Notes for ENGR489

Design and Use of QuickCheck

<https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html>

Generate random test cases to falsify program properties.

Need size parameter (maximum number generated) and input value of a specific type

**Property-based testing**

A property of a program is an observation that we expect to hold true regardless of the program’s inputs. However property-based tests approximate deductive confidence with less work by checking properties for a finite number of randomized inputs called test cases.

For inputs - How long should the list be? How large should the integer be? Size parameter for bounding.

Results to store when the test is finished:

* Test status
* Statistics
* Reason for failure and input used
* Exceptions thrown

## C++ version

<http://software.legiasoft.com/quickcheck/>

There are two types of data generators:

General purpose data generators are associated with a type and allows the generation of values of this type at random.

Special purpose data generators are associated with a property and allows the generation of randomly generated arguments for this property.

Need generators for all primitive types.

Could extend to enable users to write their own generators

## ArbitCheck: A highly automated property-based testing tool for Java

Dealing with user-defined types

QuickCheck tells the user to make you a custom generator for user-define types -> Return only validv values for the type.

However, user-defined generators sometimes have troubles when (1) the generator itself contains a bug, or (2) there is a gap between the valid values and the possible values for a type. Both leads to an unnecessarily limited distribution of the generated values, and inability to find faults that would be revealed by the values unnecessarily filtered out.

For (2), difficult to write a generator without knowing the existence of the bugs. Generator that reveals both bugs cannot be used after bugs are fixed.

**Feedback-directed random test generation**

Input = list of classes, output = sequence of statements. A sequence corresponds with a test case and has class to public constructors and methods from class list -> Objects created and mutated in each test case.

Extends existing sequences or stop extending if exception thrown.

Blackbox, no static/dynamic analysis required.

## QuickCheck: A lightweight testing tool for Random Testing

Only use test cases that satisfy the first condition (precondition) e.g. x <= y 🡺 max x y == y

Custom test generator to control the distribution of input values. Automatic test generator may skew the inputs to e.g. have small lists, or small integer values.

A generator is a function which can generate *a* in a pseudo random way.

User defined types – Look into pre-processor and polytypic programming (A polytypic (or generic) program captures a common pattern of computation over different datatypes by abstracting over the structure of the datatype. Examples of algorithms that can be defined polytypically are equality tests, mapping functions and pretty printers.)

Programmer limits the size in custom test generator

Begin testing property on small test cases then increase size bound as testing progresses. Why? Greater variety of tests cases -> more effective test and improves change of finding enough test cases to satisfy precondition. Also more likely to find small counter example.

Function generator and result generator??? Possibly due to functions being first class, functions can be passed as an argument into other functions.

Systematic methods often use a test adequacy criterion e.g. 100% test code coverage, QuickCheck’s implementation does not consider this.

To generate more complex data, need to know description of the data’s structure.

Context-free grammar cannot express all desired properties of test data. Termination problem with recursive grammars.

Size bound VS probability of termination. Cannot have non-termination as a test result as cannot compute it

Why is it good? Encourages programmers to write formal specification and therefore understand their programs better. Short-term payoff is automated testing and some confidence properties hold.

No test coverage measurement.

## QuickCheck Testing for fun and profit

Most of the time, trying to understand why a complex case failed or simplifying a failing case by hand. QuickCheck automatically shrinks failing test cases so it reports a minimal set of sequences that caused a failure. Makes it easier to locate a fault.

Testing functions with side effects -> Made a state machine like version(DIFFICULT). Base test generation on state machine modelling. User writes a module exporting the number of callbacks which tell QuickCheck e.g. initial\_state() how the state machine is supposed to behave.

Initial\_state () -> #state{pids=[], regs=[]}

States represented as a record of components.

Tests start from initial state and generate sequence of commands (command generator) until it generates the atom stop. Need precondition for each command.

User defines callback functions: command, precondition, postcondition, initial\_state, next\_state

Shrinking failing test cases is a powerful diagnostic technique, due to Hildebrandt and Zeller [11], who used it, for example, to shrink a test case that crashed Mozilla from 95 user actions on a web page consisting of almost 900 lines of HTML, to three user actions on one line of HTML ! Their *delta debugging* method starts from *two* tests, a successful one and a failing one, and uses a generic algorithm to search the space between them for two most-similar tests, one successful, and one failing. QuickCheck searches only from a failed test, towards smaller test cases, but using arbitrary user-defined shrinking methods.

Changing actual program to test boundary cases e.g. reducing constant values.

Combining concrete and symbolic execution, generate constrains on path

## DART: Directed Automated Random Testing

Static source code parsing, random testing, dynamic analysis of program behaviour and automatic generation of new test inputs to direct systematically the execution along alternative program paths.

1. *automated* extraction of the interface of a program with its external environment using static source-code parsing;

2. automatic generation of a test driver for this interface that performs *random* testing to simulate the most general environment the program can operate in; and

3. dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to *direct* systematically the execution along alternative program paths.

Other way to check correctness is code inspection (like Lint, FindBugs).

DART uses a directed search to dynamic gather knowledge about program’s execution. Gathers the path constraints (conditions) after first test and tries to go through the path in subsequent tests.

DART uses random inputs and also uses path constraints. Requires using RAM

Each run (except the first) is executed with the help of a record of the conditional statements executed in the previous run. For each conditional, we record a *branch* value, which is either 1 (the *then* branch is taken) or 0 (the *else* branch is taken), as well as a *done* value, which is 0 when only one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is 1 otherwise. This information associated with each conditional statement of the last execution path is stored in a list variable called *stack*, kept in a file between executions.

If theorem prover cannot decide symbolic condition (e.g. x > 0), then just replace with concrete value. Dynamic test generation and instrumenting the program to check whether the input values generated next have the expected effect on the program.

Symbolic execution limited in practice due to imprecision of static analysis and of theorem provers.

## CUTE: A Concolic Unit Testing Engine for C

Random testing may lead to inputs with same behaviour therefore is redundant, probability of selecting particular inputs that detect buggy behaviour may be small.

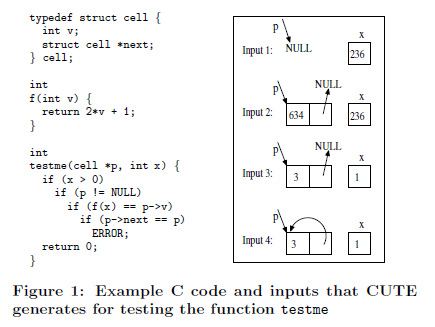
Symbolic execution, program is executed with symbolic variables instead of concrete values as input.

Goal: Generate concrete values as inputs to explore different execution paths.

Implements a solver for arithmetic and pointer constraints to incrementally generate test inputs.

High branch coverage.

Logical input map represents a memory graph. Maps logical address to values which are primitive values or logical addresses. E.g. logical input for Input 4 is (3,1,3, 0) . 1st value is address of p, 2nd value is x, 3rd value is p->v, 4th value is p->next (0 represents null).



Tries to explore all execution paths of a program

Bounded, depth-first search.

Replaces symbolic expression with concrete value if the constraint cannot be solved.

# Considerations for designing Whiley version of QuickCheck

Property-based testing. Need program's contract to be able to test? Precondition == what the inputs should be. Postcondition == whether the test was successful or not?

Generator to randomly create inputs needed for a test

Need to guarantee data is not infinitely generated i.e. will terminate

Detecting errors in specification and program == OK. Error in test generator == BAD

Whether to generate the test case if it fails

Pointers/References

JCrasher – execute tests until an exception is thrown.

User defined types

* Get the user to write a custom generator
* Feedback-directed random test generation

Results to store when the test is finished:

* Test status – Passed, failed
* Statistics – Percentage that passed or failed
* Reason for failure and input used
* Exceptions thrown

Testing approaches

* Random testing e.g QuickCheck
* Swarm testing
* Adaptive random testing
* RecGen
* Dynamic Symbolic execution
* Conocolic testing
* Partition testing

Features

* Be able to classify test properties?

# Other notes to consider for report

Testing is expensive. Spend a lot of time and money testing, can also be tedious.