Budget-aware Random Testing with T3

Benchmarking at the SBST2016 Testing Tool Contest

Wishnu Prasetya, Utrecht University

http://www.cs.uu.nl/~wishnu

https://git.science.uu.nl/prase101/t3/wikis/home

T3

- Random testing tool for Java Class
  - Provide convenient way for user to specify custom test data/generator
- Typical use case:
  - to fastly generate large amount of test sequences
- Test suites can be generated interactively
  - combined interactively: suite = suite1 + suite2
  - interactive query
  - analyzed, e.g. to infer invariants

Querying test suite

- H = **hoare**({s → s.arg[0] ≤ s.tobj.cutOff ()},
  ”calculateTax”,
  {s → s.retval == 0})

- **ltlquery**(suite).**with**(always(H)).valid()
- **filter**(suite).**with**(eventually(H.antecedent()))

Budget aware suite generation

- Use case: running automated testing on a whole project, with an overall budget e.g. 1 hour.
- Current implementation: pre-calculated fixed budget per class, e.g. 1 minute.
- Class-level budgeting:
  - over inner classes
  - over test goals per target class

Test Goal

- Test goal: a public/protected method of CUT. → generate a test suite for it.
- All TGs are put in a **worklist**, to be processed in some order
  - Process TG m: generate/refine its suite. If **not done**, put m back in the worklist.
    - There is a limit on the max. number of this put-backs (in the competition: set to 8)
  - Repeat until either worklist is empty, or we run out of budget.

Refining suites

- Let m be a TG. We maintain a test suite S_m, generated for m so far. Generate new set of test sequences, each of the form:

$$\sigma \;++\; \text{o.m(...)} \;++\; \tau$$

- Only add a new sequence to S_m if it improves coverage.
- Keeping in mind: proportionality.

Generating prefixes

- For efficiency, prefixes are generated collectively and incrementally over all TGs
- Maintain a set P of prefixes so we have so far, and only grow it incrementally :
  - If all TGs of generation *k* are processed, and worklist is not empty, we grow P by generating K fresh prefixes, but only adding those than can **refine** P.
- Refinement: also keep track “unique” object structures
  - project object structures to trees
  - project primitive values to logarithmic representations

Processing order policy of the TGs

- Random?
- Used policy:
  - when budget is still ok (0.5 B), we just pick the next TG randomly
  - after that “easier” TG is favored.
  - linear over generations, to enforce fairness

Overall budget policy

- CUT-level dynamic budget allocation:
  - Given a CUT and time budget B0, determine the set of classes in CUT to target. Each C gets is allocated a fragment of B0, proportional to its complexity.
  - When we are done with C, budget allocation is re-calculated based on remaining time at that moment.

- T3 is tuned to use budget considerately, and not aggresively trying to exhaust all budget.

Result

	60s			120s			240s			480s	
	C	M	T	C	M	T	C	M	T	C	M	T
RAN	54.0	64.1	1439	57.2	67.2	2785	59.7	68.8	5493	62.3	70.5	11181
T3	59.2	74.4	1062	63.6	76.9	1579	64.8	77.9	2052	65.5	78.0	2780
EVO	44.1	63.1	1410	50.2	69.5	2601	60.6	80.0	4870	65.5	83.4	8805
JT	63.5	72.5	1653	68.1	79.9	2832	69.3	79.5	5143	70.8	84.4	9435

On subset of 22 CUTs of the original 80 CUTs in the SBST2016 benchmark, on which no tools crash, and on which the benchmarking tool itself has no issue.

Productivity

	60s	120s	240s	480s
RAN		0.14 (7)	0.06 (18)	0.03 (36)
T3		0.51 (2)	0.15 (7)	0.06 (17)
EVO		0.31 (3)	0.28 (4)	0.07 (13)
JT		0.23 (4)	0.03 (32)	0.02 (48)

productivity = additional % cover- age gained per additional minute spent.

Conclusion & future work

- When budget efficiency matters, enforcing a budget control algorithm makes sense.
- On big budget, T3’s BCA is justified to stop its effort.
- On low budget, T3’s BCA stops too early. Future work: smarter BCA.
- Future work: project-level BCA.