Experimental assessment of Random Testing for Object Oriented Softwares

http://portal.acm.org/citation.cfm?id=1273476

1. Introduction

- Automated testing much required when testing huge softwares.
- Random Testing is one of the strategies for it.
- Effectiveness of testing strategies should be evaluated on basis of hard experiment evidence and not just intuitions or priori arguments
- literature often deprecates because of preconceptions - claim intuitional rather than experiment basis [18]
- advantages of RT
    - simplicity of implementation
    - speed of execution
    - absence of human bias
- Authors analysed the effectiveness of RT by intensive experiments.
- Testing strategies assessed – "how many bugs found and how fast?"
- Authors assessed the effectiveness of RT on a significant code base which had 2 properties: it is effectively used in production and has many bugs.

2. Test Bed

- AutoTest - tool used for running experiment
- Implemented as a framework so that the various strategies of the input value generation can be easily plugged in - configurable random strategy
- Tested on Eiffel applications equipped with "contracts".

2.1 Input generation

- Two ways to do it
    - Constructive manner – call the constructor of the targeted class and then optionally call other methods to change the state
    - Brute force manner – allocating obj and setting fields directly
        - Disadvantage
            - Obj created this way can violate its class invariant

- Autotest employs the first approach.
- Autotest keeps a pool of objects available for testing. Objects used for testing and returned to the pool after testing.
- Input generation algorithm steps
    - Given a method *m* of class C currently tested.
    - To test *m*, a new obj and arguments for m are required
    - With probability $\mathbf{P_{GenNew}}$ either new obj created or one selected from the pool

- If new object, Autotest calls randomly one of the constructors of the class. (if abstract class of the first non abstract descendant)
- The arguments are generated randomly with probability **($P_{GenBasicRand}$)** from a set of values or from a set of predefined special values (which has high bug revealing rate)
- The object pool objects should be brought to states that are more factual. **P_{Div}** defined as probability go a diversification operation to take place. (Example: List class object will be in a state when constructor is called. But that is not enough. We need to also consider the various states it'll be in with various diversification operations performed like add, delete nodes).

2.2 Test execution

- AutoTest used two process approach
  - Master process – implements the actual test strategy
  - Slave process – test execution
- Slave gets small commands from master like create the obj, method call etc.
- This approach has adv of robustness: in case there is failure in the execution, the entire testing process need not be redone form beginning. Only the execution part needs to be restarted.

2.3 Automated oracle

- Autotest uses contracts (method pre- and post conditions and class invariants) present in the code as an automated oracle.
- In Eiffel these contracts are Boolean exp.
- These contracts contain specification of the s/w and can be evaluated at runtime.
- AutoTest checks these and reports contract violation.
- Authors look for faults, not failures. One fault can lead to more than one failure which they consider the same. So results as number of faults. Also refer to as bugs.

2.4 Experiment Setup

- ISE Eiffel compiler version 5.6 - 32 identical machines - Each dual core Pentium III - 1 GHz - 1GB - Fedora Core 1
- Classes chosen to be tested – from Eiffel libraries, some student written classes in course and some mutated classes.
- Tested each class for 30 mins with three diff seeds of pseudo random generator for combinations of
- P_{GenNew} (probability of creating new obj as input rather than using existing one) {0,.25,.5,.75,1}
- P_{Div} (calling procedure on an obj chosen randomly from pool after running each test case) {0,.25,.5,.75,1}
- $P_{GenBasicRand}$ ([rob of generating values for basic types randomly rather picking from predefined special set of values) {0, .25, .5, .75, 1}
- So - 3 seeds for 5 of each => 3*5*5*5 =375 test runs per class for 30 mins. Total 1500 hours.

- Hence, for each combination of class, seed, timeout and probability values the “total number of bugs” is obtained.
- “Number of bugs found in a set of time” focussed for results.

3. Results

Tried to answer 5 questions.

1. **How does number of found bugs evolve over time?**
    a. Highest number of bugs (averaged over 3 seeds) for each timeout for every class
    b. Evolution was inversely proportional to elapsed time
    c. Best fit curve f(x) = a/x+b
    d. Figure 3, 4 and 5
    e. Parameters quantifying goodness of fit
        i. Sum of squared error
        ii. R-Square
        iii. Root Mean Square Error

2. **How much does the seed influence the number of found bugs?**
    a. Results show that seed has high influence.
    b. For HASH_TABLE one see showed 5 bugs and another showed 23 (for same time and same combination of probabilities)
    c. Table 3
    d. No correlation between the maximum differences that the seed can make in number of found bugs and the testing timeout.
    e. Should not lead to believe that a certain seed value constantly delivers better results than another
    f. Instead results indicate that RT needs to be performed several times with various seed values to compensate for high variability.
3. **How much do the values of probabilities influence the number of found bugs for every timeout**
    a. Table 4
    b. Results show min num of bugs stay constant or increase very little with increase in timeout
    c. In other words – there exists several combinations of probabilities which perform badly compared to others
    d. For all classes (except PRIME) the value of 0 for probability of generating new object delivered bad results.
    e. Probability of 1 for generating the basic values randomly had bad results too
    f. Performance of random testing can vary widely with the combination of probabilities chosen

4. **Which version of random generation algorithm maximises number of found bugs?**
    a. Goal of this analysis is to find the combination of probabilities that maximises the number of bugs found over all tested classes, and over all timeouts and then individually per class and per timeout.
    b. Since seeds influence the results, seeds are averaged over 3 seeds
    c. Best combination of probabilities averaged over all classes and timeouts – C_0 – $P_{GenNew0}$ =0.25; P_{Div0} =0.5 $P_{GenBasicRand0}$=0.25
    d. Here average percent of bugs lost is 23% - compared to highest number of bugs found (optimal combination of probabilities for that class and timeout).
    e. If low timeouts were excluded, C_0 misses 44% of bugs compared to the best of 43%.
    f. The low timeout values introduce high level of noise due to less number of tests performed.
    g. Grouping results class-wise – Table 5.
    h. Shows for every class the combination of P_{New} and P_{Basic} that found highest no. of bugs. If the second highest was not much different from first, even that is listed. Question mark for inconclusive results where diff probabilities found same no of bugs throughout.
    i. Results show that the most effective probabilities differed for different classes.
    j. But the combination doesn't change or changes very small for different timeout values for each class.
5. **Are more bugs found due to contract violation or due to other exceptions being thrown?**
    a. Generally Contracts written along with implementation
    b. Eiffel follows same practice, though their contracts are usually weaker.
    c. Since contracts are used as oracle here, conditions not expressed in them cannot be checked and bugs will be missed.
    d. So uncaught exceptions were also considered to signal bugs.
    e. Fig 8 shows bugs thru contracts and thru other exceptions for STRING class.
    f. Values in the graph are average number of bugs found with different version of the Random also.
    g. Most other classes had similar graph.
    h. Fig 9 – is different. For class BOUNDED_STACK, contract based bugs were always greater than those found thru exceptions.

6. Discussion

6.1 Methodology

- Contracts and Exceptions used as oracles.
- Bugs which do not trigger contract violation or throw exceptions are missed. Because of the size of the study, manual evaluation was impossible.

- Chances of false positive. A contract violation signals bug, whether it is in the contract or even in the implementation. Bugs in the specification are also important. Thus wrong assumptions made by programmers will be caught here.

6.2 Threats to Validity

- Testing higher number of classes naturally would increase the reliability of the results
- The authors could not test many classes.
- But the classes were selected form different sources.
- Still, these classes did not show all the kinds of bugs that a typical OOS would exhibit.
- Higher number seeds could have improved the robustness of the results.
- Authors claim that the greatest threat to the validity of results would be due to existence of bugs in the testing tool. But they weren't aware of any during the study.

7. Conclusion

- Extensive case study performed
- Evaluating the performance of random unit testing on OO applications over fixed timeout.
- To determine the best practice for RT strategy.
- Contracts and Exceptions used as oracle.
- For smaller timeouts(1 and 2 mins), exceptions found more bugs but situation reversed for higher timeouts(5, 10 and 30 mins)
- The study evaluates the implementation choices on the input generation algorithm
    - Frequency with a new obj is created w.r.t selecting from the pool of already created objects ($P_{GenNew)}$
    - Frequency of basic values (parameters) to be generated or used from predefines values set. ($P_{GenBasicRand)}$
    - Frequency with which the object pool is diversified. ($R_{Div)}$
    - 0.25, 0.25, 0.5 combination gave best overall results.
- The values of the probability combinations had an impact on the results
- High number of bugs found in the first few minutes. Thus, RT strategy is a good candidate for tests over short timeouts.
- Comparison of RT with other testing strategies still not addressed.

ARTOO – Adaptive Random Testing for Object Oriented Softwares

http://portal.acm.org/citation.cfm?id=1145744

1. Introduction

- Efficiency of RT is improved if inputs are evenly spread
- Adaptive Random Testing (ART) follows this idea – evenly spread input space.

- ART is based on the intuition that a non point type of failure is more likely to be detected by an evenly spread test case than ordinary RT.
- ART was initially proposed for numerical inputs and a concept of distance was introduced.
- ARTOO is ART for OOS
- Is an extension of this idea where the notion of distance between objects is brought in.
- Selects an input object that has the highest average distance to that of the tested objects.
- A feedback system
- ART was shown to reduce the number of tests required to reveal the first fault by as much as 50% over RT.

2. Object Distance

- Measure of how different two objects are
- Generally objects are characterized with:
  - Values
  - Dynamic types
  - Primitive values of the attributes,
- Object distance takes into account 3 dimensions:
  - **Elementary distance** – difference between the direct values of the objects (values of the references in case of reference types and embedded values in case of primitive types)
    - **For numbers:** F(|p-q|) where F is a non decreasing monotonic function with F(0)=0
    - **For character**: 0 if identical, C otherwise
    - **For Boolean:** 0 if identical, B otherwise
    - **For String:** Levenshtein distance
    - **For references:** 0 if identical. R if none are void, V if one is void.
    - **C, B, R, V** conventionally chosen positive values.
  - **Type distance** – difference between objects' type; independent of the value
    - Is an increasing monotonic function of the sum of their path lengths to any closest common ancestor and the number of their non-shared features(features not inherited from the ancestors)
    - Eiffel (ANY class) , Java (java.lang.Object class)
    - type_distance(t,u)= λ* path_length(t,u) + $\sum_{a\ non\text{-}shared(t,u)} weight_a$
  - **Field distance** – measure of difference between object's individual fields
    - Recursively applying the distance calculation of all matching pairs of fields
    - Field_distance(p,q) - $\sum_a weight_a * (p.a <-> q.a)$
- P <-> q = combination(elementary_distance(p,q), type_distance((type(p),type(q)), field_distance({[p.a<->q.a] | a belongs to Attributes(type(p), type(q))}))
- combination() would be a weighted sum of the three components(normalised)

3. Adaptive Random Testing for OO software (ARTOO)

- algorithm proposed which keeps track of available and already used objects
- always chooses the input from the available set of objects that has highest average distance from already used objects
- Algorithm : Figure 1

4. Implementation

- ARTOO implemented as a plugin for AutoTest
- AutoTest performs automatic unit test of Eiffel code.
- Takes contracts in the Eiffel code as oracle to report fault. Considers exceptions thrown too.
- AutoTest, by default, has its own RT strategy. But a new strategy can be plugged in.
- ARTOO just affects the usual testing process of AutoTest in the selection of input process
- Sections 4.2 and 4.3 discuss about incorporating ARTOO with AutoTest
- Example in Section 4.4 of class BANK_ACCOUNT

5. Experiments

5.1 Setup

- Subject of evaluation – classes fro Eiffel Library 5.6(industrial grade library)
- No changes made to the library
- Table 1 lists the classes selected.
- Tests run with ISE Eiffel compiler 5.6
- On Pentium M - 2.13 GHz – 2 gb – Windows XP SP2
- Test applied both directed RT strategy and ARTOO strategy
- Reminder: as seen in the example RAND is not purely a random strategy. The input value is made to be chosen from a selected set of data that is assumed to be more probable in revealing a bug.
- Results evaluated based on
    - Number of tests to first fault
    - Time to first fault

5.2 Results

- Table 2 – routine by routine comparison of RAND and ARTOO for two classes
- The values are averages over 5 seeds
- Table 3 shows a class wise comparison (averaging the values of all the methods in each class)
- Figure 2 and 3.
- ARTOO successfully reduces the number of test cases required to find faults
- But the distance calculation is time consuming
- So ARTOO needs fewer tests and RAND needs less time to find first fault.
- Table 4 – list classes/routines where only ARTOO finds a fault and RAND fails to.
- Also true that RAND finds fault which ARTOO does not.
- Results show that ARTOO is less sensitive to variation of seed value – so performance more predictable; so we have better control.
- Table 5 shows the standard deviation of no. of faults found by the two strategies for 4 timeouts.

5.3 Discussion

- Results suggest that both the strategies have different strength – so automated testing should use a combination of both for best results
- Previous paper showed that the rate of finding faults, by RAND, is inversely proportional to time elapsed. So ARTOO can be put to use at this point when rate of finding faults decreases in RAND.
- Classes considered here are not computation intensive. For computationally intensive softwares, the number of tests that can be run over a given period of time will be less
- In such cases ARTOO will be favoured.

5.4 Improvements

- Since we have the concept of distance here, one can employ clustering techniques to group the objects
    - ARTOO can be optimized then by computing the distance to the cluster centres alone.
    - A preliminary implementation shows improvement of time to find first fault by 25%.
- Use information contained in the manual tests for further guide search of fault revealing input.
    - Manually written tests have inputs that are likely to reveal faults than RT.
    - So, the usual random way of selecting input is to be followed, but two measures are assessed
        - How far is the new object from already used classes
        - How close is it to manually used inputs
    - Preliminary experiment show that it can reduce number of tests to the first fault by average factor of 2 compared to basic ARTOO
- ARTOO follows full definition of object distance. A less computationally intensive definition can reduce the time.
- Definition of object distance takes into account only the syntactic form of the objects. Considering semantic forms can improve accuracy.
    - Human intervention will be required in this case.

6. Conclusion

- Testing strategy called ARTOO introduced.
    - Implemented, evaluated with experiment results
- Based on the idea of Adaptive Random Testing (ART)
    - Selecting inputs farthest from the already selected ones
- ART was introduced only for numeric values, but authors find means to apply it on OO software by defining Object Distance
    - Elementary distance between the direct values of the objects
    - Distance between the types pf the objects
    - Distance between the fields of the objects.
- Experiments comparing ARTOO with RAND.
- ARTOO significantly reduces the number of tests required to reveal first fault than RAND (average by factor of 5)
- But it consumes considerable amount of time to do so, in comparison with RAND (on average 1.6 times slower)
- Improvements
    - Considering manual inputs
    - Clustering of objects
- Results show that ARTOO revealed many faults that RAND failed to detect.

- ARTOO is less sensitive to the seed of the random test generator, than RAND
