Search-Based Software Testing & Test Data Generation for a **Dynamic** Programming Language

Stefan Mairhofer, Robert Feldt & Richard Torkar

14th of July 2011, GECCO, Dublin

SBST for Complex Test Data & DynLang

SBST for Complex Test Data & DynLang

- SB SW Testing for (Code) Coverage:

SBST for Complex Test Data & DynLang

- SB SW Testing for (Code) Coverage:
  - Majority of work focus on simple test data (Numbers)

SBST for Complex Test Data & DynLang

- SB SW Testing for (Code) Coverage:
  - Majority of work focus on simple test data (Numbers)
  - Statically typed languages

SBST for Complex Test Data & DynLang

- SB SW Testing for (Code) Coverage:
  - Majority of work focus on simple test data (Numbers)
  - Statically typed languages
- This study:

SBST for Complex Test Data & DynLang

- SB SW Testing for (Code) Coverage:
  - Majority of work focus on simple test data (Numbers)
  - Statically typed languages
- This study:
  - Complex data types

SBST for Complex Test Data & DynLang

- SB SW Testing for (Code) Coverage:
  - Majority of work focus on simple test data (Numbers)
  - Statically typed languages
- This study:
  - Complex data types
  - Dynamic programming language (Ruby)

How is this different?

How is this different?

- Dynamic languages typically:

How is this different?

- Dynamic languages typically:
  - **Interpreted** rather than compiled

How is this different?

- Dynamic languages typically:
  - **Interpreted** rather than compiled
  - Allow **runtime modification**

How is this different?

- Dynamic languages typically:
  - **Interpreted** rather than compiled
  - Allow **runtime modification**
  - **Dynamically typed**, i.e. less/no type constraints in code

How is this different?

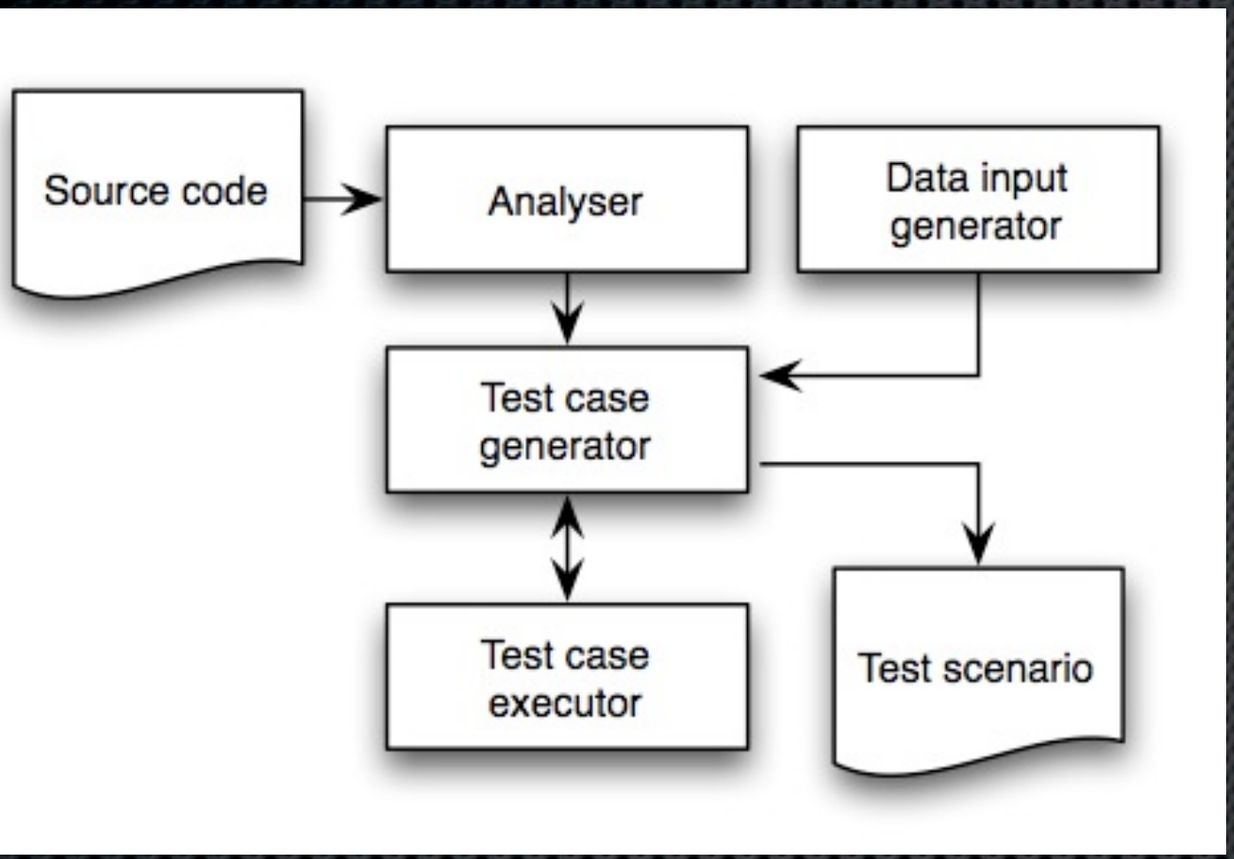
- Dynamic languages typically:
  - **Interpreted** rather than compiled
  - Allow **runtime modification**
  - **Dynamically typed**, i.e. less/no type constraints in code
- Partly because of dynamisms + Object-oriented:

How is this different?

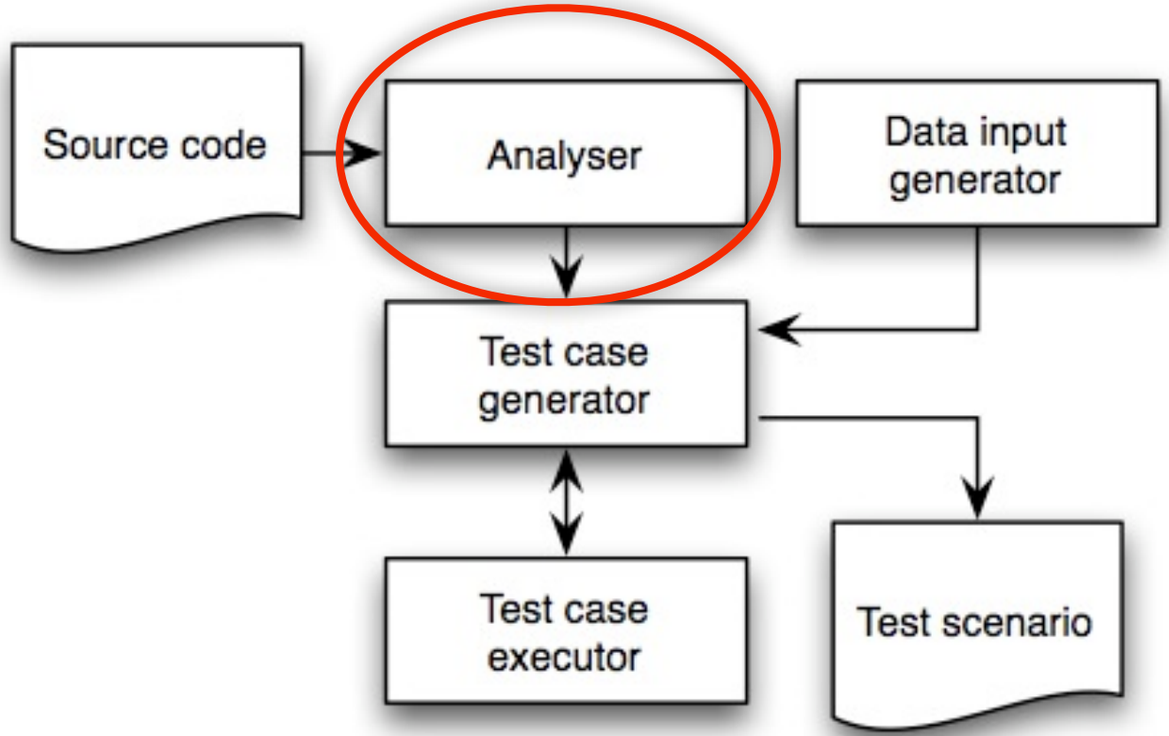
- Dynamic languages typically:
  - **Interpreted** rather than compiled
  - Allow **runtime modification**
  - **Dynamically typed**, i.e. less/no type constraints in code
- Partly because of dynamisms + Object-oriented:
  - **More complex**/own data types

```
class TestClass

  def method1(a,b)
    sum = a + b
    puts sum
  end

  def method2(*c)
    l = c.length
    puts l
  end

end
```

RUTEG = RUby Test Case Generator

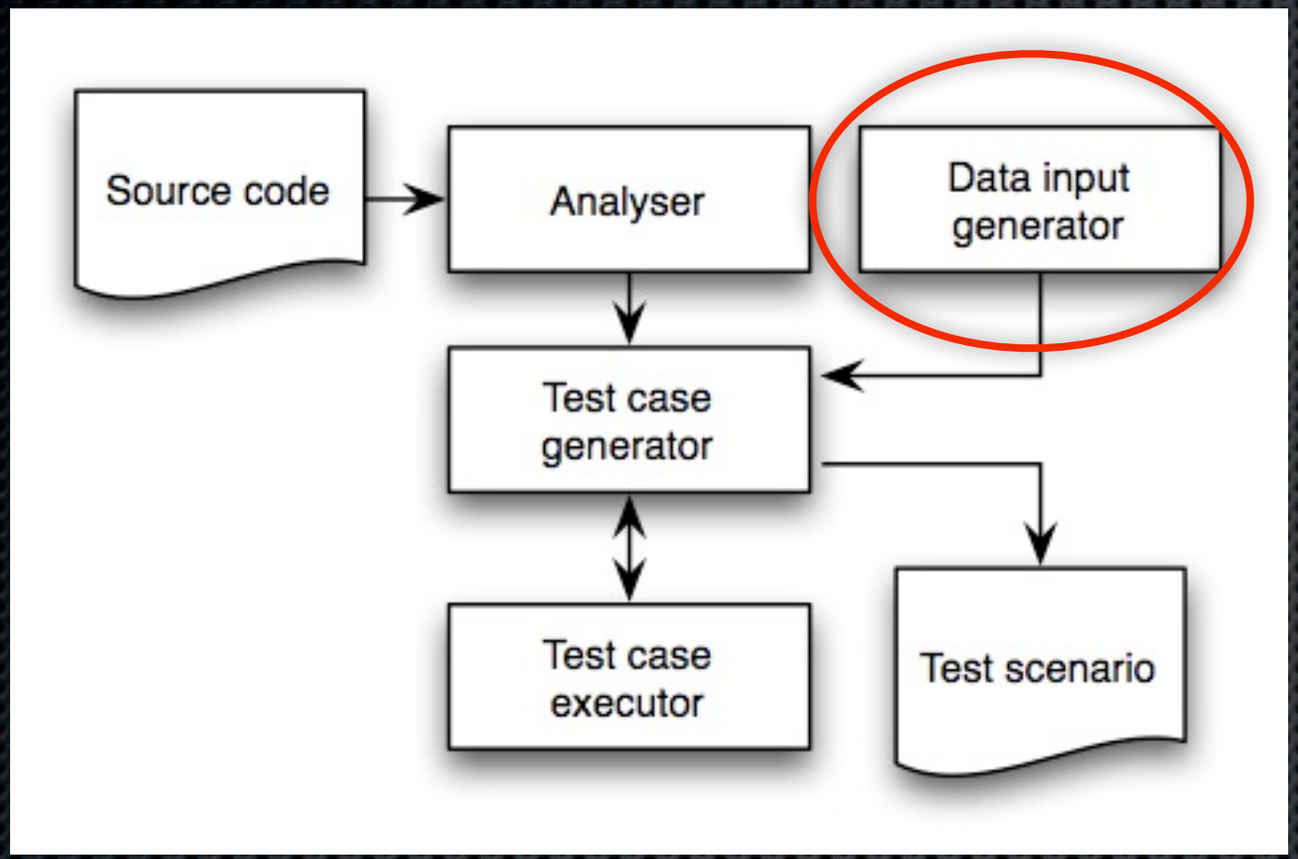


RUTEG = RUby Test Case Generator



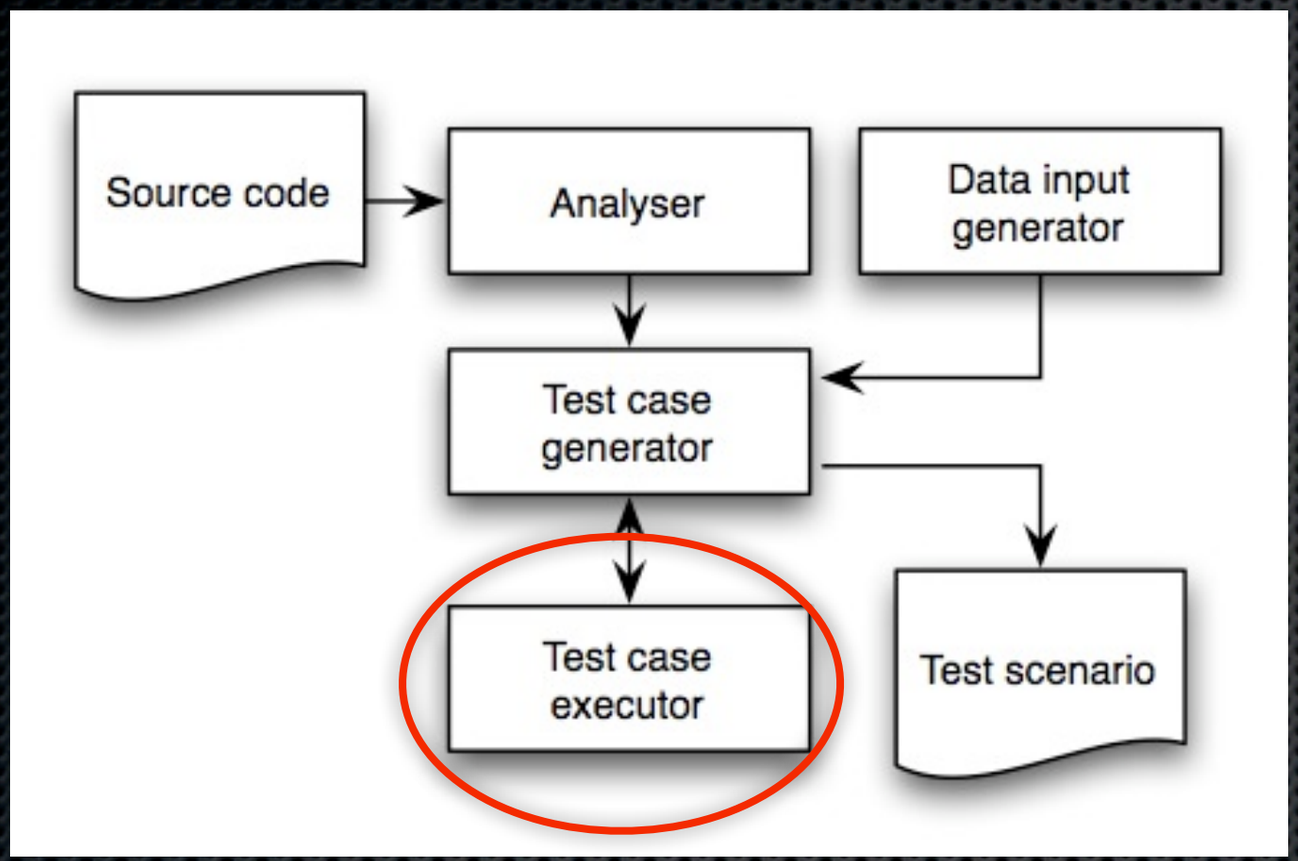
- Static code analysis
- Goal: Reduce search space
- Returns info on:
  - Method names
  - Argument lists
  - Mandatory/Optional args
  - Default values
  - Methods called on all args

RUTEG = RUby Test Case Generator



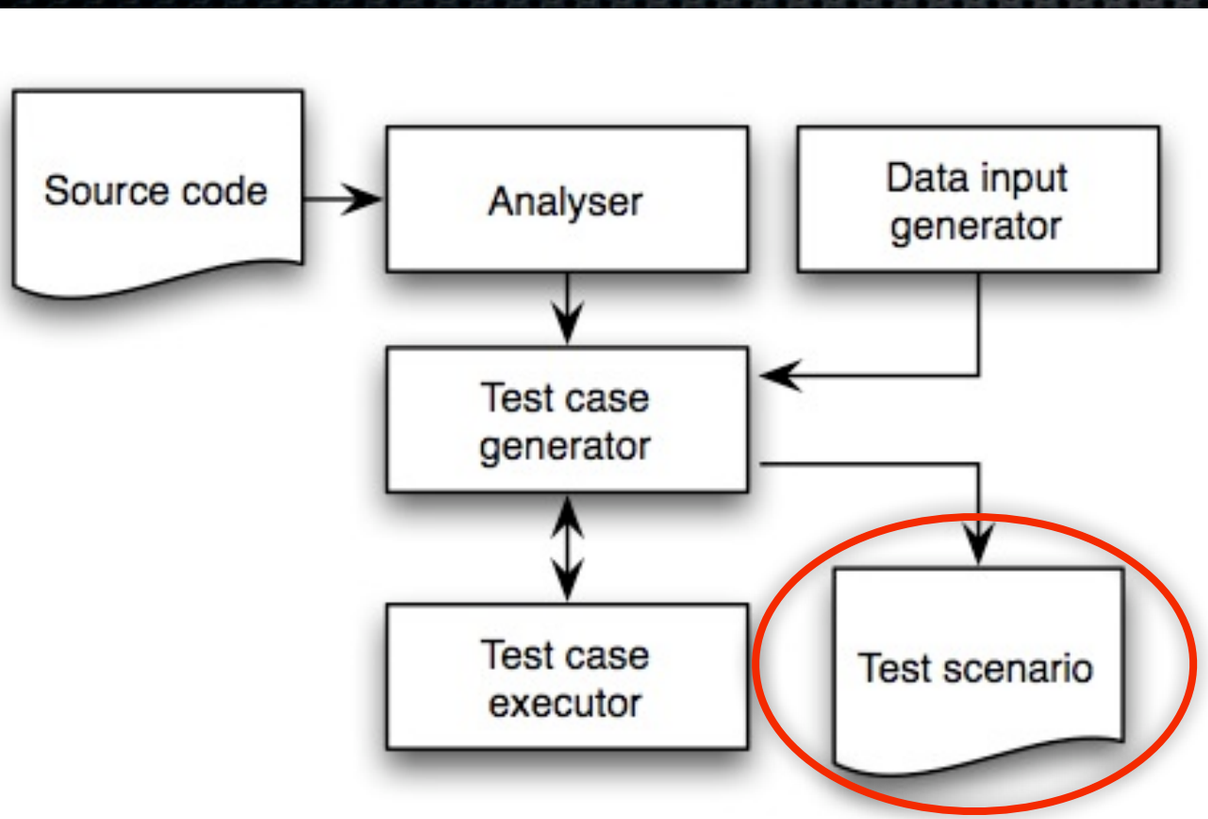
- Problem-specific generators
- Simple OO design
- Basic types supported:
  - Fixnum, Float, String
  - Nil, Object, ArrayOf

RUTEG = RUby Test Case Generator



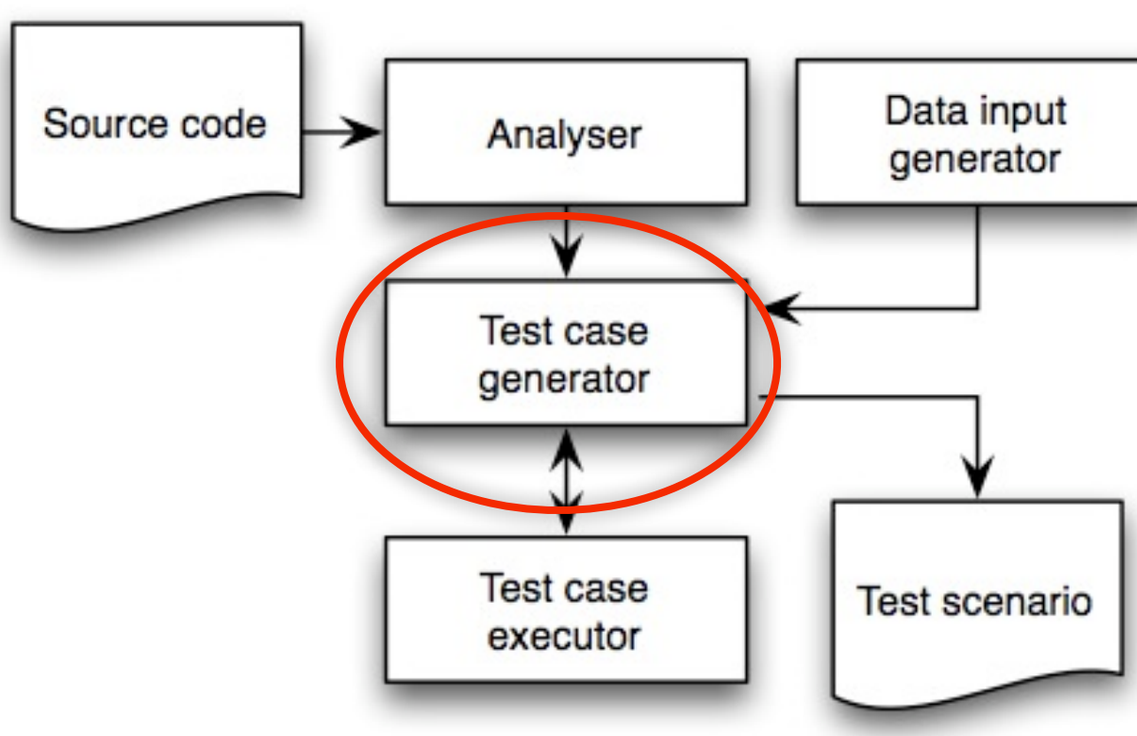
Runs test case and collects coverage info

RUTEG = RUby Test Case Generator

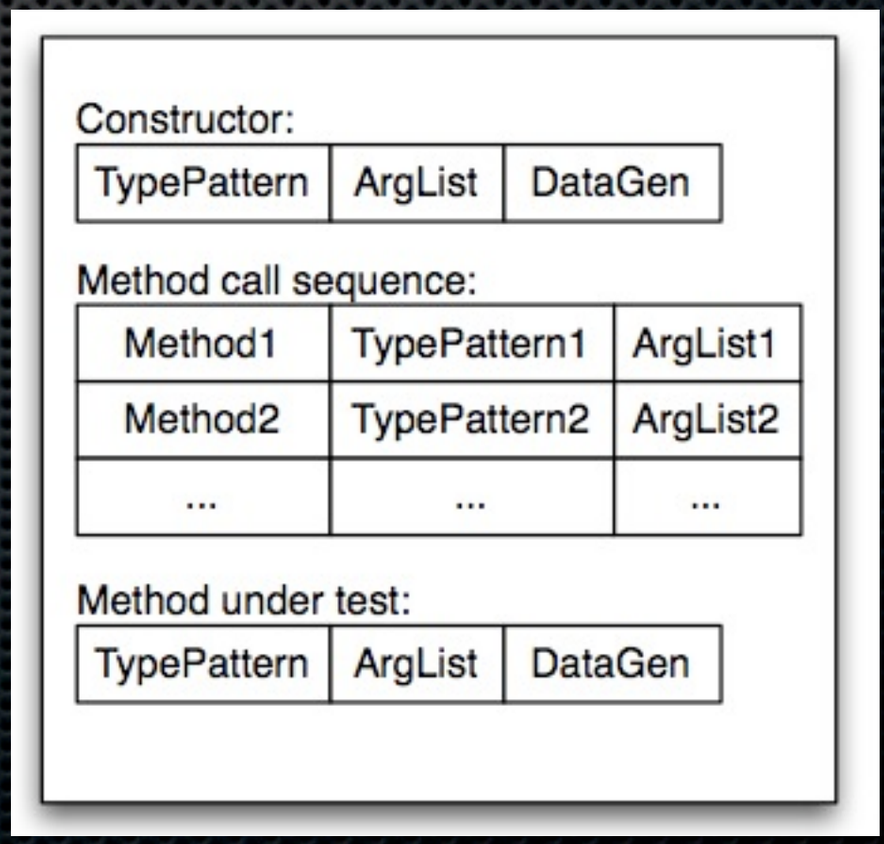


Individuals can be dumped as Ruby test/unit tests

RUTEG = RUby Test Case Generator



GA with individuals:



Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

(Fixnum, Fixnum) or (String, String) ok!

Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

(Fixnum, Fixnum) or (String, String) ok!

(String, Fixnum) or (Fixnum, String) not!

Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

(Fixnum, Fixnum) or (String, String) ok!

(String, Fixnum) or (Fixnum, String) not!

For each method application maintain sets of type patterns:

Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

(Fixnum, Fixnum) or (String, String) ok!

(String, Fixnum) or (Fixnum, String) not!

For each method application maintain sets of type patterns:

Applicable

Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

(Fixnum, Fixnum) or (String, String) ok!

(String, Fixnum) or (Fixnum, String) not!

For each method application maintain sets of type patterns:

Applicable

Suspicious

Argument Type Selection

- Fitness of *type* for *arg*:
  - For fitness-proportionate type selection
  - Ratio of existing methods to invoked methods (for each arg.)
- Not enough since not independent between arguments:

```
def add(a, b)
  a+b
end
```

(Fixnum, Fixnum) or (String, String) ok!

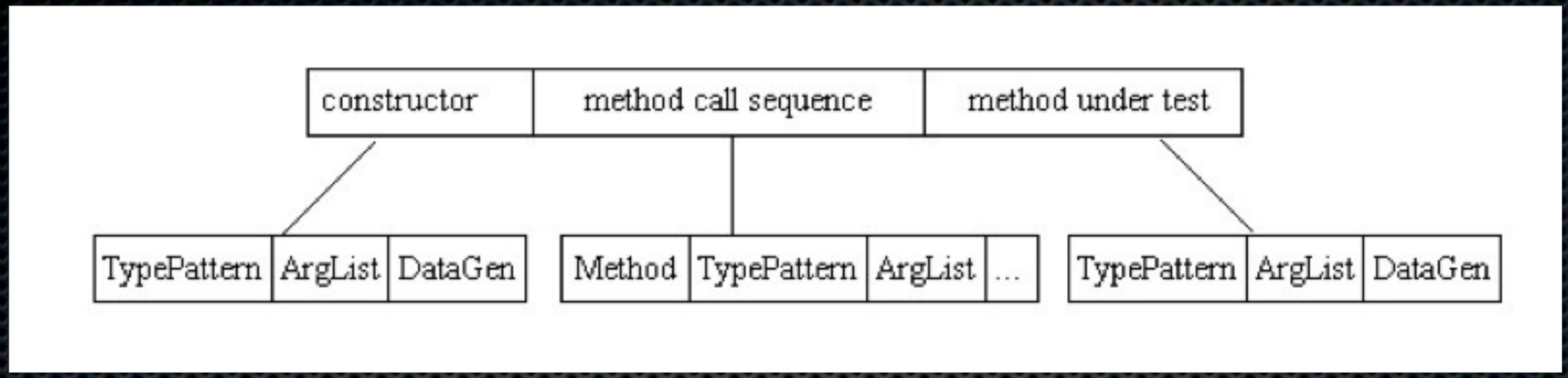
(String, Fixnum) or (Fixnum, String) not!

For each method application maintain sets of type patterns:

Applicable

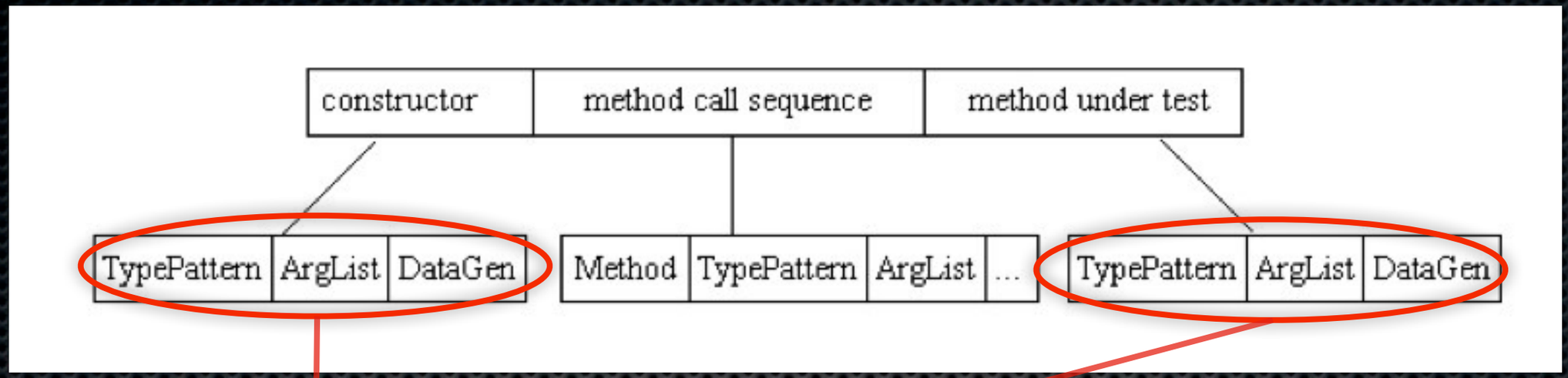
Suspicious

Discarded



XOver:

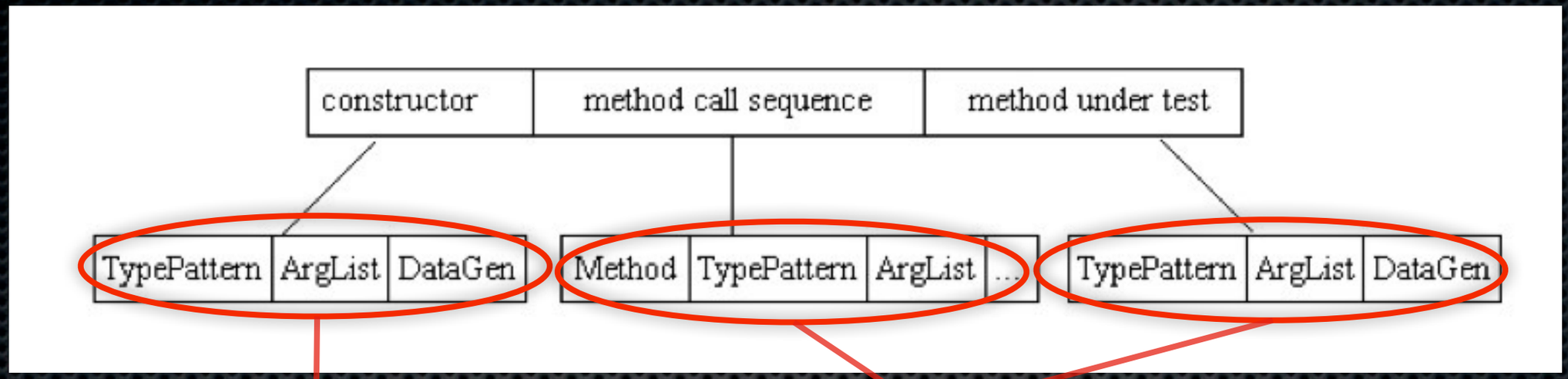
Mutation:



XOver:

One-point xover that handles default & var.length

Mutation:



XOver:

One-point xover that handles default & var.length

Cut & Splice xover

Mutation:

- New type patterns
- New data generators for given type pattern
- New methods in call sequence (randomly from CUT)

$$f_{fitness} = (cov \cdot p) + \left(\frac{executed\_cs}{total\_cs} \cdot (1 - p) \right)$$

$$f_{fitness} = (cov \cdot p) + \left(\frac{executed\_cs}{total\_cs} \cdot (1 - p) \right)$$

Code coverage

Calc by rcov

$$f_{fitness} = (cov \cdot p) + \left(\frac{executed\_cs}{total\_cs} \cdot (1 - p)\right)$$

Code coverage

Calc by rcov

Num. executed/existing control structure lines

From static analysis + rcov

$$f_{fitness} = (cov \cdot p) + \left(\frac{executed\_cs}{total\_cs} \cdot (1 - p)\right)$$

Code coverage

Calc by rcov

0.5

Num. executed/existing control structure lines

From static analysis + rcov

Experiment

- RUTEG vs Random Testing on 14 methods under test:

Project	Method	SLOC	CC
Triangle	triangle_type	26	8
ISBN Checker	valid_isbn10?	18	7
	valid_isbn13?	13	6
AddressBook	add_address	10	3
RBTree	rb_insert	49	7
Bootstrap	bootstrapping	38	9
RubyStat	gamma	116	6
RubyGraph	bfs	39	12
	dfs	34	10
	warshall_floyd_shortest_paths	26	11
Ruby 1.8	rank	56	13
	** (power!)	59	16
RubyChess	canBlockACheck	23	10
	move	111	26
TOTAL (Average):		44.1	10.3

Experiment

- RUTEG vs Random Testing on 14 methods under test:

Project	Method	SLOC	CC
Triangle	triangle_type	26	8
ISBN Checker	valid_isbn10?	18	7
	valid_isbn13?	13	6
AddressBook	add_address	10	3
RBTree	rb_insert	49	7
Bootstrap	bootstrapping	38	9
RubyStat	gamma	116	6
RubyGraph	bfs	39	12
	dfs	34	10
	warshall_floyd_shortest_paths	26	11
Ruby 1.8	rank	56	13
	** (power!)	59	16
RubyChess	canBlockACheck	23	10
	move	111	26
TOTAL (Average):		44.1	10.3

- 30 runs for each method

Experiment

- RUTEG vs Random Testing on 14 methods under test:

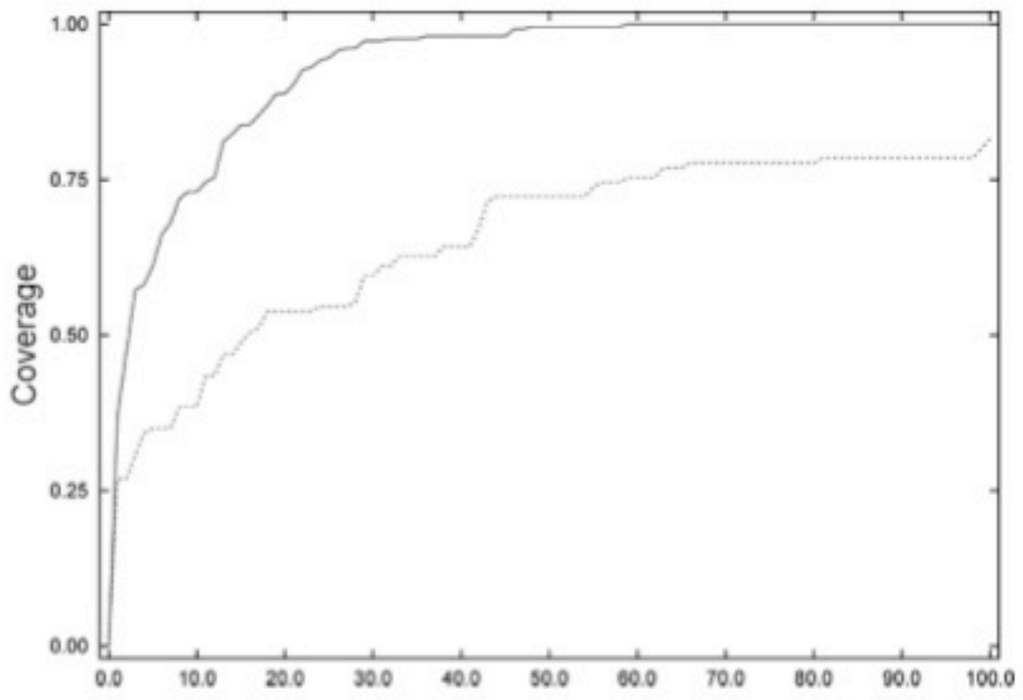
Project	Method	SLOC	CC
Triangle	triangle_type	26	8
ISBN Checker	valid_isbn10?	18	7
	valid_isbn13?	13	6
AddressBook	add_address	10	3
RBTree	rb_insert	49	7
Bootstrap	bootstrapping	38	9
RubyStat	gamma	116	6
RubyGraph	bfs	39	12
	dfs	34	10
	warshall_floyd_shortest_paths	26	11
Ruby 1.8	rank	56	13
	** (power!)	59	16
RubyChess	canBlockACheck	23	10
	move	111	26
TOTAL (Average):		44.1	10.3

- 30 runs for each method
- RUTEG-specific code is 780 LOCs

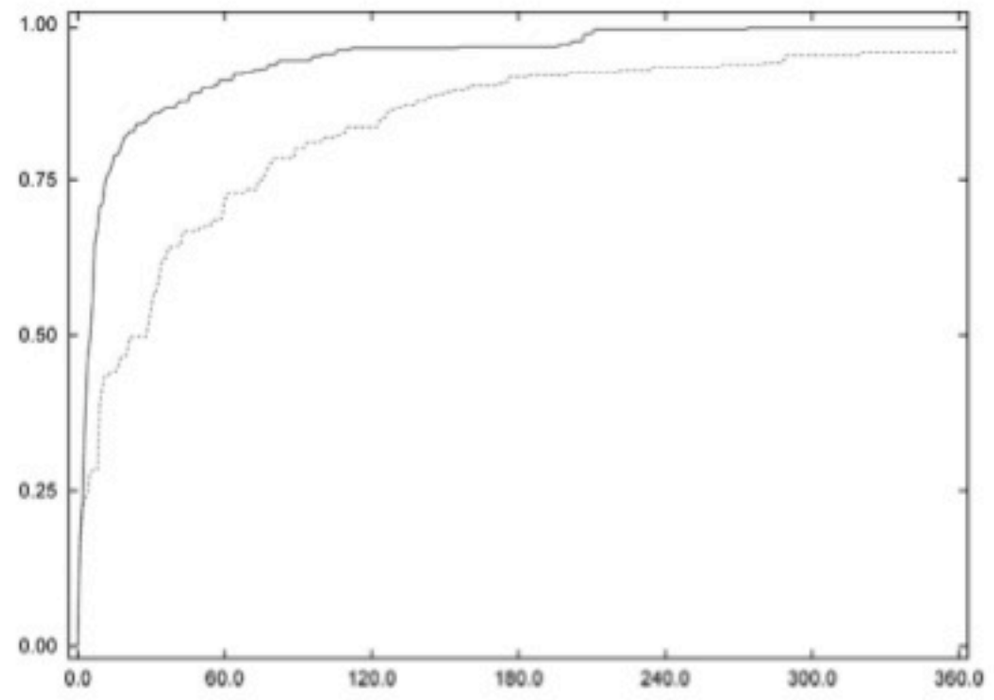
Experiment: Results

Method	Cov. RuTeG	Cov. RT		Time RuTeG	Time RT
triangle_type	100%	81%	**	59	99
valid_isbn10?	100%	100%		29	84
valid_isbn13?	100%	100%		34	80
add_address	100%	100%		56	97
rb_insert	100%	88%	**	68	92
bootstrapping	100%	86%	*	54	88
gamma	98%	92%	**	209	213
bfs	100%	93%	*	79	86
dfs	100%	96%	*	70	72
warshall_floyd_shortest_paths	100%	100%		155	196
rank	100%	92%	*	111	202
** (power!)	100%	96%	**	274	356
canBlockACheck	94%	74%	**	285	333
move	88%	68%	**	356	143
	98.6%	90.4%		131.4	152.9

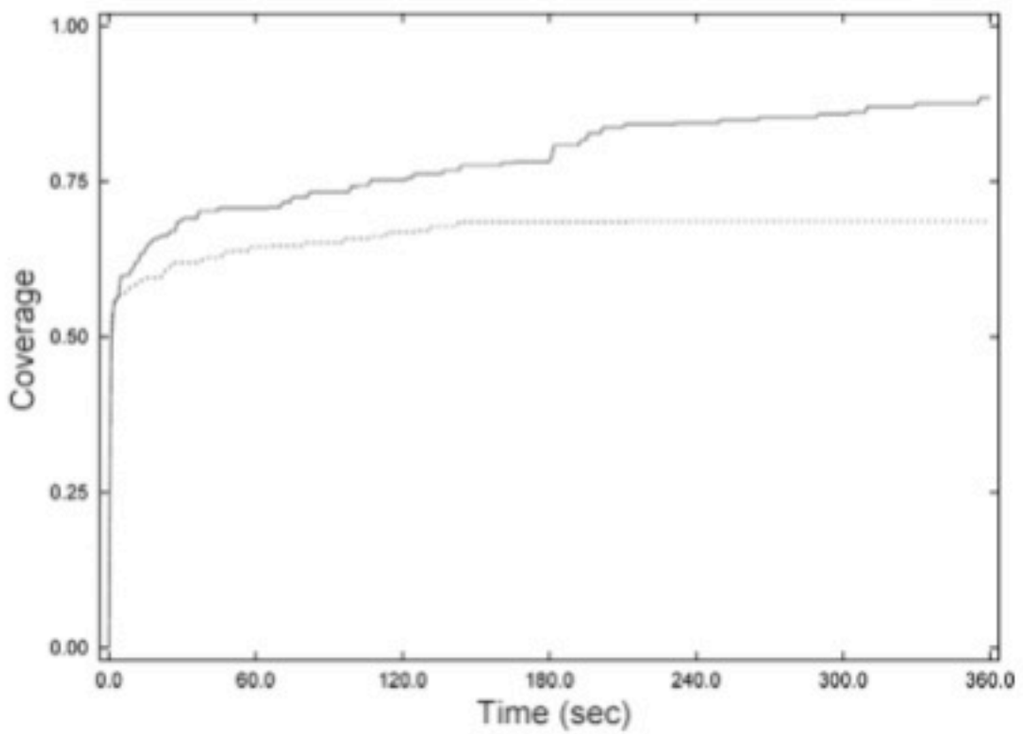
Table 2: Average code coverage achieved by RuTeG and random testing (RT), with t-test where * indicates $p < 0.05$ and ** indicates $p < 0.01$; and the time to maximum coverage expressed in seconds.

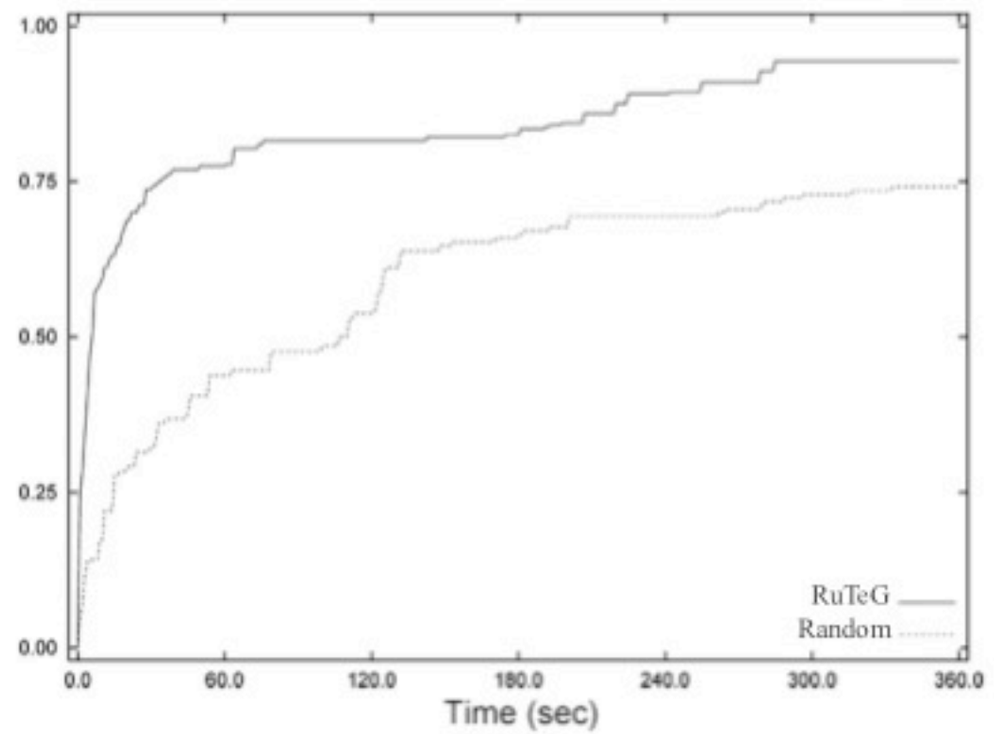


(a) triangle_type



(b) ** (power!)





Discussion

Discussion

- Why not branch coverage?

Discussion

- Why not branch coverage?
  - Simple reason: No branch coverage tools!

Discussion

- Why not branch coverage?
  - Simple reason: No branch coverage tools!
  - To keep things simple! Would need more advanced parser and fitness function

Discussion

- Why not branch coverage?
  - Simple reason: No branch coverage tools!
  - To keep things simple! Would need more advanced parser and fitness function
- Basic (line) coverage enough!

Discussion

- Why not branch coverage?
  - Simple reason: No branch coverage tools!
  - To keep things simple! Would need more advanced parser and fitness function
  - Basic (line) coverage enough!
  - Controversial: SBST research often overfits to eval examples. Why use general search if adaptation needed?

Discussion

- Why not branch coverage?
  - Simple reason: No branch coverage tools!
  - To keep things simple! Would need more advanced parser and fitness function
  - Basic (line) coverage enough!
  - Controversial: SBST research often overfits to eval examples. Why use general search if adaptation needed?
- What is the “right” random generator to compare to?

Discussion

- Why not branch coverage?
  - Simple reason: No branch coverage tools!
  - To keep things simple! Would need more advanced parser and fitness function
  - Basic (line) coverage enough!
  - Controversial: SBST research often overfits to eval examples. Why use general search if adaptation needed?
- What is the “right” random generator to compare to?
  - Hard question for much of SBST, there is often a continuum

Discussion

Discussion

- Limitations?

Discussion

- Limitations?
  - Took non-linearly longer when dependencies in call sequence

Discussion

- Limitations?
  - Took non-linearly longer when dependencies in call sequence
- Tester need to add problem-specific Data Generators

Discussion

- Limitations?
  - Took non-linearly longer when dependencies in call sequence
  - Tester need to add problem-specific Data Generators
  - Cannot generate code blocks

Table 1: Parameter settings for the experiment.

Param.	Setting	Comment
Population size	50	
Selection method	Tournament	Tournament size = 4
Mutation rate	0.2	Mutate the input pattern or mutate an input value
Cross-over rate	1.0	One-point xover for argument lists, cut & splice for method call lists
Initial sequence length	Random	Random with a max length of two times the number of methods in the class under test, cut & splice cross-over can change length dynamically
Weight param. α	0.5	Half fitness on coverage, half on covering control structures