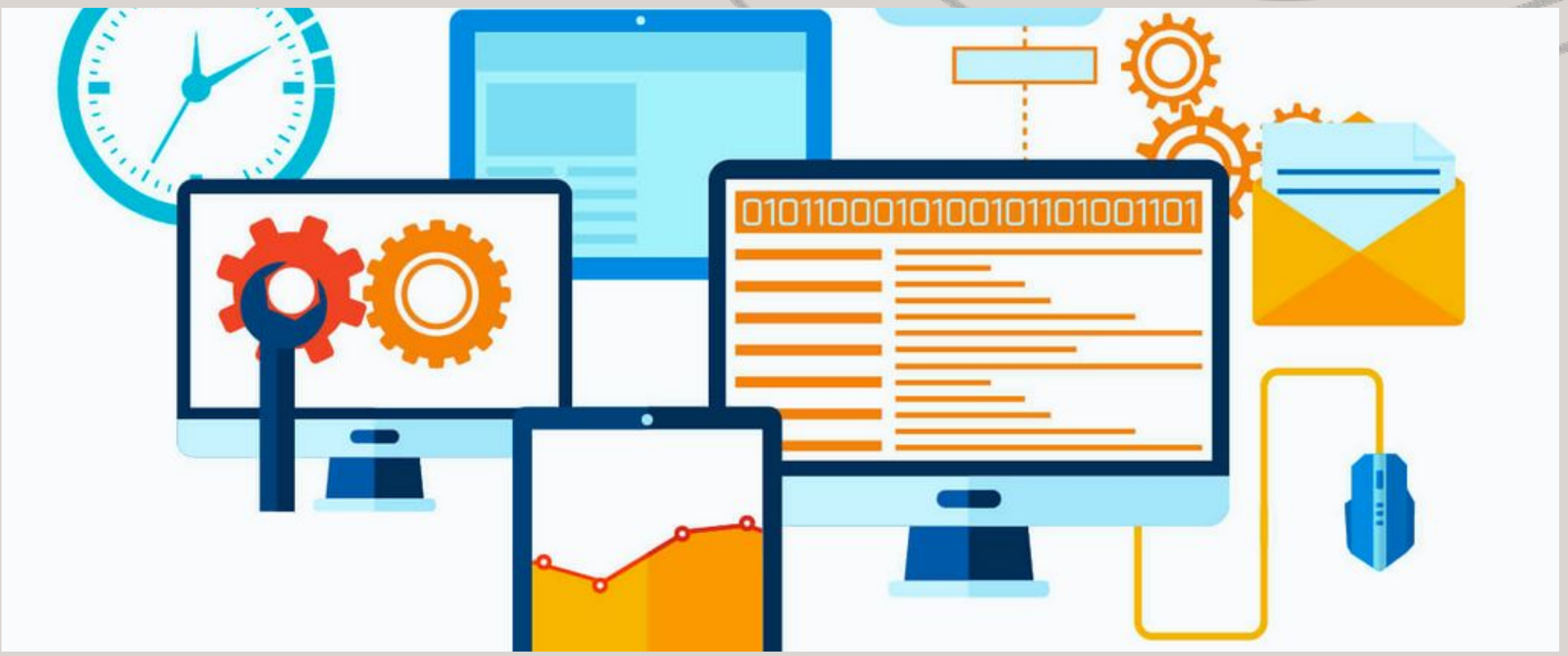
ARTOO: Adaptive Random Testing for Object-Oriented Software

Ana Jin 99176 | Juliana Marcelino 99261 | Madalena Cunha 99267

Software Testing and Validation

Introduction

- Evaluation criteria for testing strategies:
  - effectiveness in finding faults
  - efficiency in speed
- Automated solutions can’t test every scenario in complex programs



Random Testing

Easy to implement

No input selection overhead

Avoids human bias

High applicability

Fast execution.

Adaptive Random Testing (ART)

01 Enhances efficiency while maintaining benefits of random testing

02 Guides testing for non-random selection, improving fault discovery

03 Prioritizes even distribution of test cases for fewer tests needed

Results: Up to 50% reduction in tests needed to reveal first fault

Extension to object-oriented software: ARTOO introduced for adaptive testing with "object distance" calculation.

Object Distance

$$
\begin{aligned}
p \leftrightarrow q = combination\,(&\\
&elementary\_distance\,(p,q),\\
&type\_distance\,(type\,(p),type\,(q)),\\
&field\_distance\,(\{[p.a \leftrightarrow q.a]\\
&\quad |a \in Attributes\,(type\,(p),type\,(q))\}))
\end{aligned}
$$

Elementary Distance: fixed functions for each value type (e.g., numbers, characters)

Type Distance: summing path lengths to closest common ancestor and non-shared features* count

*Non-shared features are features not inherited from a common ancestor

Field Distance: recursively applying distance calculation to matching fields of compared objects

ARTOO

Objective: Select input objects with the highest average distance to those already used as test inputs.

Distance Calculation: Uses the combination function of elementary, type, and field distances.

Algorithm Overview:

- Calculates the sum of distances between each candidate object and all used test objects.
- Selects the candidate with the highest sum of distances (equivalent to the highest average distance since the divisor is constant).

```
used_objects: SET [ANY]
candidate_objects: SET [ANY]
current_best_distance: DOUBLE
current_best_object: ANY
v0, v1: ANY
current_accumulation: DOUBLE
...

current_best_distance := 0.0
foreach v0 in candidate_objects
do
   current_accumulation := 0.0
   foreach v1 in used_objects
   do
      current_accumulation :=
         current_accumulation + distance(v0, v1)
   end
   if (current_accumulation > current_best_distance)
   then
      current_best_distance := current_accumulation
      current_best_object := v0
   end
end
candidate_objects.remove(current_best_object)
used_objects.add(current_best_object)
run_test(current_best_object)
```

Example: r (o1: A; o2: B)

Implementation

AutoTest

- Automatically tests Eiffel software
- Uses contracts (rules) in the code to verify behavior
- Process:
    - Input Generation
    - Execution
    - Monitoring
- Two-Process Model:
    - Master Process
    - Slave Process

AutoTest

Random Input Generation (RAND)

- AutoTest stores test input objects, creating new instances or using existing ones.
- Input generation algorithm:
  - For target object & arguments, call a random constructor of the class.
  - For primitive types, choose values from all possible values or predefined special values.
- Diversification operations performed on objects in the pool for more variety.
- Generating objects by calling constructors & routines ensures class invariant satisfaction.

AutoTest

ARTOO

Integration with AutoTest:

- Implemented as a plugin for input generation.
- Other parts of the testing process remain unchanged.
- Allows for objective performance comparison between strategies.

Key Features:

- Object Creation and Diversification
- Infinite Recursion Handling
- Distance Calculation

Example

```
class BANK_ACCOUNT
create
  make

feature -- Bank account data
  owner: STRING
  balance: INTEGER

feature -- Initialization
  make (s: STRING; init_bal: INTEGER) is
      -- Create a new bank account.
    require
      positive_initial_balance : init_bal >
      owner_not_void: s /= Void
    do
      owner := s
      balance := init_bal
    ensure
      owner_set: owner = s
      balance_set : balance = init_bal
    end

ature -- Operation

withdraw (sum: INTEGER) is ...

deposit (sum: INTEGER) is ...

transfer (other_account: BANK_ACCOUNT; sum: INTI
    ) is
    -- Transfer 'sum' to 'other_account'.
  require
    can_withdraw: sum <= balance
  do
    balance := balance - sum
    other_account.deposit (sum)
  ensure
    balance_decreased: balance < old balance
    sum_deposited_to_other_account : other_account.b
        > old other_account.balance
  end
ivariant
owner_not_void: owner /= Void
positive_balance : balance >= 0
nd
```

Example

ba1: *BANK_ACCOUNT*, *ba1.owner*=**"A"**, *ba1.balance*=675234
ba2: *BANK_ACCOUNT*, *ba2.owner*=**"B"**, *ba2.balance*=10
ba3: *BANK_ACCOUNT*, *ba3.owner*=**"O"**, *ba3.balance*=99
ba4 = *Void*
$i1$: *INTEGER*, $i1 = 100$
$i2$: *INTEGER*, $i2 = 284749$
$i3$: *INTEGER*, $i3 = 0$
$i4$: *INTEGER*, $i4 = -36452$
$i5$: *INTEGER*, $i5 = 1$

$ba3.\ transfer\ (ba1,\ i5)$

ba1: *BANK_ACCOUNT*, *ba1.owner*=**"A"**, *ba1.balance*=675235
ba2: *BANK_ACCOUNT*, *ba2.owner*=**"B"**, *ba2.balance*=10
ba3: *BANK_ACCOUNT*, *ba3.owner*=**"O"**, *ba3.balance*=98
ba4 = *Void*
$i1$: *INTEGER*, $i1 = 100$
$i2$: *INTEGER*, $i2 = 284749$
$i3$: *INTEGER*, $i3 = 0$
$i4$: *INTEGER*, $i4 = -36452$
$i5$: *INTEGER*, $i5 = 1$

$ba1.\ transfer\ (ba4,\ i2)$

$ba2.\ transfer\ (ba2,\ i4)$

Experimental Results

Experimental Setup

Use real-world code for evaluating testing tools

Testing on unchanged classes from EiffelBase library

These classes are publicly available

All faults mentioned are real faults present in this library

Class	Total lines of code	Lines of contract code	#Routines	#Attributes	#Parent classes
ACTION_SEQUENCE	2477	164	156	16	24
ARRAY	1208	98	86	4	11
ARRAYED_LIST	2164	146	39	6	23
BOUNDED_STACK	779	56	62	4	10
FIXED_TREE	1623	82	125	6	4
HASH_TABLE	1791	178	122	13	9
LINKED_LIST	1893	92	106	6	19
STRING	2980	296	171	4	16

Table 1: Properties of the classes under test

Experimental Setup

Testing Methodology & Results Evaluation

01 Test one class at a time using both RAND and ARTOO

02 Average Results out over 5, 10-minute tests of each class using different seeds

03 Results evaluated according to two factors:

- Number of tests to first fault
- Time to first fault

04 Most crucial attribute: Prioritize the fault-detecting ability of a testing strategy

Results

Routine-by-Routine Comparison

Class	Routine	Tests to first fault			Time to first fault (seconds)		
		ARTOO	RAND	$\frac{ARTOO}{RAND}$	ARTOO	RAND	$\frac{ARTOO}{RAND}$
ARRAYED_LIST	append	**432**	**5517**	**0.08**	311	191	1.62
	do_all	**296**	**737**	**0.40**	137	18	7.48
	do_if	**16**	**1258**	**0.01**	**2**	**39**	**0.05**
	fill	**159**	**7130**	**0.02**	**40**	**256**	**0.16**
	for_all	**303**	**517**	**0.59**	138	17	7.93
	is_inserted	**31**	**126**	**0.25**	**3**	**7**	**0.43**
	make	23	3	7.44	2	1	2.80
	make_filled	**13**	**117**	**0.11**	**2**	**4**	**0.50**
	prune_all	**51**	**10798**	**0.00**	**3**	**367**	**0.01**
	put	96	89	1.08	11	4	2.67
	put_left	**146**	**9739**	**0.01**	**32**	**331**	**0.10**
	put_right	**278**	**8222**	**0.03**	**132**	**291**	**0.45**
	resize	**355**	**1143**	**0.31**	320	30	10.40
	there_exists	**307**	**518**	**0.59**	151	17	8.78
	wipe_out	**594**	**3848**	**0.15**	546	123	4.41

Better performance by ARTOO are highlighted in bold

Class	Routine	Tests to first fault			Time to first fault (seconds)		
		ARTOO	RAND	$\frac{ARTOO}{RAND}$	ARTOO	RAND	$\frac{ARTOO}{RAND}$
ACTION_SEQUENCE	arrayed_list_make	**748**	**6800**	**0.11**	564	174	3.24
	call	**109**	**2382**	**0.05**	**10**	**67**	**0.15**
	duplicate	**378**	**410**	**0.92**	196	13	14.46
	for_all	**286**	**623**	**0.46**	64	21	3.00
	is_inserted	115	95	1.21	5	2	2.36
	make_filled	**183**	**449**	**0.41**	49	13	3.65
	put	81	67	1.21	4	4	1.15
	remove_right	**448**	**17892**	**0.03**	**201**	**475**	**0.42**
	resize	**399**	**5351**	**0.07**	187	160	1.17
	set_source_connection_agent	**265**	**3771**	**0.07**	**96**	**112**	**0.86**
	there_exists	215	104	2.07	67	2	33.83

Table 2: Results for two of the tested classes, showing the time and number of tests required by ARTOO and RAND to uncover the first fault in each routine in which they both found at least one fault, and their relative performance. In most cases ARTOO requires significantly less tests to find a fault, but entails a time overhead.

Results

Class-Level Comparison

Class	Tests to first fault			Time to first fault (seconds)		
	ARTOO	RAND	ARTOO/RAND	ARTOO	RAND	ARTOO/RAND
ACTION_SEQUENCE	**293.72**	**3449.76**	**0.09**	131.53	95.11	1.38
ARRAY	**437.19**	**856.39**	**0.51**	133.21	21.23	6.27
ARRAYED_LIST	**206.80**	**3317.80**	**0.06**	122.16	113.42	1.07
BOUNDED_STACK	**282.50**	**357.17**	**0.79**	128.00	11.45	11.18
FIXED_TREE	**333.99**	**463.91**	**0.71**	**127.73**	**136.64**	**0.93**
HASH_TABLE	**581.21**	**2734.42**	**0.21**	164.41	65.85	2.49
LINKED_LIST	**238.20**	**616.71**	**0.38**	98.39	18.14	5.42
STRING	**279.64**	**1561.60**	**0.17**	**85.03**	**144.28**	**0.58**
Overall averages	*331.66*	*1669.72*	*0.19*	*123.81*	*75.77*	*1.63*

Table 3: Averaged results per class. ARTOO constantly requires fewer tests to find the first fault: on average 5 times less tests than RAND. The overhead that the distance calculations introduce in the testing process causes ARTOO to require on average 1.6 times more time than RAND to find the first fault.

ARTOO:

- Fewer Tests
- Faster Fault Detection
- Slower for Simple Tests
- Ideal for Complex Software

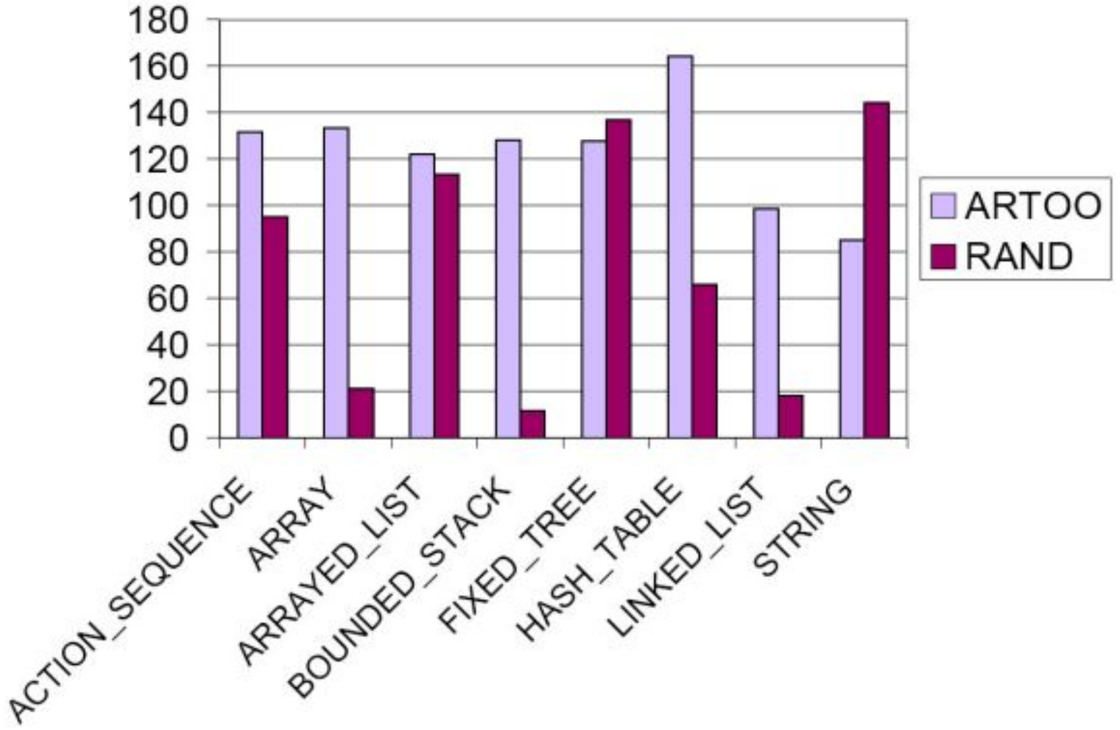


Figure 3: Comparison of the average time to first fault required by the two strategies for every class. RAND is generally better than ARTOO.

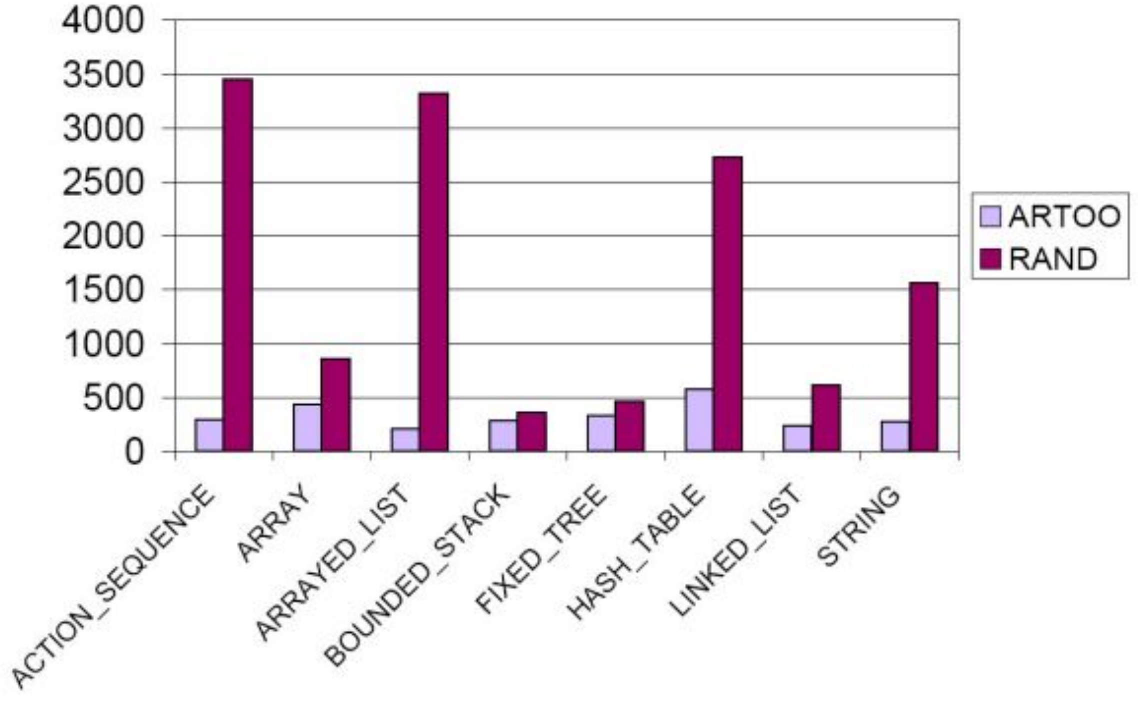


Figure 2: Comparison of the average number of tests cases to first fault required by the two strategies for every class. ARTOO constantly outperforms RAND.

Class	Routine	Tests to first fault	Time to first fault (seconds)	#faults
ARRAYED_LIST	remove	167	46	1
FIXED_TREE	child_is_last	717	283	1
FIXED_TREE	duplicate	422	134	1
STRING	grow	492	163	2
STRING	multiply	76	17	2

Table 4: Faults which only ARTOO finds

Class	Timeout (minutes)	StDev(NumberFoundFaults) ARTOO	RAND
ACTION_SEQUENCE	1	1.87	2.92
	2	1.14	2.59
	5	0.89	1.22
	10	1.64	0.71
ARRAY	1	2.30	13.22
	2	2.45	16.81
	5	2.77	17.04
	10	5.27	17.04
ARRAYED_LIST	1	2.95	1.52
	2	3.08	3.51
	5	3.81	8.37
	10	4.93	12.60
BOUNDED_STACK	1	2.35	1.10
	2	3.56	0.84
	5	3.11	1.22
	10	2.17	1.48
FIXED_TREE	1	2.30	3.91
	2	1.30	2.70
	5	1.64	2.70
	10	2.59	2.17
HASH_TABLE	1	0.89	2.12
	2	1.64	2.05
	5	2.05	5.15
	10	3.11	7.91
LINKED_LIST	1	0.55	1.48
	2	0.45	1.79
	5	1.34	2.17
	10	1.14	4.22
STRING	1	2.07	1.14
	2	3.13	0.44
	5	3.7	1
	10	3.91	0.83
Average		*2.37*	*4.49*
Standard deviation		*1.19*	*5.14*

Table 5: Standard deviations of numbers of found faults for each strategy due to the influence of the seed for the pseudo-random number generator, and the average and standard deviation of the standard deviations for each strategy. ARTOO is generally less sensitive to the choice of seed.

Results

ARTOO

- Finds faults RAND misses within the same test duration
- Valuable for detecting overlooked faults by RAND
- Intelligent test case selection with fewer, targeted tests
- Less sensitive to seed variations, ensuring consistent performance

RAND

- Exhaustive search finds faults ARTOO might miss
- Complements ARTOO by addressing detection gaps
- Using both ARTOO and RAND enhances fault detection and reliability

Optimizations

Cluster Center Distance Calculation → Improves time to first fault by 25%

Manual Test Guidance
- → Reduces tests to first fault by 2x
- → Based on closeness to manual inputs and distance from automatic inputs

Future Enhancements
- → Explore less intensive distance calculations
- → Incorporate semantics for better accuracy

Conclusions

Thank You