Spectrum-Based Fault Localization for Context-Free Grammars

Moeketsi Raselimo and Bernd Fischer
Stellenbosch University
South Africa

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Grammars are software. Software contains bugs.

What does a bug in a grammar look like?

Specification of conditional statements for a compiler course:

> A **conditional statement** consists of the keyword **if**, a Boolean **expression**, the keyword **then**, a **statement**, the keyword **else**, and another **statement**. The **else-clause** (that is, the keyword and the statement) is **optional**.

Student's implementation:

cond → **if** expr **then** stmt **else** stmt

Grammars are software.
Software contains bugs.

```
grammar pascal;

program
   : programHeading (
   ;
programHeading
   : PROGRAM id (LP
   | UNIT id SEMI
   ;
block
   : (labelDeclPart | co
      typeDefPart   | va
      procAndFuncDec
      IMPLEMENTATIO
   ;
usesUnitsPart
   : USES idList SEMI
   ;
labelDeclPart
   : LABEL label (CO
   ;
label
   : unsignedInteger
   ;
constantDefPart
   : CONST (constant
   ;
constantDef
   : id EQUAL constar
   ;
constantChr
   : CHR LPAREN uns
   ;
constant
   : unsignedNumber
   | sign unsignedNum
   | id
   | sign id
   | string
   | constantChr
   ;
```

```
unsignedNumber
   : unsignedInteger
   | unsignedReal
   ;
unsignedInteger
   : NUM_INT
   ;
unsignedReal
   : NUM_REAL
   ;
sign
   : PLUS
   | MINUS
   ;
bool
   : TRUE
   | FALSE
   ;
string
   : STRING_LITERAL
   ;
typeDefPart
   : TYPE (typeDef SEMI) -
   ;
typeDef
   : id EQUAL (type | funcT
   ;
funcType
   : FUNCTION (formalPar
resultType
   ;
procType
   : PROCEDURE (formalP
   ;
type
   : simpleType
   | structuredType
   | pointerType
   ;
```

```
simpleType
   : scalarType
   | subrangeType
   | typeId
   | stringtype
   ;
scalarType
   : LPAREN idList RPARE
   ;
subrangeType
   : constant DOTDOT con
   ;
typeId
   : id
   | (CHAR | BOOLEAN | I
   ;
structuredType
   : PACKED unpackedStr
   | unpackedStructuredTy
   ;
unpackedStructuredType
   : arrayType
   | recordType
   | setType
   | fileType
   ;
stringtype
   : STRING LBRACK (id |
   ;
arrayType
   : ARRAY LBRACK type
componentType
   | ARRAY LBRACK2 typ
componentType
   ;
typeList
   : indexType (COMMA in
   ;
indexType
   : simpleType
   ;
```

```
componentType
   : type
   ;
recordType
   : RECORD fieldList? EN
   ;
fieldList
   : fixedPart (SEMI variar
   | variantPart
   ;
fixedPart
   : recordSect (SEMI rec
   ;
recordSect
   : idList COLON type
   ;
variantPart
   : CASE tag OF variant
   ;
tag
   : id COLON typeId
   | typeId
   ;
variant
   : constList COLON LPA
   ;
setType
   : SET OF baseType
   ;
baseType
   : simpleType
   ;
fileType
   : FILE OF type
   | FILE
   ;
pointerType
   : POINTER typeId
   ;
varDeclPart
   : VAR varDecl (SEMI va
   ;
```

```
varDecl
   : idList COLON type
   ;
procAndFuncDeclPart
   : procOrFuncDecl SEMI
   ;
procOrFuncDecl
   : procDecl
   | funcDecl
   ;
procDecl
   : PROCEDURE id (forma
   ;
formalParamList
   : LPAREN formalParamS
   ;
formalParamSect
   : paramGroup
   | VAR paramGroup
   | FUNCTION paramGrou
   | PROCEDURE paramG
   ;
paramGroup
   : idList COLON typeId
   ;
idList
   : id (COMMA id)*
   ;
constList
   : constant (COMMA cons
   ;
funcDecl
   : FUNCTION id (formalP
     COLON resultType SEI
   ;
resultType
   : typeId
   ;
stmt
   : label COLON unlabelle
   | unlabelledStmt
   ;
```

```
unlabelledStmt
   : simpleStmt
   | structuredStmt
   ;
simpleStmt
   : assignmentStmt
   | procStmt
   | gotoStmt
   | emptyStmt
   ;
assignmentStmt
   : var ASSIGN expr
   ;
var
   : (AT id | id) (LBRACK expr
LBRACK2
expr (COMMA expr)* RBRAC
   ;
expr
   : simpleExpr (relationaloper
   ;
relationaloperator
   : EQUAL
   | NOT_EQUAL
   | LT
   | LE
   | GE
   | GT
   | IN
   ;
simpleExpr
   : term (additiveoperator sim
   ;
additiveoperator
   : PLUS
   | MINUS
   | OR
   ;
term
   : signedFactor (multiplicativ
   ;
```

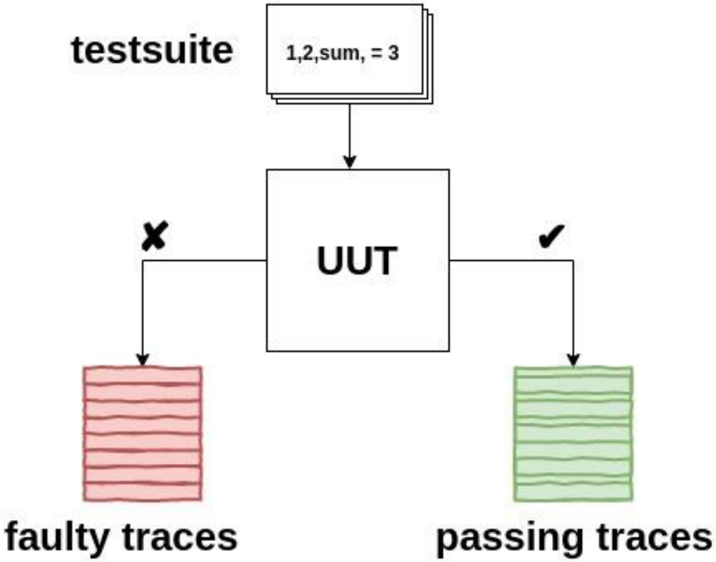
```
multiplicativeoperator
   : STAR
   | SLASH
   | DIV
   | MOD
   | AND
   ;
signedFactor
   : (PLUS | MINUS)? factor
   ;
factor
   : var
   | LPAREN expr RPAREN
   | funcDesignator
   | unsignedConstant
   | set
   | NOT factor
   | bool
   ;
unsignedConstant
   : unsignedNumber
   | constantChr
   | string
   | NIL
   ;
funcDesignator
   : id LPAREN paramList RPAREN
   ;
paramList
   : actualParam (COMMA actualParam)*
   ;
set
   : LBRACK elementList RBRACK
   | LBRACK2 elementList RBRACK2
   ;
elementList
   : element (COMMA element)*
   |
   ;
element
   : expr (DOTDOT expr)?
   ;
```

RQ: How can we automatically locate such bugs in larger (production) grammars?

Spectrum-Based Fault Localization (SBFL)

SBFL is a **heuristic**, **coverage-based**, **dynamic** method to identify faulty program elements (typically statements).

1. Execute unit under test (UUT) over test suite, collect ***coverage*** for each individual test case
2. Correlate coverage with outcomes, aggregate into ***spectrum***
3. Compute ***suspiciousness score*** for elements and ***rank***: higher scores and ranks indicate higher bug likelihood

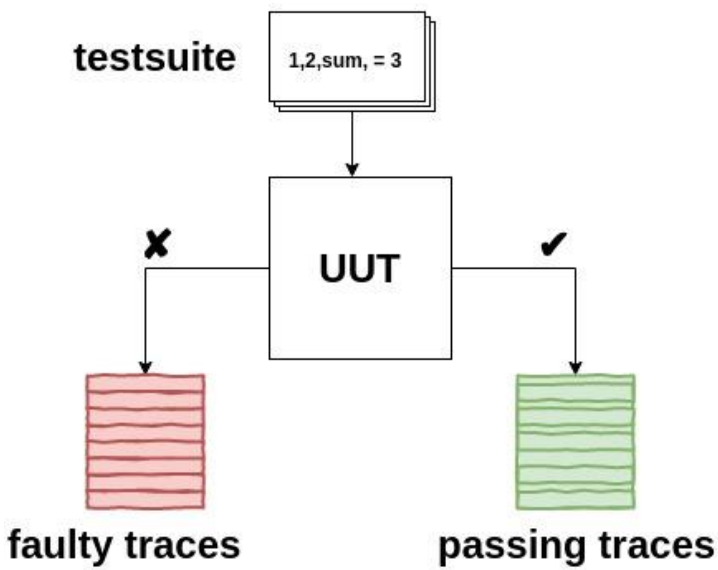


#	program	t1	t2	t3	t4	t5	t6
1	read(a);	✘	✘	✔	✔	✔	✔
2	read(b);	✘	✘	✔	✔	✔	✔
3	read(op);	✘	✘	✔	✔	✔	✔
4	if (op == "sum")	✘	✘	✔	✔	✔	✔
5	res = a - b; //fault	✘	✘	✔			
6	else if (op == "average")				✔	✔	✔
7	res = (a + b)/2;				✔	✔	✔
8	print(res);	✘	✘	✔	✔	✔	✔

Spectrum-Based Fault Localization (SBFL)

SBFL is a **heuristic**, **coverage-based**, **dynamic** method to identify faulty program elements (typically statements).

1. Execute unit under test (UUT) over test suite, collect ***coverage*** for each individual test case
2. Correlate coverage with outcomes, aggregate into ***spectrum***
3. Compute ***suspiciousness score*** for elements and ***rank***: higher scores and ranks indicate higher bug likelihood



#	program	t1	t2	t3	t4	t5	t6	sus	rank
1	read(a);	✘	✘	✔	✔	✔	✔	0.33	2
2	read(b);	✘	✘	✔	✔	✔	✔	0.33	2
3	read(op);	✘	✘	✔	✔	✔	✔	0.33	2
4	if (op == "sum")	✘	✘	✔	✔	✔	✔	0.33	2
5	res = a - b; //fault	✘	✘	✔				0.66	1
6	else if (op == "average")				✔	✔	✔	0	7
7	res = (a + b)/2;				✔	✔	✔	0	7
8	print(res);	✘	✘	✔	✔	✔	✔	0.33	2

How are scores computed?

1. Reduce spectra into four basic counts for each element *e*:

ep(e):	# ***passed***	tests in which **e** is	***executed***
ef(e):	# ***failed***	tests in which **e** is	***executed***
np(e):	# ***passed***	tests in which **e** is	***not executed***
nf(e):	# ***failed***	tests in which **e** is	***not executed***

2. Define ***ranking metric*** using basic counts

Tarantula $$\frac{\frac{ef(e)}{ef(e)+nf(e)}}{\frac{ef(e)}{ef(e)+nf(e)}+\frac{ep(e)}{ep(e)+np(e)}}$$

Jaccard $$\frac{ef(e)}{ef(e)+nf(e)+ep(e)}$$

Ochiai $$\frac{ef(e)}{\sqrt{(ef(e)+nf(e))\cdot(ef(e)+ep(e))}}$$

D* $$\frac{ef(e)^n}{nf(e)+ep(e)}$$

⇒ require at least one failing test

SBFL for Context-Free Grammars

Key insight: framework applies with minimal change

> "executed" grammar rules instead of program statements
> ⇒ ***grammar spectra***

Advantage #1: low-cost approach

- reuse existing framework and tooling

Advantage #2: domain-specific approach

- higher precision: ignores parser boilerplate code
- higher utility: localization at rule level

  ⇒ no tracking through code required

Grammar Spectra

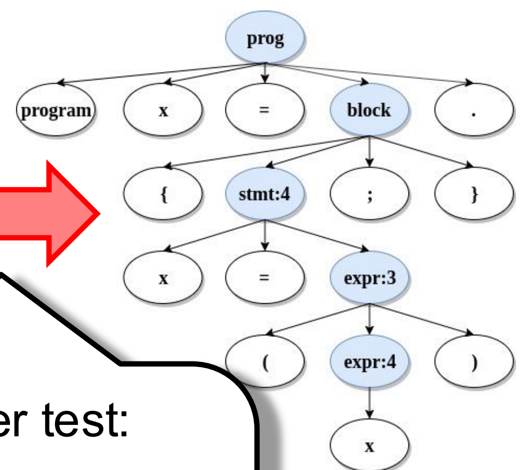
Given a ***grammar*** $G = (N,T,P,S)$ and a ***test suite*** $TS \subseteq T^*$, the ***grammar spectrum*** for TS comprises the

sets of rules $\boldsymbol{R \subseteq P}$ applied when each $\boldsymbol{w \in TS}$ is parsed.

```
...
program x={x=(x);}.
...
```

parse with grammar under test:

prog → **program** id **=** block **.**
block → **{** (decl **;**)* (stmt **;**)* **}**
decl → **var id :** type
type → **bool** | **int**
stmt → **sleep** | **if** expr **then** stmt **else** stmt
| **while** expr **do** block | **id =** expr | block
expr → expr **=** expr | expr **+** expr | **(**expr **)** | **id** | **num**



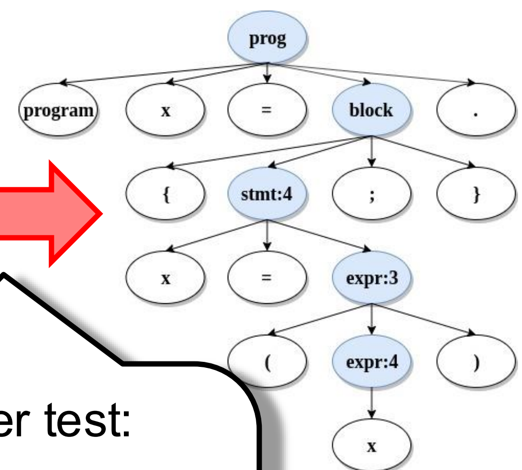
prog
block
stmt:4
expr:3
expr:4

Grammar Spectra

Given a ***grammar*** *G* = (*N*,*T*,*P*,*S*) and a ***test suite*** *TS* ⊆ *T**, the ***grammar spectrum*** for *TS* comprises the

> sets of rules ***R*** **⊆** ***P*** applied when each ***w*** **∈** ***TS*** is parsed.

```
...
program x={x=(x);}.
...
```



prog
block
stmt:4
expr:3
expr:4

parse with grammar under test:

prog → **program** id **=** block **.**
block → **{** (decl **;**)* (stmt **;**)* **}**
decl → **var id :** type
type → **bool** | **int**
stmt → **sleep** | **if** expr **then** stmt **else stmt**
  | **while** expr **do** block | **id =** expr | block
expr → expr **=** expr | expr **+** expr | **(**expr **)** | **id** | **num**

two faults

Grammar Spectra and Localization

```
program x={ x = (x); }.
program x={ x = x + x; }.
program x={ x = x; }.
program x={ x = x = x; }.
program x={ x = 0; }.
program x={ if x then sleep; }.
program x={ if x then sleep else sleep; }.
program x={ sleep; }.
program x={ var x : bool; }.
program x={ var x : int; }.
program x={ while x do sleep; }.
program x={ { }; }.
program x={ }.
```

rule	1	2	3	4	5	6	7	8	9	10	11	12	13
prog	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
block	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓
decl									✓	✓			
type:1													
type:2													
stmt:1						✗	✓	✓					
stmt:2						✗	✓						
stmt:3											✗		
stmt:4	✓	✓	✓	✓	✓								
stmt:5												✓	
expr:1													
expr:2		✓											
expr:3	✓												
expr:4	✓	✓	✓	✓		✗	✓				✗		
expr:5					✓								

if expr **then** stmt **else stmt**

while expr **do** block

id

Grammar Spectra and Localization

```
program x={ x = (x); }.
program x={ x = x + x; }.
program x={ x = x; }.
program x={ x = x = x; }.
program x={ x = 0; }.
program x={ if x then sleep; }.
program x={ if x then sleep else sleep; }.
program x={ sleep; }.
program x={ var x : bool; }.
program x={ var x : int; }.
program x={ while x do sleep; }.
program x={ { }; }.
program x={ }.
```

rule	1	2	3	4	5	6	7	8	9	10	11	12	13	*ep*	*np*	*ef*	*nf*
prog	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	11	0	2	0
block	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	11	0	2	0
decl									✓	✓				2	9	0	2
type:1														1	10	0	2
type:2														1	10	0	2
stmt:1						✗	✓	✓						2	9	1	1
stmt:2						✗	✓							**1**	**10**	**1**	**1**
stmt:3											✗			**0**	**11**	**1**	**1**
stmt:4	✓	✓	✓	✓	✓									5	6	0	2
stmt:5												✓		1	10	0	2
expr:1														1	10	0	2
expr:2		✓												1	10	0	2
expr:3	✓													1	10	0	2
expr:4	✓	✓	✓	✓		✗	✓				✗			5	6	2	0
expr:5					✓									1	10	0	2

if expr **then** stmt **else stmt**

while expr **do** block

id

Grammar Spectra and Localization

```
program x={ x = (x); }.
program x={ x = x + x; }.
program x={ x = x; }.
program x={ x = x = x; }.
program x={ x = 0; }.
program x={ if x then sleep; }.
program x={ if x then sleep else sleep; }.
program x={ sleep; }.
program x={ var x : bool; }.
program x={ var x : int; }.
program x={ while x do sleep; }.
program x={ { }; }.
program x={ }.
```

rule	1	2	3	4	5	6	7	8	9	10	11	12	13	*ep*	*np*	*ef*	*nf*	Tarantula		Ochiai		Jaccard		DStar	
prog	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	11	0	2	0	0.50	=5	0.39	=4	0.15	=5	0.36	=5
block	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	11	0	2	0	0.50	=5	0.39	=4	0.15	=5	0.36	=5
decl									✓	✓				2	9	0	2	0.00	-	0.00	-	0.00	-	0.00	-
type:1														1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
type:2														1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
stmt:1						✗	✓	✓						2	9	1	1	0.69	4	0.17	6	0.25	4	0.67	4
stmt:2						✗	✓							**1**	**10**	**1**	**1**	**0.85**	**2**	**0.50**	**3**	**0.33**	**2**	**2.00**	**2**
stmt:3											✗			**0**	**11**	**1**	**1**	**1.00**	**1**	**0.71**	**1**	**0.50**	**1**	**4.00**	**1**
stmt:4	✓	✓	✓	✓	✓									5	6	0	2	0.00	-	0.00	-	0.00	-	0.00	-
stmt:5												✓		1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:1														1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:2		✓												1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:3	✓													1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:4	✓	✓	✓	✓		✗	✓				✗			5	6	2	0	0.79	3	0.53	2	0.29	3	0.80	3
expr:5					✓									1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-

if expr **then** stmt **else stmt**

while expr **do** block

id

Implementation

Recursive-descent parsing: ANTLR

Each **rule** is implemented by its own **parsing function**

⇒ rule execution == call to a function

⇒ no difference between positive and negative tests (syntax error triggered only after function call)

How do we track parse functions?

1. use ANTLR's tree walkers
   – extend generated listeners (but requires error correction to build trees)

```
void enterEveryRule(ParserRuleContext ctx) {
  int index = ctx.getRuleIndex());
  int alt = ctx.getOuterAlt();
  print(parser.getRuleName(index)+":"+alt);
}
```

2. track internal calls to `enterOurAlt`()
   – use aspect-oriented programming (without error correction)

```
pointcut enterRuleAlt(ParserRuleContext ctx, int altNum, Parser parser) :
   call(void Parser.enterOuterAlt(ParserRuleContext, int))
   && args(ctx, altNum)
   && target(parser);
```

Table-driven LR parsing: CUP

Positive spectrum: $w \in L(G)$

$\Rightarrow$ rule execution == reduction

Negative spectrum: $w \notin L(G)$

- capture reductions for fully executed rules
- capture partial reductions on the parse stack at syntax error

```
program x = {
  while x do sleep;
}.
```

parse with grammar under test:

prog	→ **program** id **=** block **.**
block	→ **{** (decl **;**)* (stmt **;**)* **}**
decl	→ **var id :** type
type	→ **bool** | **int**
stmt	→ **sleep** | **if** expr **then** stmt **else** stmt
	| **while** expr **do** block | **id =** expr | block
expr	→ expr **=** expr | expr **+** expr | **(**expr **)** |
	id | **num**

state	corresponding kernel items
2	*prog* → **program** • id = *block* .
3	*prog* → **program** id • = *block* .
4	*prog* → **program** id = • *block* .
5	*block* → **{** • ((*decl* ;)* (*stmt* ;)*) **}**
8	*stmt* → **while** • *expr* **do** *block*
50	*stmt* → **while** *expr* • **do** *block* *expr* → *expr* • = *expr* *expr* → *expr* • + *expr*
51	*stmt* → **while** *expr* **do** • *stmt*

prog
block
stmt:3
expr:1
expr:2
expr:4

Evaluation

Evaluation #1: Fault seeding

Goals:

- evaluate effectiveness
- evaluate effects of different test suites
- evaluate effects of error correction
- compare performance for LL and LR parsers

Subject:

SIMPL grammar from 2nd-year computer architecture course

	$\lvert N \rvert$	$\lvert T \rvert$	$\lvert P \rvert$
ANTLR	42	47	84
CUP	32	47	80

Fault seeding:

- full mutation of all rules in “golden” grammar
  (symbol deletion, insertion, substitution and transposition)
- keep only compiling grammars
  (ANTLR: 32274, CUP: 26628)

Evaluation #1: Fault seeding

Test suites:

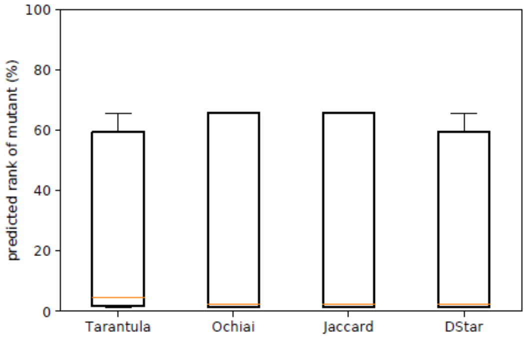
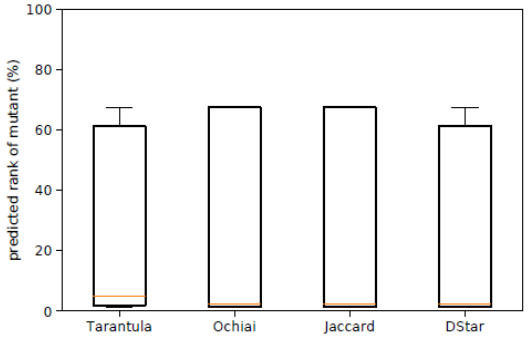
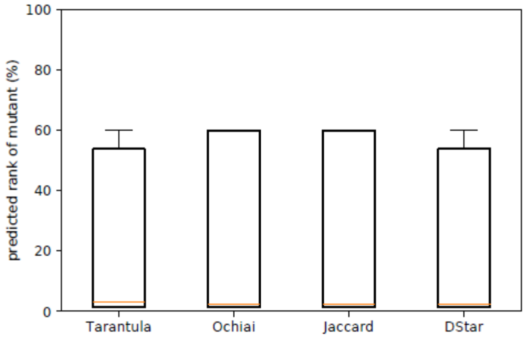
- generated from golden grammar
  - positive only: rule, cdrc
  - mixed: see “Breaking Parsers” for negative test suites
- instructor’s marking test suite

⇒ not all mutants “killed” by test suites
(i.e., localization impossible)

Measurement:

average predicted rank of mutated rule

Evaluation #1: Fault seeding - Results

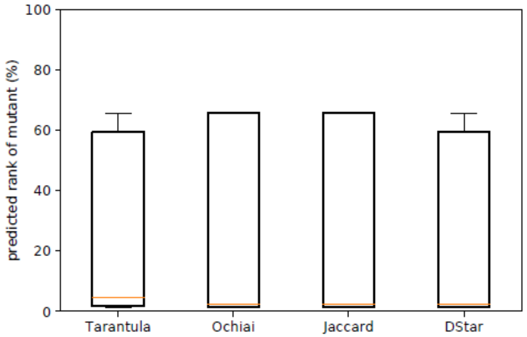
ANTLR (no error correction)	**ANTLR*** (default error correction)	**CUP**
		

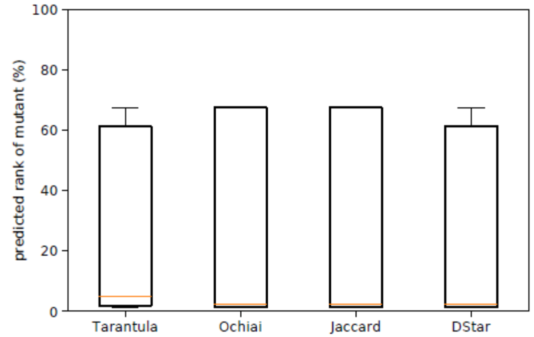
rule: 43 positive test cases

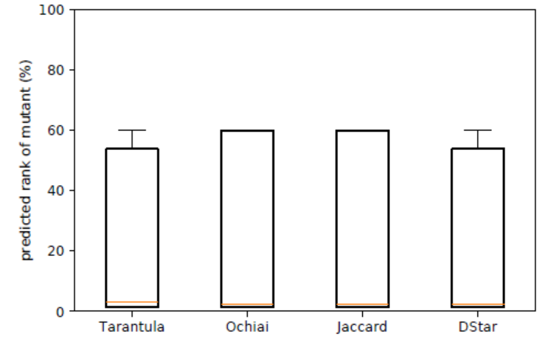
- ANTLR: ~80% killed, median rank ~5% (4 rules), big variance
  ~25% rank #1
- ANTLR*: slightly worse (especially rank #1)
- CUP: ~90% killed, median rank ~3% (3 rules), big variance
  ~40% rank #1
- Tarantula performs slightly worse, DStar slightly better

Evaluation #1: Fault seeding - Results

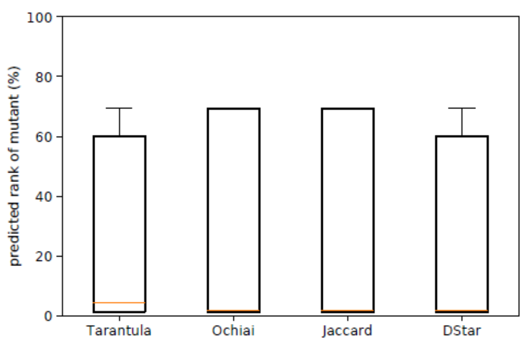
ANTLR (no error correction) | **ANTLR*** (default error correction) | **CUP**

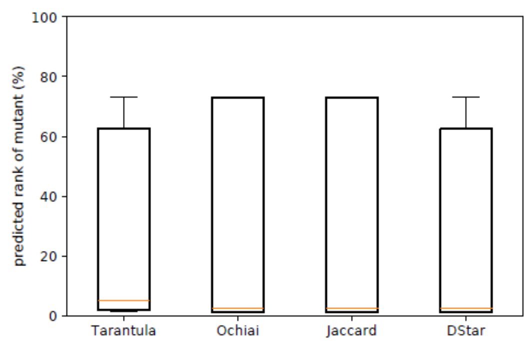


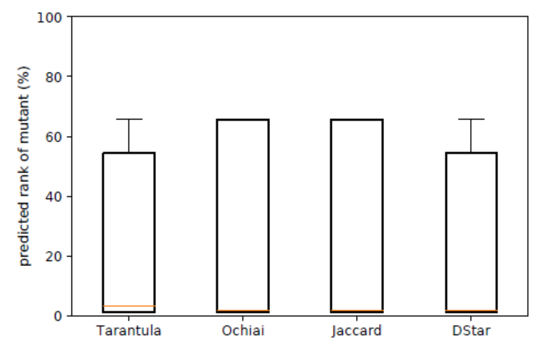




rule: 43 positive test cases





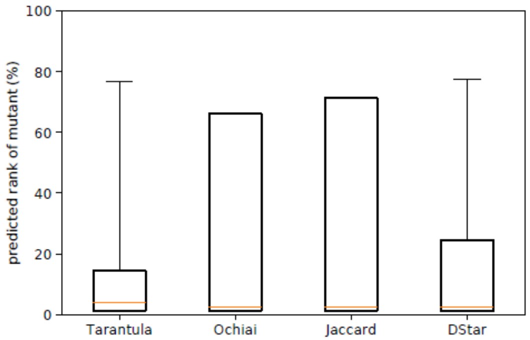


cdrc: 86 positive test cases

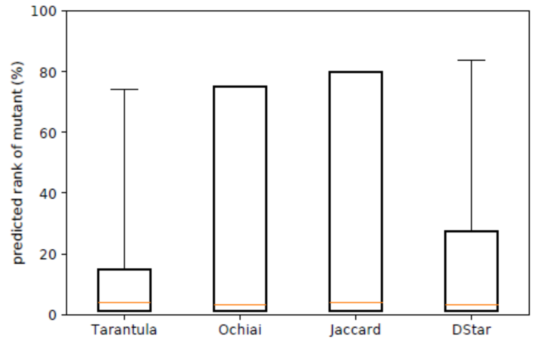
- minor improvements (especially rank #1)

Evaluation #1: Fault seeding - Results

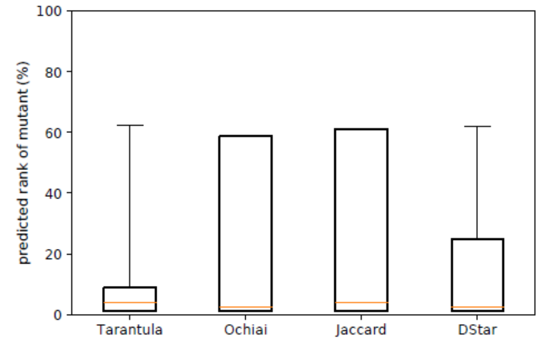
ANTLR
(no error correction)



ANTLR*
(default error correction)



CUP

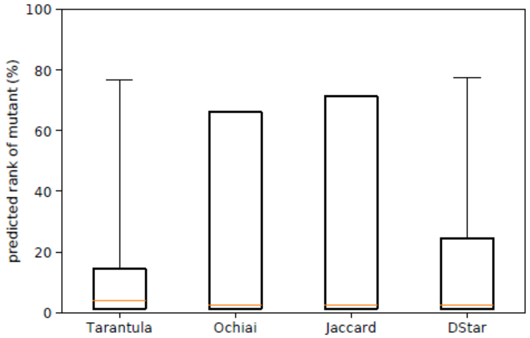


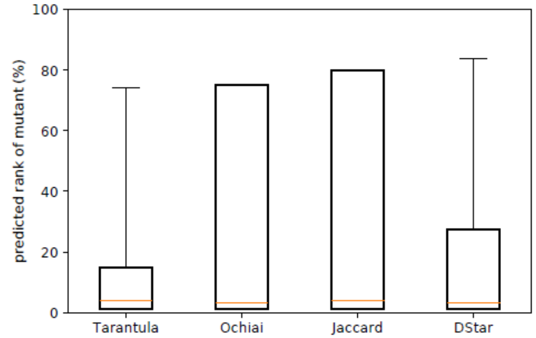
large: 2964 positive / 32157 negative test cases

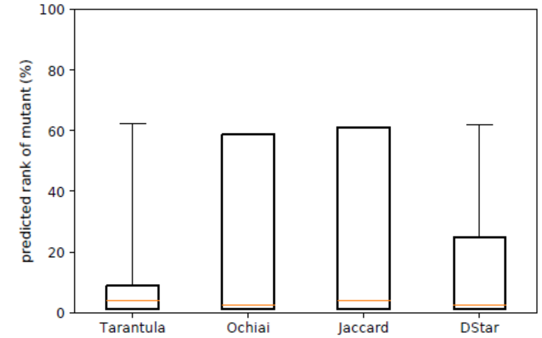
- increased kill (prediction) rates (ANTLR ~90%, CUP ~99%)
- ANTLR: large increases in rank #1 predications (~30%)
  CUP: minor decreases (loosing precision)
- Tarantula and DStar: reduced variance
  (but plots are deceiving)
- Tarantula overall best results for ANTLR,
  DStar overall best results for CUP

Evaluation #1: Fault seeding - Results

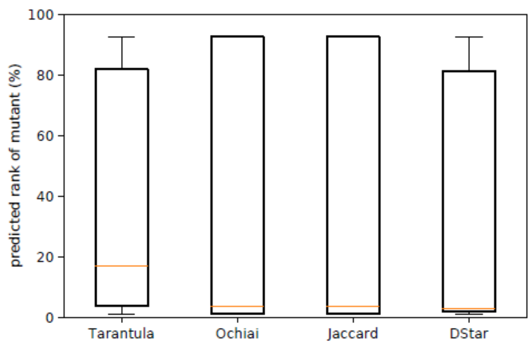
ANTLR (no error correction)	**ANTLR*** (default error correction)	**CUP**

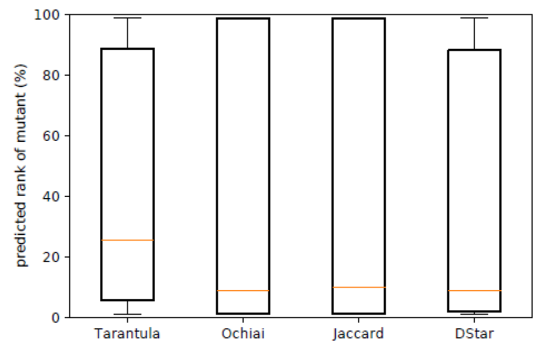


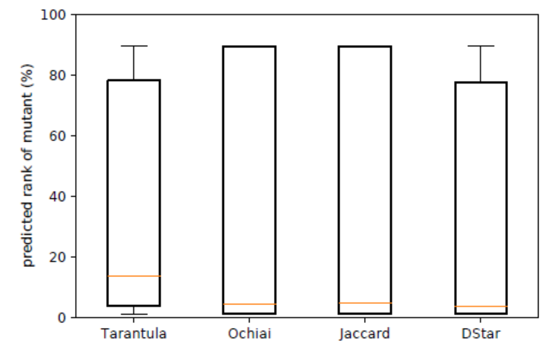




large: 2964 positive / 32157 negative test cases







instructor: 20 positive / 61 negative test cases

- results deteriorate: not enough syntactic variance

Evaluation #2: Debugging student grammars

Approach: iterative fault localization with manual repair

- Ochiai used to compute scores from *large* test suite
- manual inspection of rules in rank order
- manual repair within Top-5 rules
- repeated until no further failing test cases

Subjects: SIMPL and Blaise (similar complexity)

- student grammars from SLE course (ANTLR and CUP)
- not all submissions contained errors

Evaluation #2: Debugging student grammars

Approach: iterative fault localization with manual repair

- Ochiai used to compute scores from *large* test suite
- manual inspection of rules in rank order
- manual repair within Top-5 rules
- repeated until no further failing test cases

			iteration 1		iteration 2		iteration 3		iteration 4		iteration 5		iteration 6	
#	language	type	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank
1	SIMPL	CUP	557	1	254	1	131	1	98	1				
2	SIMPL	CUP	206	2	95	2								
3	SIMPL	CUP	498	1	40	1								
4	SIMPL	CUP	169	1	46	1								
5	SIMPL	CUP	853	1	378	1	219	1	130	1	37	1	6	1
6	SIMPL	CUP	244	1	121	9	80	✗						
7	Blaise	ANTLR	567	2	4	1	2	1						
8	Blaise	ANTLR	1082	1	535	3	7213	1	358	1	43	1	2	1
9	Blaise	ANTLR	4	3	2	2								
10	Blaise	ANTLR	1068	1	4	2	2	1						
11	Blaise	ANTLR	38	4	3	1								
12	Blaise	ANTLR	654	1	1	1								
13	Blaise	ANTLR	4	2	2	1								
14	SIMPL	ANTLR	555	1	170	1	47	2	1	1				
15	SIMPL	ANTLR	37	5	1	1								
16	SIMPL	ANTLR	361	3	46	1								
17	SIMPL	ANTLR	396	1	117	2	81	2	47	1	1	1		
18	SIMPL	ANTLR	46	2										
19	SIMPL	ANTLR	356	1	233	2	1	1						

Conclusions and Future work

Conclusion: SBFL can find bugs in grammars.

- ranks seeded faults on average within 15%-25% of rules
- pinpoints (i.e., uniquely identifies) 10%-40% of seeded faults
- can handle real and multiple faults
- results vary with metric, test suite, and parsing technology

Future work

- extend experimental evaluation:
  - more parsers and languages
  - more detailed statistical analysis
- modify ranking metrics and tie-breaking mechanisms to exploit grammar structure
- automatic grammar repair