Expanding the Scope of Grammar-Based Enumerative Testing

University of California, Irvine

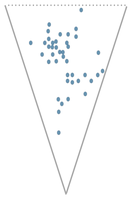
Thea U. Kjeldsmark, UC Irvine

ETH zürich

Grammar-Based Enumeration

Other testing methods

Can have problems with:



- ✗ unsystematic
- ✗ large bug triggers
- ✗ no guarantees
- ✗ miss simple bugs

Grammar-based enumeration

Enumerates smallest inputs from a context-free grammar:

- ✓ systematic
- ✓ small bug triggers
- ✓ bounded guarantees
- ✓ evolution

ET++

ET

One enumeration tool is **ET**. However, ET...

- ✗ can only be used with SMT grammars
- ✗ supports only simple grammar structures

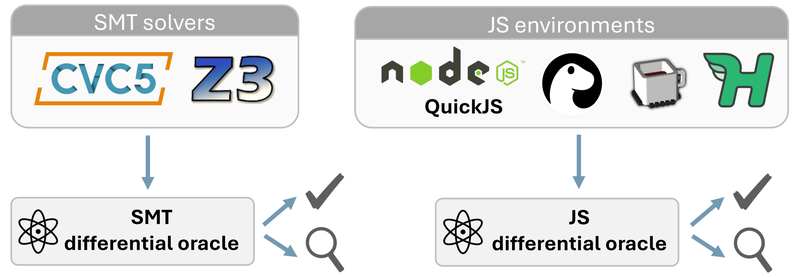
ET++

We propose **ET++**, which...

- ✓ supports grammars for arbitrary languages
- ✓ supports more complex grammar structures

Testing Setup

Differential Testing



ET++ Architecture

1. Input

ANTLR-style grammar

```
start: plus_stmt SemiC?;

plus_stmt
: var_name
| plus_stmt Plus plus_smt
;

var_name: 'INT_VAR';
Plus: '+';
SemiC: ';';
```

+

Variable declarations

```
"Variables":
{
 "INT_VAR": "var {} = 1;"
}
```

2. Translation

a. Parse the grammar using ANTLR to extract each production

b. Fix ranges and escape sequences for each rule

c. Detect recursive rules using DFS on the rule set

d. Create helper rules for quantifiers, long rules, and nested parentheses

e. Map rules to their corresponding Haskell data type (insert FEAT's pay if recursive)

3. Output

Haskell algebraic data types

```
data Plus_stmt = C0_plus_stmt Var_name
               | C1_plus_stmt Plus_stmt Plus Plus_stmt

instance Show Plus_stmt where
    show (C0_plus_stmt var_name) = show var_name
    show (C1_plus_stmt plus_stmt plus plus_stmt2) =
    show plus_stmt ++ " " ++ show plus ++ " " ++ show plus_stmt2

instance Enumerable Plus_stmt where
    enumerate =
    share $ aconcat [c1 C0_plus_stmt, pay(c3 C1_plus_stmt)]
```

N enumerated tests generated using FEAT

```
1  var INT_VAR_a = 1; INT_VAR_a
2  var INT_VAR_a = 1; INT_VAR_a ;
3  var INT_VAR_a = 1; INT_VAR_a + INT_VAR_a
4  var INT_VAR_b = 1; var INT_VAR_a = 1; INT_VAR_a + INT_VAR_b
...
```

Grammar Engineering

General vs. Specific Grammars

```
start: VARS 'console.log' ParO stmt ParC SemiC;

stmt: int_smt | bool_stmt | string_stmt;

int_stmt
: var_name_int
| var_name_array_int OpenB '0' CloseB
| ParO int_stmt (Div | Mod) int_stmt ParC
| array_stmt '.length'
| var_name_array_int '.push' ParO '1' ParC
| var_name_array_int '.unshift' ParO '2' ParC
;

array_stmt
: var_name_array_int
| array_stmt '.reverse' ParO ParC
| array_stmt '.concat' ParO array_stmt ParC
;
```

Part of the array-specific JS grammar

JS

General grammar:

ANTLR's JS lexer and parser (adjusted to limit invalid test generation)

Specific grammars:

targets specific language parts since the general grammar is prone to combinatorial blow-up

SMT

Combined type grammars: arrays/ints, reals/ints, etc.

Evaluation

Bugs Detected

Status	Z3	cvc5	Espruino	Total
Reported	1	5	8	14
Confirmed	0	4	4	8
Fixed	0	2	4	6
Duplicate	0	0	1	1
Won't fix	0	0	3	3

Using ET++, we found:

- **6 bugs** in SMT solvers
- **8 bugs** in Espruino (JS interpreter)

Type	Z3	cvc5	Espruino
Correctness	1	0	8
Crash	0	5	0

Example Bug

Espruino Bug

Test case from the array-specific JS grammar that led to an Espruino bug:

```
> var a = [1];
> console.log(a.unshift(2) / (a[0] / a
      .reverse().length));
= 4 // expected (2 / (2 / 2))
```

Future Work

For future work with ET++, we aim to...

- Do more bug search with SMT and JS grammars
- Test on additional languages
- Explore how to make the grammar engineering more systematic