Breaking Parsers: Mutation-Based Generation of Programs with Guaranteed Syntax Errors

SOLVED!

Moeketsi Raselimo, Jan Taljaard, and **Bernd Fischer**
Stellenbosch University
South Africa

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Generating Well-Formed Programs for Fuzzing and Testing

Bernd Fischer
Stellenbosch University
South Africa

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Grammar-Based Testing

Test suite construction:

```
prog  → module prio id = block .
prio  → [ num ]
block → begin (decl ;)* (stmt ;)* end
decl  → var id : type
type  → bool | int
stmt  → if expr then stmt (else stmt)? |
        while expr do stmt | id = expr | block
expr  → expr = expr | expr + expr | ( expr ) | id | num
```

grammar *G*

sentence generation

test suite *TS* ⊆ L(*G*)

```
module[1] x = begin begin end; end.
module[2] y = begin end.
module[3] z = begin x = (y); end.
module[1] z = begin x = x + y; end.
module[2] x = begin y = z; end.
module[3] z = begin x = z = y; end.
module[1] y = begin y = 1; end.
module[2] y = begin if x then begin end; end.
module[3] y = begin var x : bool; end.
module[2] z = begin var z : int; end.
module[1] x = begin while x do begin end; end.
...
```

execution

… on unit under test U

Testing:

- some test fails ⇒ L(*G*) $\not\subseteq$ L(*U*)
  - since *TS* ⊆ L(*G*)
- what about contextual constraints?

Scoping and Typing in CFGs

Usual solution: attributed grammars

- destroys conceptual simplicity
- provides unconstrained escape

Elegant solution: domain-specific mark-up languages (NaBL)

- destroys conceptual simplicity

Pragmatic solution (hack): mark-up tokens

- provides unconstrained escape