Logic-based Software Testing and Verification

Emanuele De Angelis

Istituto di Analisi dei Sistemi ed Informatica
“Antonio Ruberti”
Consiglio Nazionale delle Ricerche

Software And Knowledge-based Systems

joint work with: **Fabio Fioravanti**[1], **Adrián Palacios**[2], **Alberto Pettorossi**[3], **Maurizio Proietti**[4]

[1] University of Chieti-Pescara “G. d'Annunzio”, Italy

[2] AWS, USA

[3] University of Rome “Tor Vergata”, Italy

[4] IASI-CNR, Italy

What's about this (short) talk?

Logic-based Software Testing and Verification

Constrained Horn Clauses

Property-based Testing

Contract-based Verification

Example-based Testing

automation level 1 – test cases execution

a developer/test engineer designs a collection of examples of the program behaviour in the form of pairs < **input, expected output** > and compares the **expected output** against the **actual output**

```
int max(int a[], int n) {
  int i = 1;
  int m = a[0];

  for (i = 1; i < n-1; i++)
    if (a[i] > m)
      m = a[i];

  return m;
}
```

Input a , n	expected output	actual output
[60,1], 2	60	60
[-3,110,1], 3	110	110
[9], 1	9	9
[7,3,23,42], 4	42	**23**

Example-based Testing

automation level 1 – test cases execution

a developer/test engineer designs a collection of examples of the program behaviour in the form of pairs < **input, expected output** > and compares the **expected output** against the **actual output**

Input a , n	expected output	actual output	
[60,1], 2	60	60	✓
[-3,110,1], 3	110	110	✓
[9], 1	9	9	✓
[redacted]			

Straightforward process,
but does not give very high
guarantees of program correctness

Property-based Testing

automation level 2 – test cases generation & execution

instead of designing specific examples,
a developer/test engineer provides a
specification for program **inputs** and **outputs**

> Given any array of integers `a` of size `n`,
> `max(a,n)` is the largest element of `a`

Automated generate & test approach

1. randomly generate an **input** that meets the specification
2. run the program with that input
3. test whether or not the **output** meets the specification

Property-based Testing

automation level 2 – test cases generation & execution

So far, so good – to generate simple test inputs (e.g., lists of integers)

> Given any array of integers `a` of size `n`,
> `max(a,n)` is the largest element of `a`

SIMPLE

but what happens if the inputs have **constraints** on

- the **content** of data structure (e.g., sorted lists)
- the **shape** of the data structure (e.g., balanced trees)
- **both** (e.g., AVL trees)

Generating (complex) inputs from (complex) specifications

```
ordered(L) -> case L of
  [A,B|T] -> A =< B andalso ordered([B|T]);
   _ -> true
end.
```

<<== ordered list

- constraints on the elements

```
avl(T) -> case T of
  leaf -> true;
  {node,L,V,R} -> B = height(L) - height(R)    andalso
                  B >= -1  andalso  B =< 1     andalso
                  ltt(L,V) andalso  gtt(R,V)   andalso
                  avl(L)   andalso  avl(R)     ;
  _ -> false
end.
```

<<== AVL tree

- binary search tree
  constraints on the elements
- height-balanced
  constraints on the shape

Random generation of inputs from the specification can be quite **inefficient.**
State-of-the-art approaches use ad-hoc, hand-written, generators.

Property-based Testing

automation level 3 – generator of (test cases) generators

Constrained Horn Clauses (CHCs)
(a fragment of First Order Predicate Calculus)
to **formalize specifications**

Operational Semantics of CHCs : Constraint Logic Programming (CLP)
CHC specifications >> runnable specifications
>> test cases generators

Smart Interpreter for CLP specifications : Constrain & Generate
(allows for a finer control of the test cases generation process)
we get an efficient generator of (test cases) generators

A glimpse of the generation process ...

```
ordered_list(L) :-
  typeof(L,list(integer)) ,
  eval(apply('ordered',[var('L')]),[('L',L)],lit(atom,true)) .
```

generetes a non-ground CLP term representing a list of integers

enforces constraints on the list (ascending order of its elements)

```
?- ordered_list(L).
L = nil ;
L = cons(lit(int,X),nil), X in inf..sup ;
L = cons(lit(int,X),cons(lit(int,Y),nil), Y #>= X ;
L = cons(lit(int,X),cons(lit(int,Y),cons(lit(int,Z),nil))) , Y #>= X, Z #>= Y
```

Verification

Certifying Correctness

- **testing,** by exercising the program on specific inputs, can only **find bugs**

- **verification** aims at proving that there are **no bugs for any input** (even in infinite domains)

Contract-based Verification

Certifying Correctness

Specification S
+ Program P

```
def rev(l: List[BigInt]): List[BigInt] = {
  require(is_asorted(l))
  l match {
    case Nil() => Nil[BigInt]()
    case Cons(x, xs) => snoc(rev(xs),x)
  }
} ensuring { res => is_dsorted(res) }
```

precondition of `rev`

function contract

Specification S – collection of function contracts

postcondition of `rev`

SW WORLD

LOGIC WORLD

CHCs Generator → CHCs Transformer → CHCs Solver

SAT >> **P satisfies S**

UNK (undecidable prob.)

UNSAT >> **P violates S**

find a counterexample (i.e, a bug)

to conclude …

A few pointers

- Verifying Catamorphism-Based Contracts using Constrained Horn Clauses
  E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti
  Theory and Practice of Logic Programming, 2022
- Analysis and Transformation of Constrained Horn Clauses for Program Verification
  E. De Angelis, F. Fioravanti, A. Pettorossi,
  J.P. Gallagher, M.V. Hermenegildo, M. Proietti
  Theory and Practice of Logic Programming, 2021
- Property-Based Test Case Generators for Free
  E. De Angelis, A. Palacios, F. Fioravanti, A. Pettorossi, M. Proietti
  LNCS 11823, Springer, 2019

Program	Problems	Queries	SPACER		VeriCaT_{mq}	
			sat	unsat	sat	unsat
List Membership	2	6	0	1	1	1
List Permutation	8	24	0	4	4	4
List Concatenation	18	18	0	8	9	9
Reverse	20	40	0	10	10	10
Double Reverse	4	12	0	2	2	2
Reverse w/Accumulator	6	18	0	3	3	3
Bubblesort	12	36	0	6	6	6
Heapsort	8	48	0	4	4	4
Insertionsort	12	24	0	6	6	6
Mergesort	18	84	0	9	9	9
Quicksort (version 1)	12	38	0	6	6	6
Quicksort (version 2)	12	36	0	6	6	6
Selectionsort	14	42	0	7	7	7
Treesort	4	20	0	2	2	2
Binary Search Tree	20	24	0	10	10	10
Total	170	470	0	84	85	85

Ongoing work

CHC testing & verification of rule-based XAI models: do not test the AI model (e.g., the NN), but the explainable model of the AI model (rule-based XAI model)