Search-based Software Testing and Test Data Generation for a Dynamic Programming Language

Stefan Mairhofer, Robert Feldt, Richard Torkar
GECCO, 2011, Dublin, Ireland.

Simon Bihel `simon.bihel@ens-rennes.fr`

Friday 28th July, 2017

COINSE Lab, KAIST, South-Korea

Table of contents

Background

Characteristics of dynamic languages

Typically:

- interpreted rather than compiled,
- allow runtime modification,
- dynamically-typed,
- complex data structures with object-oriented.

Characteristics of dynamic languages

Typically:

- interpreted rather than compiled,
- allow runtime modification,
- dynamically-typed,
- complex data structures with object-oriented.

Related Works

Focused on statically-typed languages.

Complex data generation, such as strings, arrays...

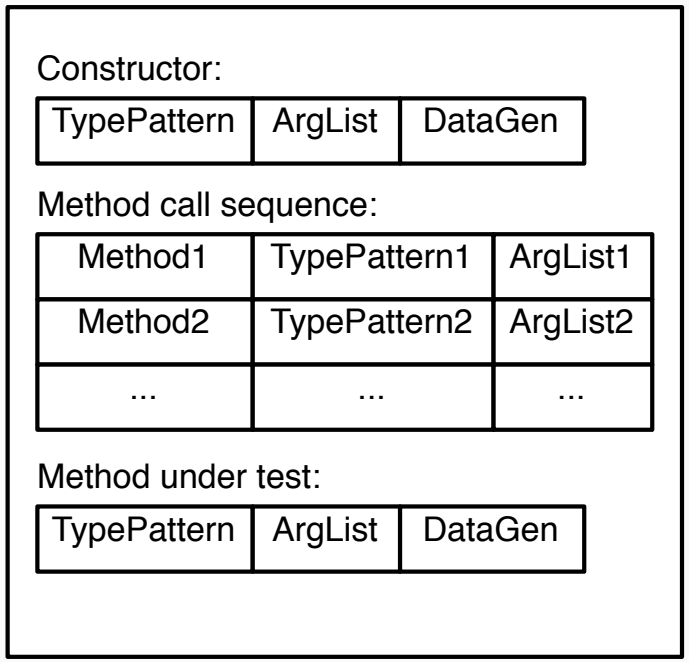
Object-oriented programs testing.

- Methods calls sequence.
- Different fitness functions, like method call distance.

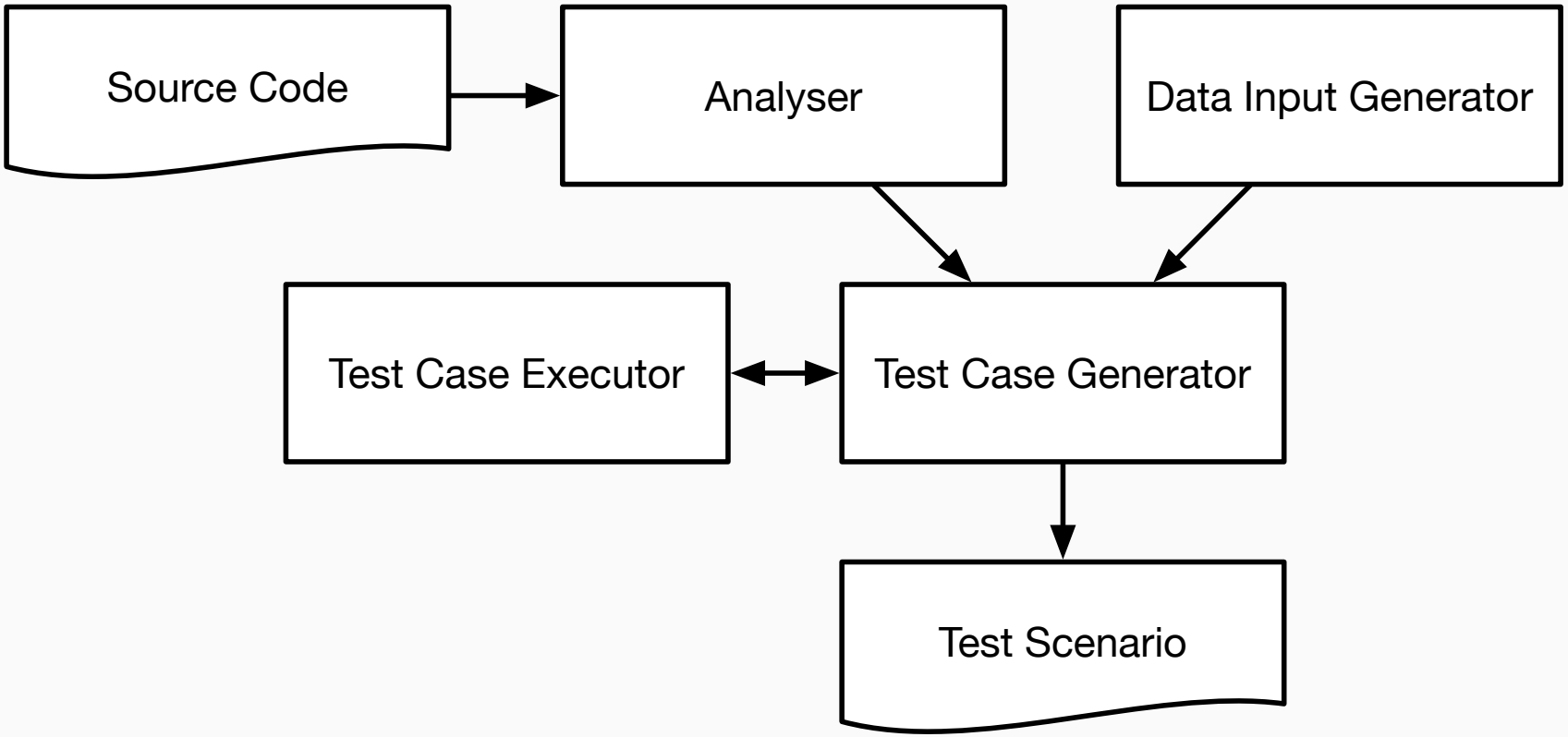
The Ruby Test Case Generator (RuTeG)

Goal

- Building an instance of the Class Under Test
- Set a methods call sequence with arguments
- Set the arguments for the final call to the Method Under Test



Architecture



Static Analysis

Reduce search space.

- Method names and argument lists
- Methods called on arguments

Identify code structure and methods impacting the state of the instance.

Data Input Generator

Basic types (numbers, strings, objects, arrays) have generators natively.

For other types, problem-specific generators have to be **provided by the user**.

Test Case generator

Genetic algorithm for each of the 3 parts:

- constructor;
- method call sequence; and
- method under test.

Fitness depending on code coverage and number of control structure lines executed.

Experiment

Setup

Comparison with a fully random test generator (written by them).

Fixed genetic algorithms parameters. Random initial sequence length. Tournament selection method.

Test methods: classical code snippets as well as more complex methods from the standard library. Each test candidate was tested 30 times.

Results

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.

Project	Method	SLOC	CC	Cov. RuTeG	Cov. RT		Time RuTeG	Time RT
Triangle	triangle_type	26	8	100%	81%	**	59	99
ISBN Checker	valid_isbn10?	18	7	100%	100%		29	84
	valid_isbn13?	13	6	100%	100%		34	80
AddressBook	add_address	10	3	100%	100%		56	97
RBTree	rb_insert	49	7	100%	88%	**	68	92
Bootstrap	bootstrapping	38	9	100%	86%	*	54	88
RubyStat	gamma	116	6	98%	92%	**	209	213
RubyGraph	bfs	39	12	100%	93%	*	79	86
	dfs	34	10	100%	96%	*	70	72
	warshall_floyd_shortest_paths	26	11	100%	100%		155	196
Ruby 1.8	rank	56	13	100%	92%	*	111	202
	** (power!)	59	16	100%	96%	**	274	356
RubyChess	canBlockACheck	23	10	94%	74%	**	285	333
	move	111	26	88%	68%	**	356	143
TOTAL (Average):		44.1	10.3	98.6%	90.4%		131.4	152.9

Results

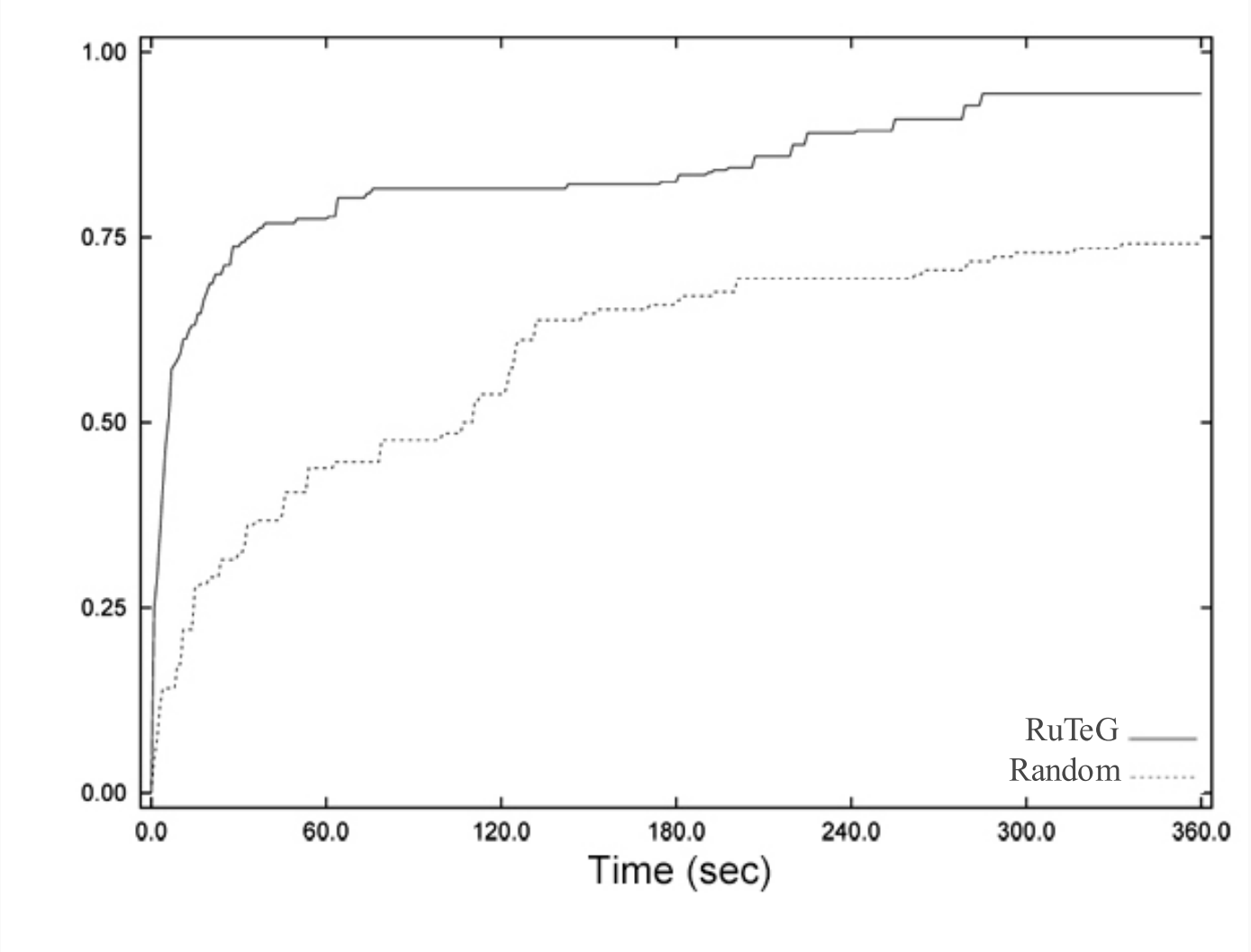


Figure 4: Average code coverage for `canBlockACheck`.

Results

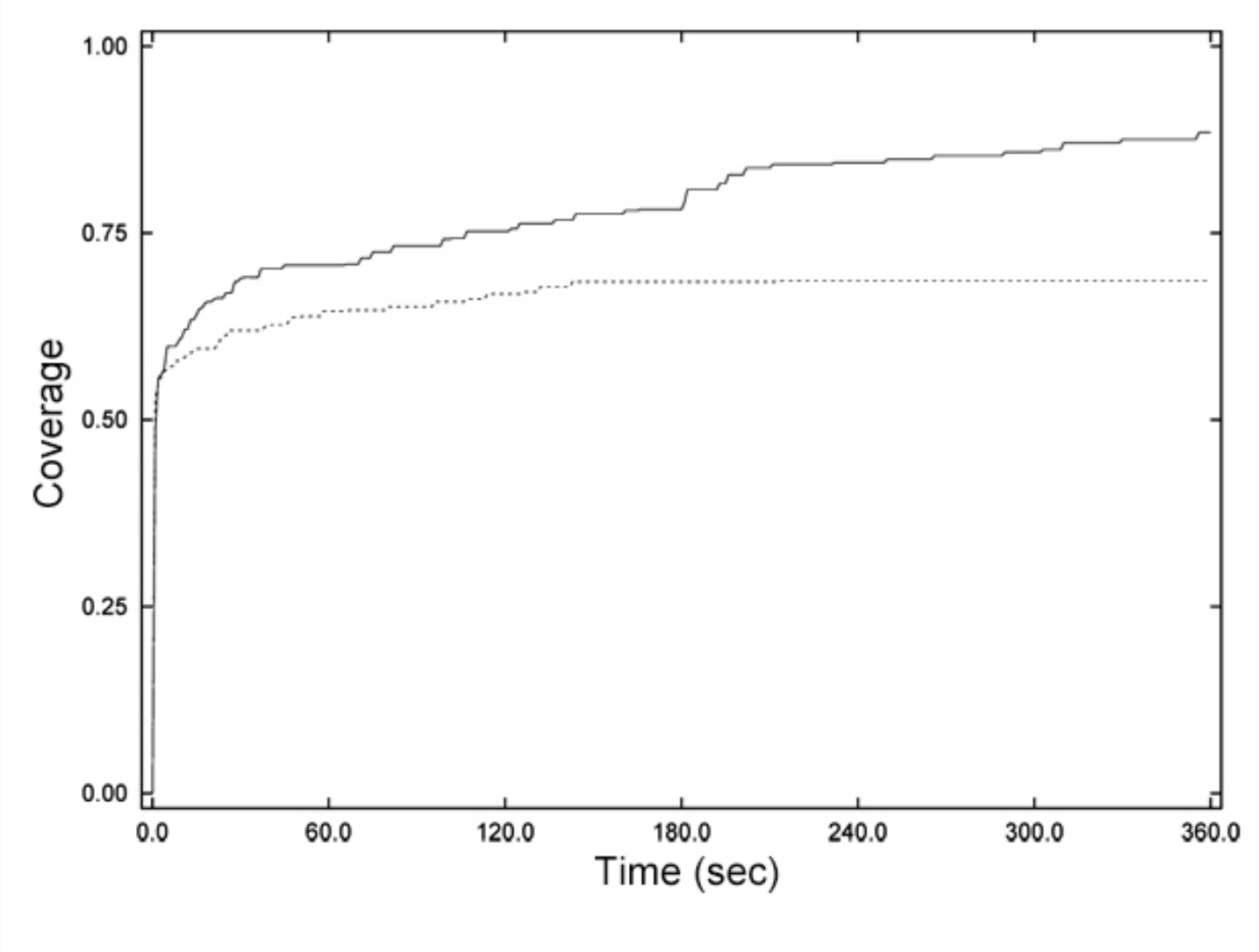


Figure 3: Average code coverage for `move` method.

Discussion

Discussion from this paper

- Difficulty to find appropriate call sequences, especially for real life libraries.
- Limited by data generators.
- Generation of code blocks not handled.
- Small and complex target range values.
- Often times it is the state that will impact which part of the code will be executed.

- Disqualify invalid types (but it is not clear it would work with runtime code addition).
- Minimal guidance on the search because of the simple fitness function.

Pros/Cons

Pros:

- they built something that works on some real life code; and
- extensive work on analysing the validity of the results.

Cons:

- basic experiment observations;
- tackling so much paradigms at once; and
- the paper is not very well written.

My Experience

No effort followed this paper.

I tried to write a Python tool focusing only on the dynamic typing. It was intended for branch coverage of simple functions with primitive types and lists (containing different types and possibly other lists).

What We Need To Do

- Break down problems, focus on paradigms one at a time.
- Need for a fully random tester that eventually works in every situation before trying to have smarter tools.
- Don't underestimate technical problems and aim at generic tools.