03]: T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proc. 3rd International Conference on Quality Software (QSIC), pages 20–24.* IEEE, Nov. 2003
- [FSE’07]: B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pages* 185–194. ACM, Sept

RQ1: List of Apps Downloaded From SourceForge

#	Name	Total LOC
1	xom	23,170
2	z390	60,945
3	qamanager	4,661
4	openaccess	276,374
5	wsmo4j	67,588
6	yacy	84,080
7	mpire	4,289
8	xkms	9,277
9	flexstor	243,132
10	legalfinger	10,756
11	openjava	63,325
12	lejos	23,479

#	Name	Total LOC
13	equip2	23,866
14	fmj	121,108
15	laser	16,952
16	lockss	67,538
17	mobile	8,631
18	nessconnect	25,023
19	neuroscholar	243,254
20	opentaps	404,887
21	openuat	75,630
2	gigdatamini	

#	Name	Total LOC
25	teamelements	74,673
26	varial	45,982
27	vc2	1,077
28	vmri	13,409
29	webwordcount	42,020
30	xbnjava	19,664
31	xbus	23,507
32	xui	58,360
33	zk1	92,474