User Interface Grammars

MOBILESoft 2020 Keynote

Andreas Zeller
with Nataniel P. Borges Jr., Manuel Benz, and Eric Bodden

CISPA Helmholtz Center for Information Security

User Interface Grammars

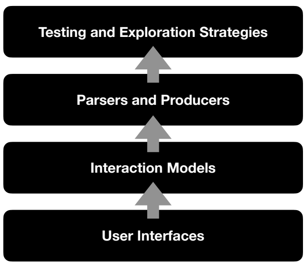
Andreas Zeller, CISPA Helmholtz Center for Information Security
Joint work with Nataniel P. Borges Jr., Manuel Benz, and Eric Bodden

Abstract. To systematically explore user interfaces, one must cover graphical interaction features (e.g. clicks, swipes) as well as textual interaction features (e.g. form input). We introduce user interface grammars as a single formalism that captures and integrates graphical and textual input languages. A UI grammar encodes graphical interactions and text input as a single (possibly nontrivial) stream of input events, allowing for their uniform treatment in test generation and/or coverage measurement. Grammars can be mined from existing systems (GUI-based or text-based), allow for simple customization by testers (say, for special inputs such as passwords or injection attacks) as well as guidance towards UI (model) coverage and code coverage. Includes live demos!

https://andreas-zeller.info/

A Call for Modularity

- Every testing tool reinvents **its own model and analysis**
- Makes it hard to **assess, compare, and reuse approaches**
- Call for solid **formal foundations** and modular algorithms
- Draw on established fields such as **formal languages**



Generating Inputs

"If you talk to a man in his language, that goes to his heart." – Nelson Mandela

Fuzzing

```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4)))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```

Interpreter

Fuzzing means to throw random inputs at a program to see if it crashes.

Dumb Fuzzing

```
(144 60 )5(5-(05*/(  * *)910)25/509505)3)/
09211762 /(7*+22)76-+/29+/4**2+

8(  )04/844)

4)632/3/7 *0525+)7*
```

Interpreter

?

But if you just take sequences of random characters and throw them at an interpreter, all you're going to get is syntax errors. (It's okay to test syntax error handling, but this should not be all.)

Grammars

Specify a language (= a set of inputs)

Expansion rule

Nonterminal symbol

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Terminal symbol

In order to get syntactically valid inputs, you need a specification. A **grammar** specifies the set of inputs as a **language**.

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

You may have seen grammars as **parsers**, but they can also be used as **producers** of inputs.

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨start⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

You start with a start symbol

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨start⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

which then subsequently gets replaced according to the production rules in the grammar.

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨term⟩ - ⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

If there are multiple alternatives, you randomly choose one.

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨term⟩ - ⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨factor⟩ - ⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨int⟩ . ⟨int⟩ - ⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

⟨digit⟩ . ⟨int⟩ - ⟨expr⟩

Nikolas Havrikov and Andreas Zeller. **Systematically Covering Input Structure.** ASE 2019.

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
⟨digit⟩ . ⟨digit⟩ - ⟨expr⟩
```

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
8. ⟨digit⟩ - ⟨expr⟩
```

Grammars as Producers

```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
8.2 - ⟨expr⟩
```

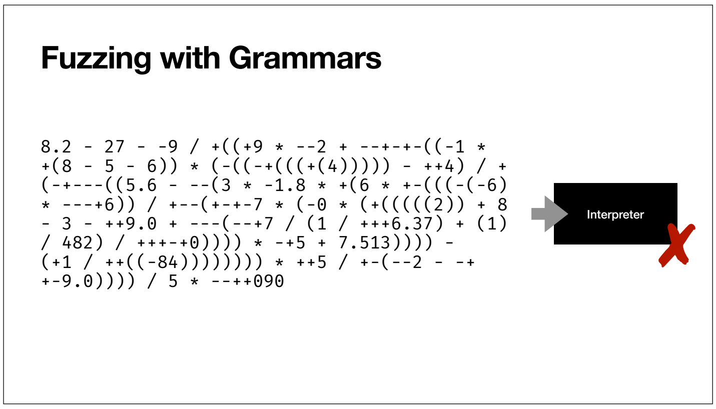

Over time, this gives you a syntactically valid input. In our case, a valid arithmetic expression.

Grammars as Producers

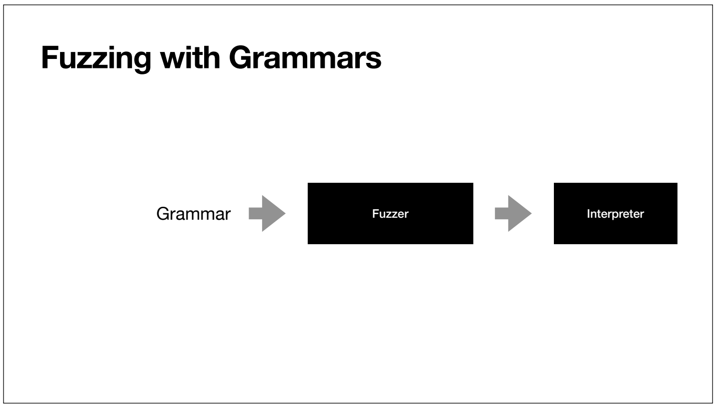
```
⟨start⟩  ::= ⟨expr⟩
⟨expr⟩   ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩   ::= ⟨term⟩ * ⟨factor⟩ | ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩    ::= ⟨digit⟩ ⟨int⟩ | ⟨digit⟩
⟨digit⟩  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

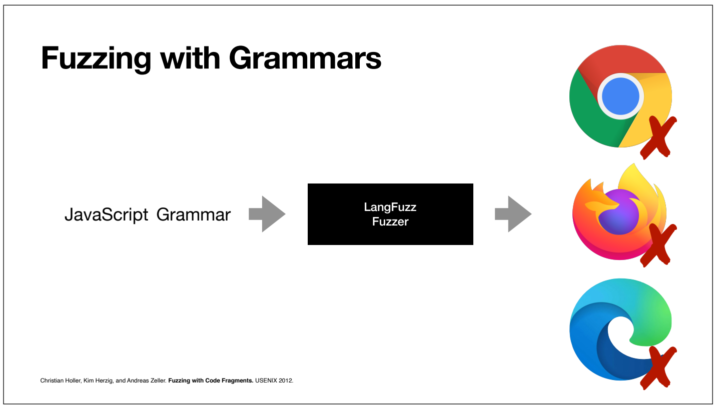
```
8.2 - 27 - -9 / +((+9 * --2 + --+-+-((-1 *
+(8 - 5 - 6)) * (-((-+(((+(4)))))) - ++4) / +
(-+---((5.6 - --(3 * -1.8 * +(6 * +-(((-(-6)
* ---+6)) / +--(+-+-7 * (-0 * (+(((((2)) + 8
- 3 - ++9.0 + ---(--+7 / (1 / +++6.37) + (1)
/ 482) / +++-+0)))) * -+5 + 7.513)))) -
(+1 / ++((-84))))))))) * ++5 / +-(--2 - -+
+-9.0)))) / 5 * --++090
```


Actually, a pretty **complex** arithmetic expression.

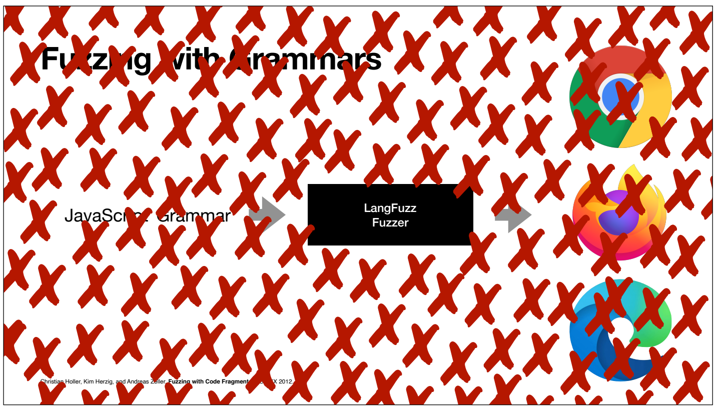


These can now be used as input to your program.





A couple of years ago, we used a JavaScript grammar to fuzz the interpreters of Firefox, Chrome and Edge.



My student Christian Holler found more than 2,600 bugs, and in the first four weeks, he netted more than $50,000 in bug bounties. If you use a browser to read this, one of the reasons your browser works as it should is because of grammar-based fuzzing.

Mining Grammars

"The more languages you know, the more you are human." – Tomáš Garrigue Masaryk

---

Fuzzing with Grammars

So where did you get this grammar from?

Mining Grammars

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
void parse_expr() {
    parse_term();
    if (lookahead() == '+') { consume(); parse_expr(); }
    if (lookahead() == '-') { consume(); parse_expr(); }
}
void parse_term() { ... }
void parse_factor() { ... }
void parse_int() { ... }
void parse_digit() { ... }
```

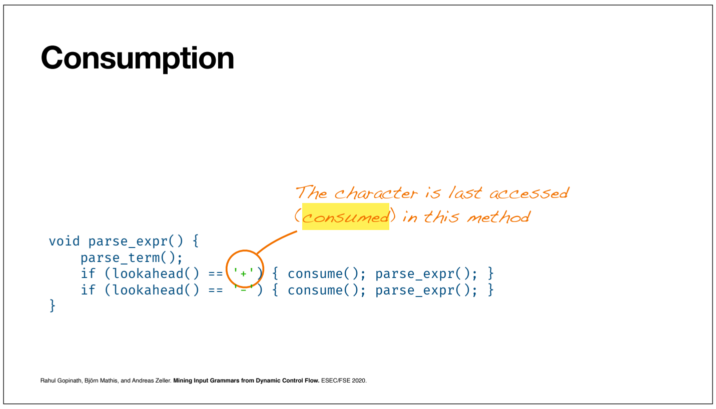
So let me tell you a bit about how to mind such grammars. The idea is to take a program that parses such inputs and extract the input grammar from it.

Rules and Locations

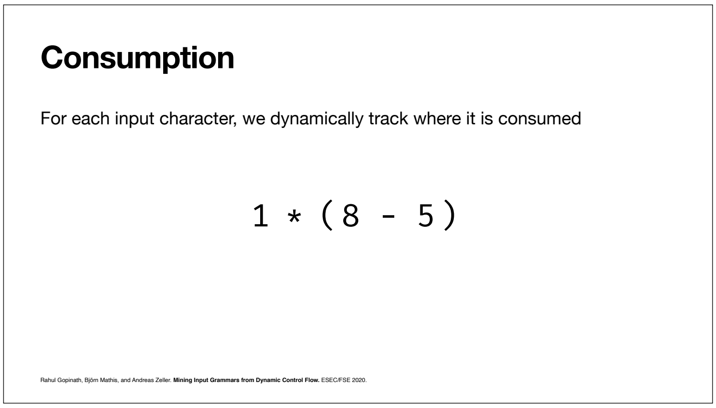
```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩  +  ⟨expr⟩ | ⟨term⟩  -  ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩  *  ⟨factor⟩ | ⟨term⟩  /  ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= + ⟨factor⟩ | - ⟨factor⟩ | ( ⟨expr⟩ ) | ⟨int⟩ | ⟨int⟩ . ⟨int⟩
⟨int⟩     ::= ⟨digit⟩  ⟨int⟩ | ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

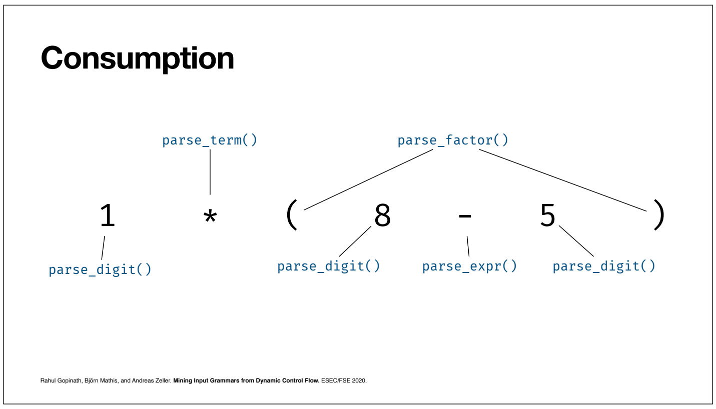
```
void parse_expr() {
    parse_term();
    if (lookahead() == '+') { consume(); parse_expr(); }
    if (lookahead() == '-') { consume(); parse_expr(); }
}
```

The interesting thing is that there is a correspondence between individual rules in the input grammar and locations in the parsing code.

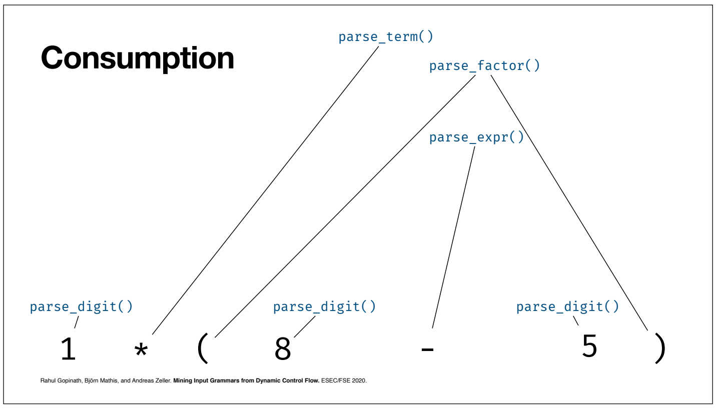


The concept of consumption establishes this correspondence. A character is **consumed** in a method *m* if *m* is the last to access it.

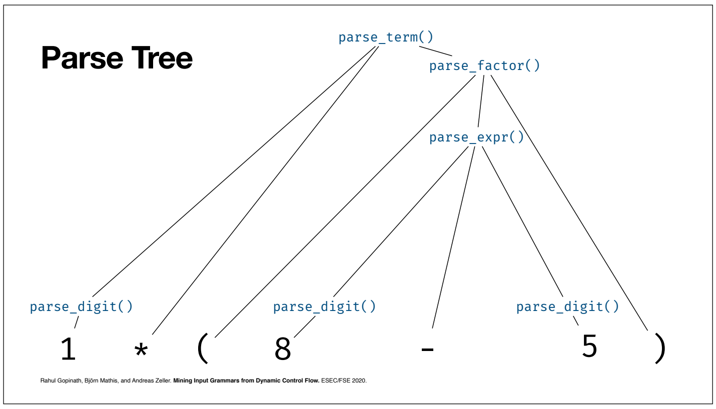




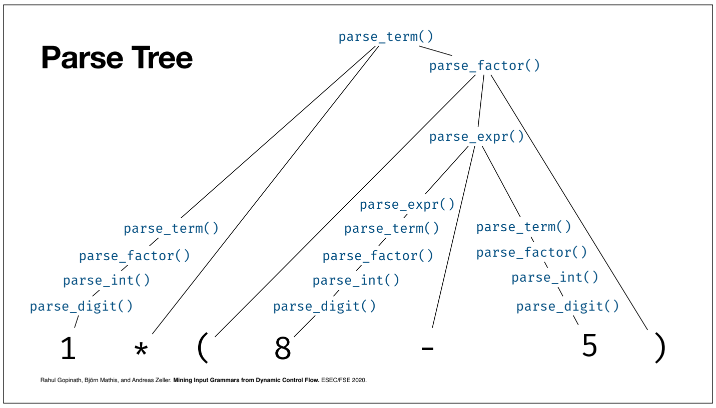
During program execution we can track where characters are consumed using dynamic tainting.



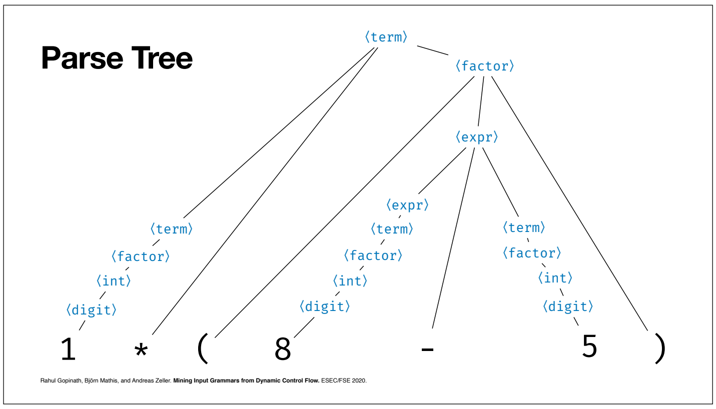
This gives us a tree like structure.



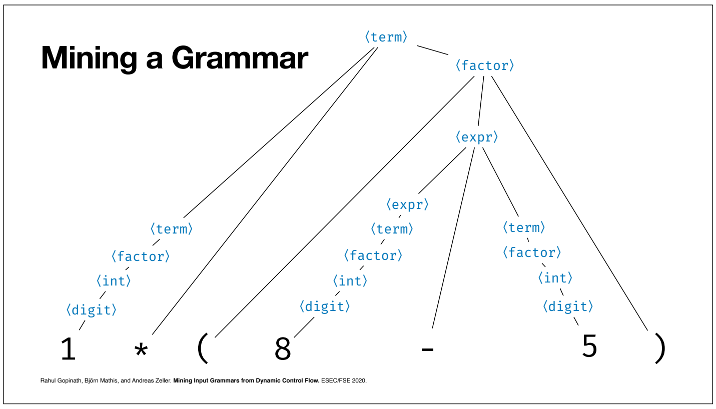
Which we can augment with caller-callee relations.



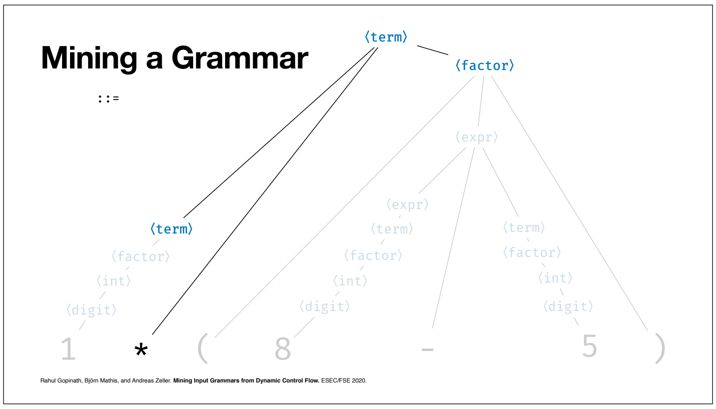
Even for those functions which do not consume anything.



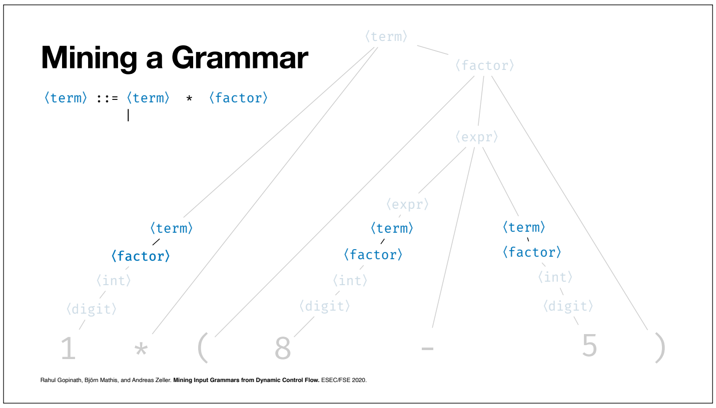
If we take the function names and only use the nouns, we can use those nouns as non-terminal symbols.



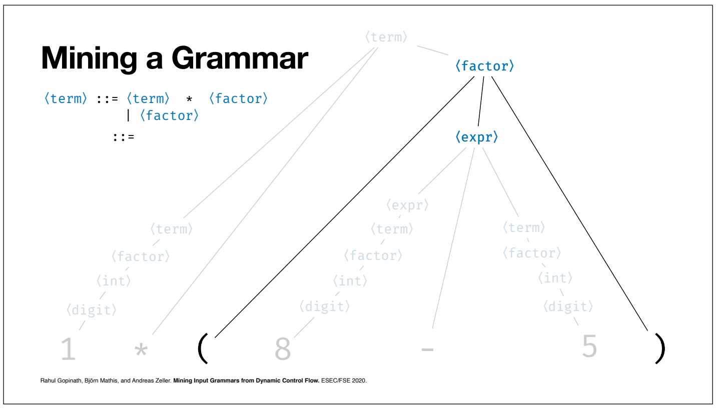
From these parse trees, we can now mine a grammar.



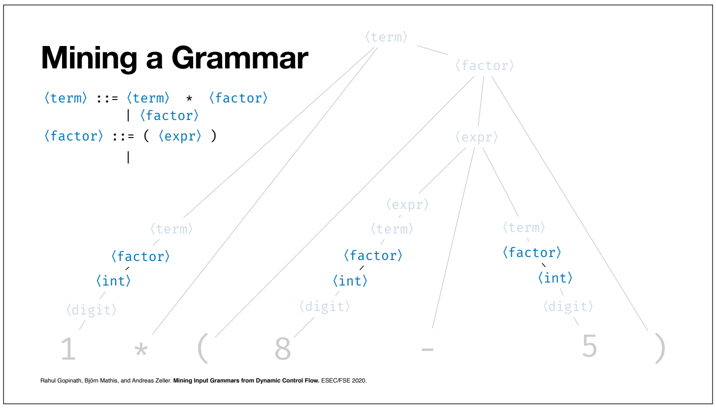
A term obviously can consist of another term, a multiplication symbol, and a factor.



So we add this as a rule to our grammar.

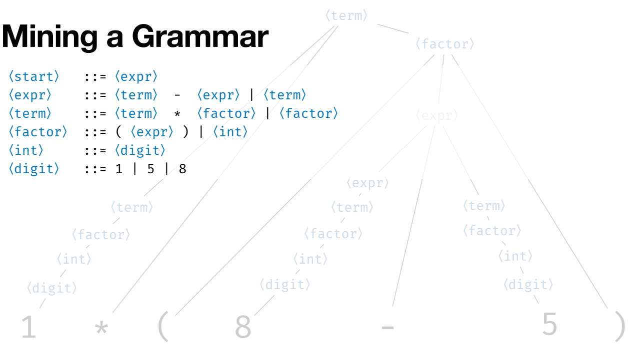


And likewise for other symbols.



Mining a Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::= 1 | 5 | 8
```



Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

From this single input, we already get the basics of a grammar.

Completing the Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::= 1 | 5 | 8
```

Parse tree

`0 + 2`

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

And if we add more inputs, ...

Completing the Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 5 | 8
```

Parse tree

`0 + 2`

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.

... the grammar reflects the structure of these additional inputs.

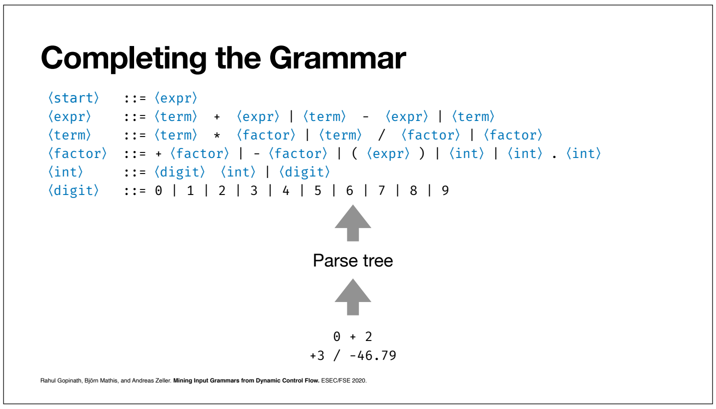
Completing the Grammar

```
⟨start⟩   ::= ⟨expr⟩
⟨expr⟩    ::= ⟨term⟩ + ⟨expr⟩ | ⟨term⟩ - ⟨expr⟩ | ⟨term⟩
⟨term⟩    ::= ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩  ::= ( ⟨expr⟩ ) | ⟨int⟩
⟨int⟩     ::= ⟨digit⟩
⟨digit⟩   ::= 0 | 1 | 2 | 5 | 8
```

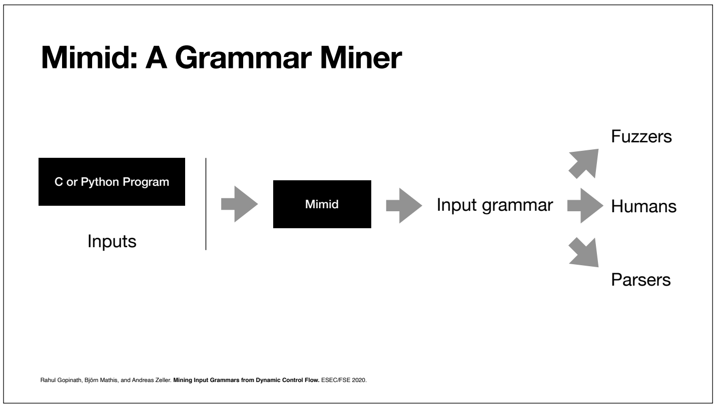
Parse tree

```
0 + 2
+3 / -46.79
```

Rahul Gopinath, Björn Mathis, and Andreas Zeller. **Mining Input Grammars from Dynamic Control Flow.** ESEC/FSE 2020.



We now have successfully mined our example grammar.



Our Mimid grammar miner takes a program and its inputs and extracts a grammar out of it. This grammar can directly be used by fuzzers, parsers, and humans.

The extracted grammars are well structured and human readable as you can see in this grammar extracted from a JSON parser.

Humans can **edit** these grammars.

Fuzzer

```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ | 10% [ ⟨json_list′⟩ | 50% { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | ⟩ | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩ (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* "
⟨json_number′⟩ ::= ⟨json_number⟩+ | ⟨json_number⟩+ e ⟨json_number⟩+
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

Humans

For instance, by assigning probabilities to individual productions.

Fuzzer

```
⟨start⟩ ::= ⟨json_raw⟩
⟨json_raw⟩ ::= " ⟨json_string′⟩ | [ ⟨json_list′⟩ | { ⟨json_dict′⟩
 | ⟨json_number′⟩ | true | false | null
⟨json_string⟩ ::= ⟨space⟩ | ! | # | $ | % | & | '
 | * | + | - | , | . | / | : | ;
 | < | = | ⟩ | ? | @ | [ | ] | ^ | _ | , | ‘ |
 | { | | | } | ~ | /[A-Za-z0-9]/ | \ ⟨decode_escape⟩
⟨decode_escape⟩ ::= " | / | b | f | n | r | t
⟨json_list′⟩ ::= ]
 | ⟨json_raw⟩ (, ⟨json_raw⟩ )* ]
 | (, ⟨json_raw⟩ )+ (, ⟨json_raw⟩ )* ]
⟨json_dict′⟩ ::= }
 | ( " ⟨json_string′⟩ : ⟨json_raw⟩ , )*
 | " ⟨json_string′⟩ : ⟨json_raw⟩ }
⟨json_string′⟩ ::= ⟨json_string⟩* " | '; DROP TABLE students"
⟨json_number′⟩ ::= ⟨json_number⟩+ | ⟨json_number⟩+ e ⟨json_number⟩+
⟨json_number⟩ ::= + | - | . | /[0-9]/ | E | e
```

Humans

Or by inserting magic strings that program analysis would have a hard time finding out.

Fuzzer

```
{ "": "'; DROP TABLE STUDENTS" , "/h?O ": [ ] , "": "" , "x": false ,
"": null }
{ "": ".qF" , "": "'; DROP TABLE STUDENTS", "": 47 }
{ "7": { "y": "" }, "": false, "X": "N7|:", "": [ true ], "": [ ], "": {
} }
{ "": [ ], "9z6}l": null }
{ "#": false, "D": { "": true }, "t": 90, "g": [ "'; DROP TABLE
STUDENTS" ], "": [ false ], "=R5": [ ], " ": "'; DROP TABLE STUDENTS",
"`l": { "": "?'L", "E": null, "": [ 70.3076998940e6 ], "Ju": true } }
{ "": true, "": "%7y", "!": false, "": true, "": { "": [ ], "":
-096860E+0, "U": 0E-5 } }
{ "'ia": [ true, "'; DROP TABLE STUDENTS", null, [ false, { } ],
true ] }
{ "@meB1T]": 0.0, "": null, "": true, "7": 208.00E4, "": true, "":
70e+10, "": "", "5zJ": [ false, false ] }
{ "": "H", "d;": "'; DROP TABLE STUDENTS" }
{ "Y!Z": ".i", "h": "'; DROP TABLE STUDENTS" }
{ "": -64.0e-06, "": [ { "p[f": false, "": "'; DROP TABLE STUDENTS",
"m": [ ], "": true, "8D": -0, "@R": true } ] }
{ "": "'; DROP TABLE STUDENTS" }
{ "r": "'; DROP TABLE STUDENTS", "zJzjT": 6.59 }
{ "oh": false }
{ "c": [ false, 304e+008520, null, false, "'; DROP TABLE STUDENTS",
"m[MD" , [ false ] ] }
```

Performance by highly trained professional. **Do not try this** at home, your university or anywhere else.

This change to the grammar injects SQL statements everywhere. Do not do this at home, folks – thank you.

Mining Grammars

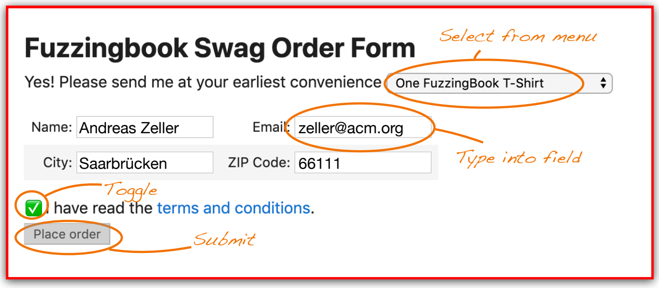
"The more languages you know, the more you are human." – Tomáš Garrigue Masaryk

Filling Forms

"Estimated time for filling out this form is three hours, 12 minutes" – US tax form

Now, let's move over to user interfaces.

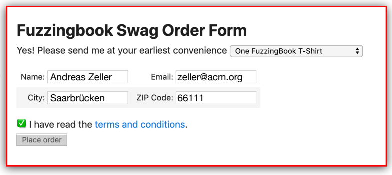
A Form



Here is a graphical user interface with various UI elements.

Interactions

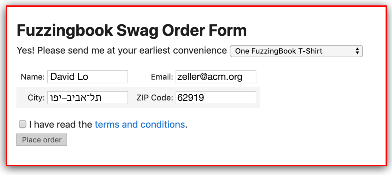
```
⟨Interaction⟩ ::=
   fill("Name", "Andreas Zeller")
   fill("Email", "zeller@acm.org")
   fill("City", "Saarbrücken")
   fill("Zip", "66111")
   set("T+C", True)
   submit("Place order")
```



We can express the interaction with the GUI through a series of commands expressing the interactions.

Interactions as Grammars

```
⟨Interaction⟩ ::=
   fill("Name", ⟨Name⟩ )
   fill("Email", ⟨Email⟩ )
   fill("City", ⟨City⟩ )
   fill("Zip", ⟨Zip⟩ )
   set("T+C", ⟨Boolean⟩ )
   submit("Place order")
```

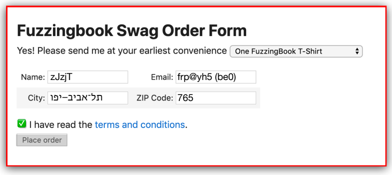



```
⟨Name⟩ ::= "Andreas Zeller" | "David Lo"
⟨Email⟩ ::= "zeller@acm.org"
⟨City⟩ ::= "Saarbrücken" | "תל־אביב–יפו"
⟨Zip⟩ ::= "66111" | "62919"
⟨Boolean⟩ ::= True | False
```

This neatly integrates with a concept of grammars as you can also express alternatives...

Interactions as Grammars

```
⟨Interaction⟩ ::=
   fill("Name", ⟨Name⟩ )
   fill("Email", ⟨Email⟩ )
   fill("City", ⟨City⟩ )
   fill("Zip", ⟨Zip⟩ )
   set("T+C", ⟨Boolean⟩ )
   submit("Place order")
```

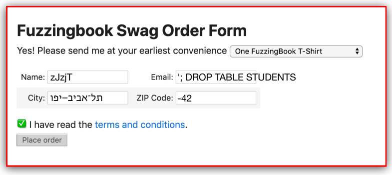



```
⟨Name⟩ ::= "Andreas Zeller" | "David Lo" | " ⟨first-name⟩ ⟨last-name⟩ " | " ⟨char⟩* "
⟨Email⟩ ::= "zeller@acm.org" | " ⟨e-mail⟩"
⟨City⟩ ::= "Saarbrücken" | "תל־אביב–יפו"
⟨Zip⟩ ::= "66111" | "12345" | " ⟨digit⟩* "
⟨Boolean⟩ ::= True | False
```

... as well as generic grammar rules for individual elements ...

Interactions as Grammars

```
⟨Interaction⟩ ::=
   fill("Name", ⟨Name⟩ )
   fill("Email", ⟨Email⟩ )
   fill("City", ⟨City⟩ )
   fill("Zip", ⟨Zip⟩ )
   set("T+C", ⟨Boolean⟩ )
   submit("Place order")
```




```
⟨Name⟩ ::= "Andreas Zeller" | "David Lo" | " ⟨first-name⟩ ⟨last-name⟩ " | " ⟨char⟩* "
⟨Email⟩ ::= "zeller@acm.org" | " ⟨e-mail⟩" | "'; DROP TABLE STUDENTS"
⟨City⟩ ::= "Saarbrücken" | "תל־אביב–יפו"
⟨Zip⟩ ::= "66111" | "12345" | " ⟨digit⟩* " | "- ⟨digit⟩* "
⟨Boolean⟩ ::= True | False
```

... and also invalid(!) values.

Interactions as Grammars

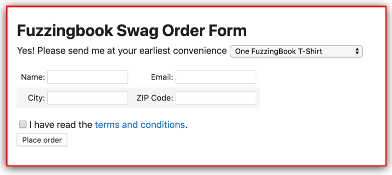
```
⟨Interaction⟩ ::=
   fill("Name", ⟨Name⟩ )
   fill("Email", ⟨Email⟩ )
   fill("City", ⟨City⟩ )
   fill("Zip", ⟨Zip⟩ )
   set("T+C", ⟨Boolean⟩ )
   submit("Place order")
```

- *Full control* over text entered
- All features of text-based fuzzing (= coverage guidance, weights, generators, …)
- *Combine* with inputs from other sources

```
⟨Name⟩ ::= "Andreas Zeller" | "David Lo" | " ⟨first-name⟩ ⟨last-name⟩ " | " ⟨char⟩* "
⟨Email⟩ ::= "zeller@acm.org" | " ⟨e-mail⟩" | "'; DROP TABLE STUDENTS"
⟨City⟩ ::= "Saarbrücken" | "תל־אביב–יפו"
⟨Zip⟩ ::= "66111" | "12345" | " ⟨digit⟩* " | "- ⟨digit⟩* "
⟨Boolean⟩ ::= True | False
```

Mining User Interface Grammars

```
⟨Interaction⟩ ::=
   fill("Name", ⟨Name⟩ )
   fill("Email", ⟨Email⟩ )
   fill("City", ⟨City⟩ )
   fill("Zip", ⟨Zip⟩ )
   set("T+C", ⟨Boolean⟩ )
   submit("Place order")
```



- For each UI element, determine the set of valid inputs
- The more accessible the form, the more precise the grammar

Such grammars can be easily mined – say, by analyzing the HTML tags.

Demo

Generating Inputs

"If you talk to a man in his language, that goes to his heart." – Nelson Mandela

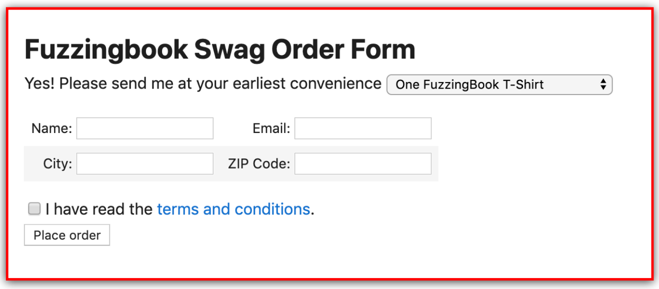
---

User Interfaces

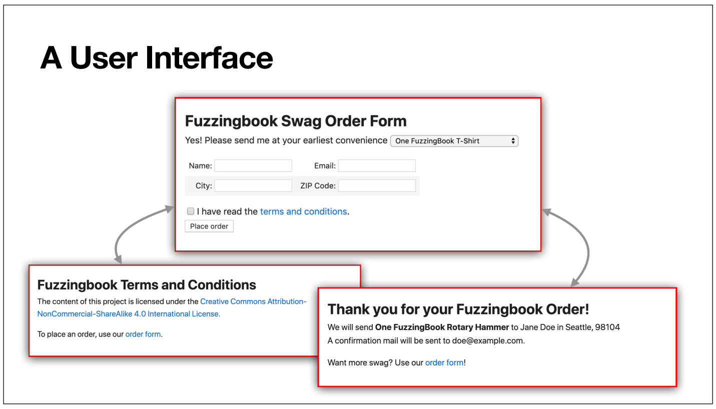
"No matter how cool your interface is, it would be better if there were less of it." — Alan Cooper

Now from forms to sequences of windows.

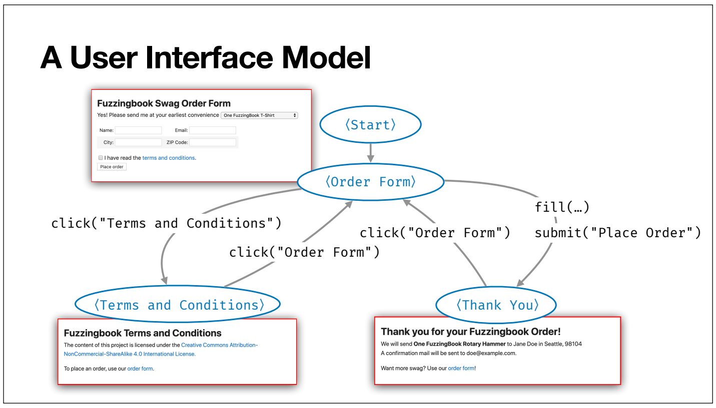
A Form



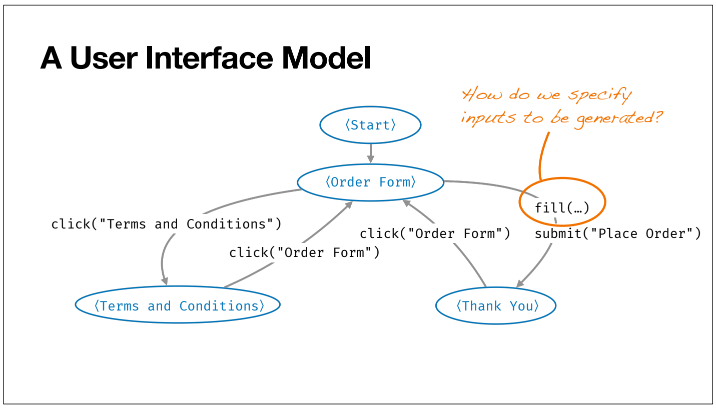
Here again, we see a form. But this is just part of a larger set of screens that are all interconnected.



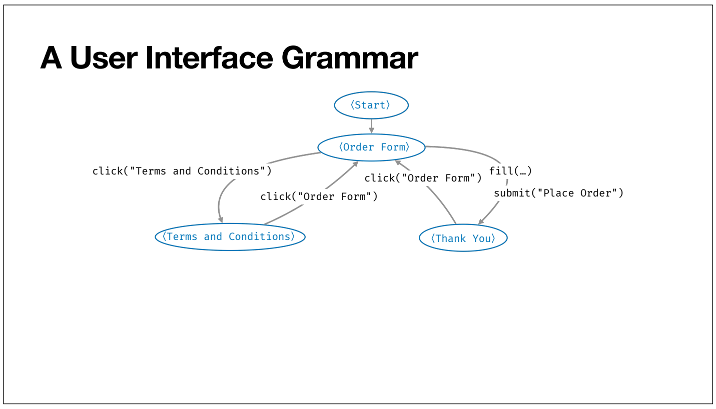
If for instance, you click on terms and conditions, you get a window that explains these terms and conditions. If you place your order, you get a confirmation window.



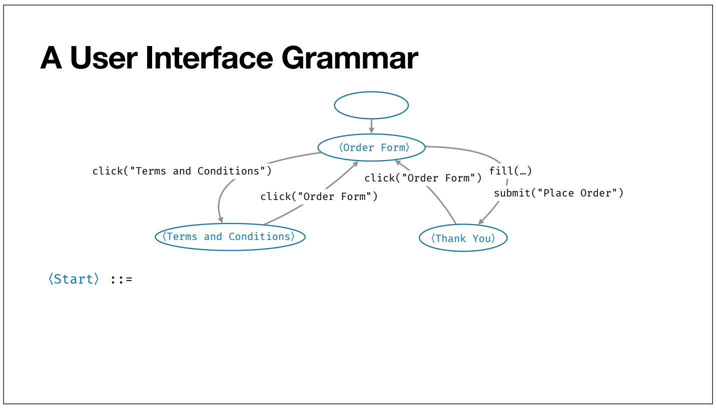
Classically, these different windows are represented as states, and interactions to change between windows become transitions.



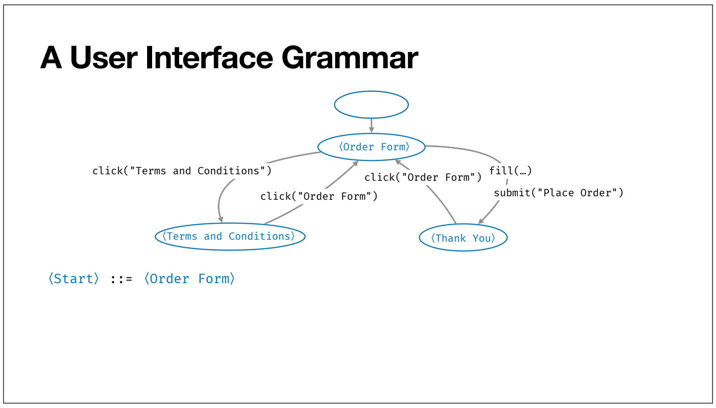
The resulting finite state model, however, does not state the rules for textual input.

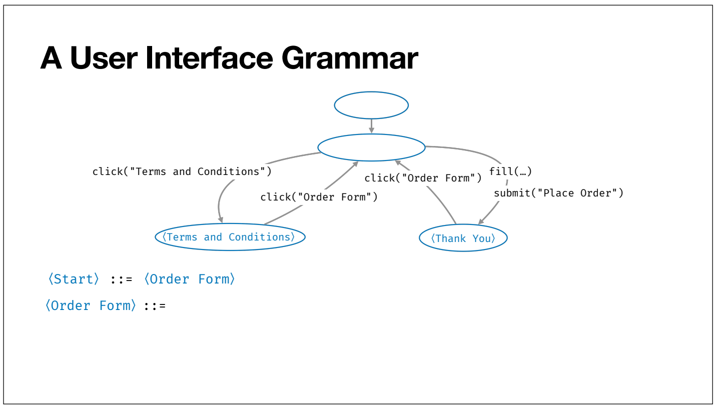


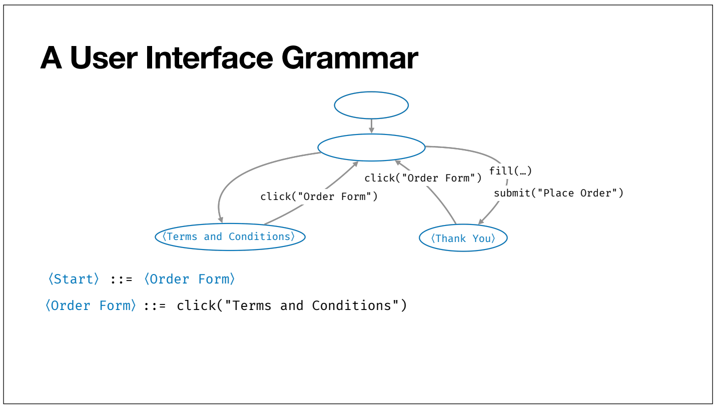
However, we can integrate both by embedding the finite state model into a grammar.



Every state in the final state model then becomes a non-terminal in the grammar

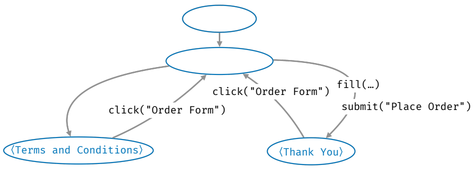






And each transition becomes an expansion of the source state (nonterminal), listing of the interactions and ending in the target state.

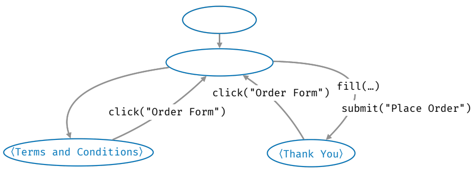
A User Interface Grammar



⟨Start⟩ ::= ⟨Order Form⟩

⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩

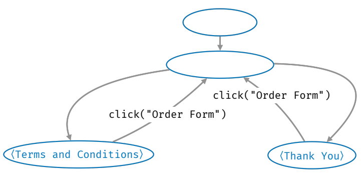
A User Interface Grammar



⟨Start⟩ ::= ⟨Order Form⟩

⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    |

A User Interface Grammar

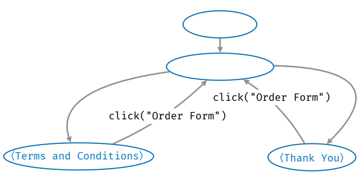


⟨Start⟩ ::= ⟨Order Form⟩

⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill(…) submit("Place Order")

Multiple transitions become alternatives.

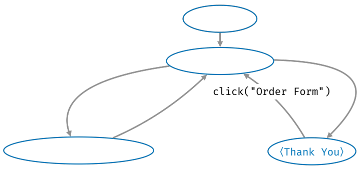
A User Interface Grammar



⟨Start⟩ ::= ⟨Order Form⟩

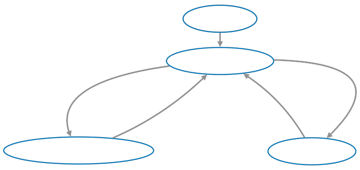
⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill(…) submit("Place Order") ⟨Thank You⟩

A User Interface Grammar




```
⟨Start⟩ ::= ⟨Order Form⟩
⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill(…) submit("Place Order") ⟨Thank You⟩
⟨Terms and Conditions⟩ ::= click("Order Form") ⟨Order Form⟩
```

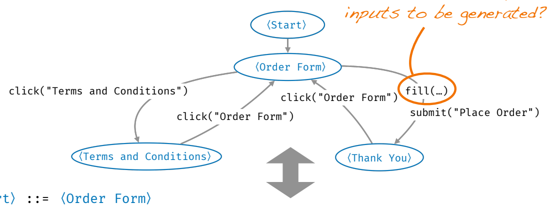
A User Interface Grammar



```
⟨Start⟩ ::= ⟨Order Form⟩
⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill(…) submit("Place Order") ⟨Thank You⟩
⟨Terms and Conditions⟩ ::= click("Order Form") ⟨Order Form⟩
⟨Thank You⟩ ::= click("Order Form") ⟨Order Form⟩
```

A User Interface Grammar

How do we specify inputs to be generated?




```
⟨Start⟩ ::= ⟨Order Form⟩
⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill(…) submit("Place Order") ⟨Thank You⟩
⟨Terms and Conditions⟩ ::= click("Order Form") ⟨Order Form⟩
⟨Thank You⟩ ::= click("Order Form") ⟨Order Form⟩
```

What you get is a grammar that is a one to one representation of the original finite state model.

A User Interface Grammar

How do we specify inputs to be generated?

```
⟨Start⟩ ::= ⟨Order Form⟩
⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill(…) submit("Place Order") ⟨Thank You⟩
⟨Terms and Conditions⟩ ::= click("Order Form") ⟨Order Form⟩
⟨Thank You⟩ ::= click("Order Form") ⟨Order Form⟩
```

Except that you can also make use of grammar features

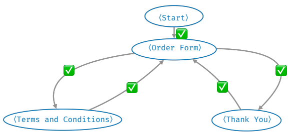
A User Interface Grammar

```
⟨Start⟩ ::= ⟨Order Form⟩

⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill("Name", ⟨Name⟩ )
      fill("Email", ⟨Email⟩ )
      fill("City", ⟨City⟩ )
      fill("Zip", ⟨Zip⟩ )
      set("T+C", ⟨Boolean⟩ )
      submit("Place order")
      ⟨Thank You⟩

⟨Terms and Conditions⟩ ::= click("Order Form") ⟨Order Form⟩

⟨Thank You⟩ ::= click("Order Form") ⟨Order Form⟩

⟨Name⟩ ::= "Andreas Zeller" | "David Lo" | "أبو علي، الحسن بن الحسن بن الهيثم"

⟨Email⟩ ::= "zeller@acm.org" | "⟨e-mail⟩"  | "'; DROP TABLE STUDENTS"
```

For instance, to characterize textual inputs.

Covering Alternatives = Model Coverage

```
⟨Start⟩ ::= ⟨Order Form⟩

⟨Order Form⟩ ::= click("Terms and Conditions") ⟨Terms and Conditions⟩
    | fill("Name", ⟨Name⟩ )
      fill("Email", ⟨Email⟩ )
      fill("City", ⟨City⟩ )
      fill("Zip", ⟨Zip⟩ )
      set("T+C", ⟨Boolean⟩ )
      submit("Place order")
      ⟨Thank You⟩

⟨Terms and Conditions⟩ ::= click("Order Form") ⟨Order Form⟩

⟨Thank You⟩ ::= click("Order Form") ⟨Order Form⟩

⟨Name⟩ ::= "Andreas Zeller" | "David Lo" | "أبو علي، الحسن بن الحسن بن الهيثم"

⟨Email⟩ ::= "zeller@acm.org" | "⟨e-mail⟩"  | "'; DROP TABLE STUDENTS"
```



If you systematically cover all alternatives in the grammar, you will also cover all transitions and all states in the finite state model.

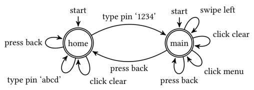
Other Benefits

- UI grammars can be used for **parsing** as well as **production**
  - ➡ Reuse and mutate existing input sequences and tests
  - ➡ Use search-based evolutionary test generation techniques
- UI grammars can encode **multiple input sources**
  - ➡ Fuzz with network inputs, intents, OS interaction
- UI grammars can be represented as visual **finite state model**
  - ➡ Just in case you prefer diagrams over text

Demo

User Interface Grammars for Android

- *Droidgram* implements UI grammar mining for Android
- Uses the *Droidmate-2* test generator to obtain initial seeds
- Seamless transition between state models and grammar models



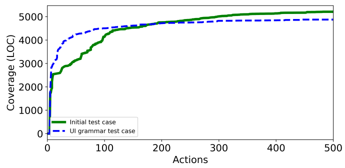

```
⟨start⟩ ::= ⟨home⟩ | ⟨main⟩
⟨home⟩ ::= Type(PIN, '1234') ⟨Type(home, PIN, '1234') ⟩ |
           Type(PIN, 'abcd') ⟨Type(home, PIN, 'abcd') ⟩ |
           Click(clear) ⟨Click(home, clear)⟩ |
           Back(home) ⟨Back(home)⟩ | ε
⟨main⟩ ::= Click(clear) ⟨Click(main, clear)⟩ |
           Click(menu) ⟨Click(main, menu)⟩ |
           Back(main) ⟨Back(main)⟩ |
           Swipe(main, left) ⟨Swipe(main, left)⟩ | ε
```

https://github.com/natanieljr/droidgram

We have implemented this approach on android, where it mines and applies UI grammars of various apps.

User Interface Grammars for Android

- Evaluated on 46 apps from F-Droid (1,347 to 72,056 statements)
- Mined UI grammars cover transitions and code *faster* with *fewer inputs*
- Integrates with any established technique for grammar-based testing



https://github.com/natanieljr/droidgram

This leads to faster exploration of states and code with fewer inputs.

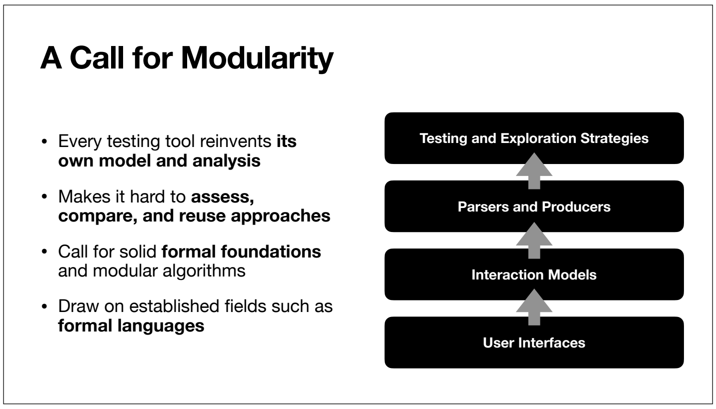
User Interfaces

"No matter how cool your interface is, it would be better if there were less of it." — *Alan Cooper*

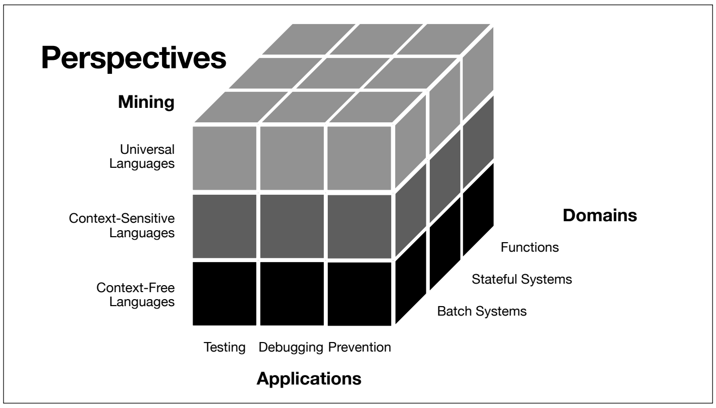
Perspectives

"It has long been an axiom of mine that the little things are infinitely the most important." — *A.C. Doyle*

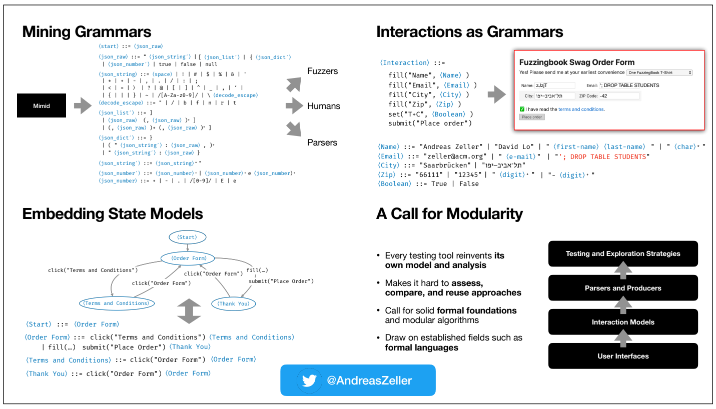
So, where does this leave us?



I think that our field can profit from a much stronger separation of models and strategies. We should make models explicit, independent from specific tools, and ground them well in theory.



Keep in mind that we are only talking about a small slice here in which formal languages can support software engineering, There are many ways languages, applications, and domains, can be combined.



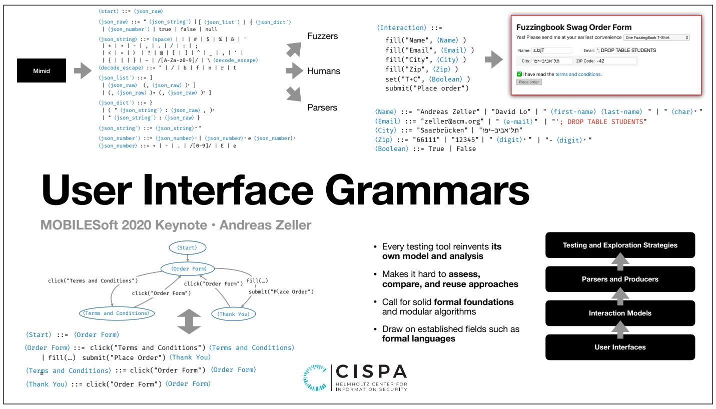
User Interface Grammars

Andreas Zeller, CISPA Helmholtz Center for Information Security
Joint work with Nataniel P. Borges Jr., Manuel Benz, and Eric Bodden

Abstract. To systematically explore user interfaces, one must cover graphical interaction features (e.g. clicks, swipes) as well as textual interaction features (e.g. form input). We introduce user interface grammars as a single formalism that captures and integrates

graphical and textual input languages. A UI grammar encodes graphical interactions and text input as a single (possibly nontrivial) stream of input events, allowing for their uniform treatment in test generation and/or coverage measurement. Grammars can be mined from existing systems (GUI-based or text-based), allow for simple customization by testers (say, for special inputs such as passwords or injection attacks) as well as guidance towards UI (model) coverage and code coverage. Includes live demos!

https://andreas-zeller.info/
https://www.cispa.saarland/



(Poster version)