Grammar-Based Testing using Realistic Domains in PHP

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, Fabrice Bouquet

April 17th, 2012
A-MOST, Montréal

Context

Web

- Its data and its languages: XML, HTML, forms, database queries, network protocols. . .
- Strings are the most used and manipulated data
- They can be complex

PHP

- Powers more than 75% of the Web
- Had nothing for automated unit test generation
- Had no types
- Is interpreted, sources are always available

Contract-Driven Testing

Principle

Exploits contracts for test purposes:

- uses preconditions to generate test data
- uses postconditions and invariants to establish test verdict by runtime assertion checking

Contracts

- Invented by B. Meyer in 1992 with Eiffel language
- Describe a model using annotations
- Express formal constraints: pre-, postconditions and invariants
- Can be included directly in the source code applied to:
  - classes attributes
  - methods arguments

Design-by-Contract

Semantics of contracts

- Contractual agreement:
  - caller commits to satisfy the pre-condition
  - called commits to establish its post-condition
- Invariants must be satisfied before and after the execution of the methods

Issue of contracts

- often expressed with logic formulæ
- hard to generate data

Previous works

- Realistic domains
  - structures to automate the validation and the generation of test data
- Praspel, a new specification language
  - adopts Design-by-Contract paradigm
  - based on realistic domains
  - implementation in PHP for PHP
- Automated unit test generator: Praspel tool
  - uses Praspel to perform Contract-Driven Testing

Motivations and contributions

Representing complex textual data

- Regular expressions are not powerful enough (regular language)
- We use grammar to represent these data (algebraic language)

Contributions

- Introduction of grammar-based testing in the Praspel tool
  - generate and validate complex textual data
  - PP, a new grammar description language
- New realistic domains: `grammar()` and `regex()`

Outline

Outline

1. Introduction
2. Context
   - Realistic domains for PHP
   - Implementation in Praspel
   - Automated unit test generator
3. Grammar-based Testing
4. Conclusion

About realistic domains

Definition and goal

- Are intended to be used for test generation purposes
- Specify a set of relevant values that can be assigned to a data for a specific context in a given program
- Provide features for the validation and generation of data values

Two important features

- **Predicability**, checks if a value belongs to the realistic domain
- **Samplability**, generates values that belong to the realistic domain

The sampler can be of many kinds: a random generator, an iterator...
Features are user-defined

Realistic domains in PHP

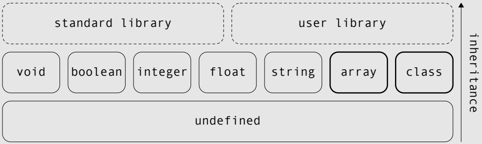
Implementation

Realistic domains as classes providing at least two methods:

- `predicate($q)`, takes a value `$q` as input, returns a boolean indicating the membership of the value to the realistic domain
- `sample($sampler)`, generates values that belong to the realistic domain according to a basic numeric-sampler `$sampler`

Hierarchical inheritance

PHP realistic domains can inherit from each other, thanks to the PHP object programming paradigm



User-defined realistic domain

How to write your own realistic domain?

- May extend an existing realistic domain
- Write the `predicate($q)` method to add constraint on the data sampled by the parent sampler
- Write a new `sample($sampler)` method (optional)

User-defined realistic domain

```
class Email extends String {

  public function predicate ( $q ) {

    $emailRegex = '...';
    return parent::predicate$(q)
        && 0 !== preg_match($emailRegex, $q);
  }

  public function sample ( $sampler ) {

    $characters  = array(...);
    $domains     = array(...);
    return $data = ...;
  }
}
```

`email()` is intended to contain all email addresses

Parameters

Principle

Realistic domains can receive parameters of many kinds: constants or realistic domains themselves

Constant arguments and realistic domains as arguments

- `boundinteger(7, 42)` contains all the integers between 7 and 42
- `string(boundinteger(4, 12), 0x20, 0x7e)` is intended to contain all the strings of length between 4 and 12 constitued of characters from 0x20 to 0x7e (Unicode code-points)

Presentation of Praspel

Praspel = PHP Realistic Annotation and SPEcification Language

- Written in the API documentation (`/** ... */`) of the PHP code
- Expresses contracts using formal constraints, called clauses, like:
  - `@invariant`, class invariant on class attributes
  - `@requires`, method precondition on class attributes and method arguments
  - `@ensures`, method postcondition on class attributes, and method arguments and result
  - `@throwable`, list of throwable exceptions by the method

Language properties

- Assignment of realistic domains to a given data (`:`)
- A predicate `\pred(...)` (expressed in the PHP syntax), enriched with the `\result` and `\old(`e`)` constructs

Class with annotations

Generic example

```
class C {

    /**
     * @invariant _foo: float();
     */
    protected $_foo = 0;

    /**
     * @requires  baz:     ... or ... and
     *            qux:     ... or ... or ...;
     * @ensures   \result: ...;
     * @throwable AnException, AnotherException;
     */
     public function bar ( $baz, $qux ) {

        return ...;
     }
}
```

Praspel clauses

Example of a short Praspel contract

```
/**
 * @requires needle:   integer() and
 *           haystack: array([to integer()], boundinteger(1, 256));
 * @ensures  \result:  boolean();
 */
public function exists ( $needle,  $haystack ) {

    $intersect = array_intersect($haystack, array($needle));

    return 0 < count($intersect);
}
```

Unit test generator and test verdict

Contract-Driven Testing

The testing process works with the two features provided by the realistic domains:

- random test data generation uses the sampler of each realistic domain composing the precondition in order to satisfy it (*samplability*)
- test verdict is given by calling the predicate of each realistic domain composing the postcondition (*predicability*)

Runtime Assertion Checking and test verdict

- The RAC is performed by instrumenting the initial PHP code with additional code that checks the contract clauses
- Detected failures can be of five kinds: precondition, postcondition, throwable, invariant or internal precondition (propagation) failure

Implementation in the Praspel tool

Environment for unit testing

- Extensible and modular framework for generating and executing online tests with a random data generator and runtime assertion checker
- Praspel and its tools are freely available in Hoa (`http://hoa-project.net`), a set of libraries for PHP

```
hywan @ hwhost /tmp/Demo: Data/Bin/myapp test:initialize Test                    11:49
Initializing a new test revision in the repository:
 * incubator from Test.                                                           [ok]
 * instrumented code.                                                             [ok]

Repository root: hoa://Data/Variable/Test/Repository/20120403115004/
hywan @ hwhost /tmp/Demo: Data/Bin/myapp test:run -r HEAD -f C.php -c C -m exists  11:50
Iteration #0

Runtime
 * C::exists(0, array(…)): The pre-condition succeed.                            [ok]
 * C::exists(0, array(…)) -> true: The post-condition succeed.                   [ok]

Contract-covering
    @requires  needle: integer()
           and haystack: array([
                   to integer()
               ], boundinteger(1, 256));
    @ensures   \result: boolean();

hywan @ hwhost /tmp/Demo: █                                                        11:50
```

Outline

1. Introduction
2. Context
3. Grammar-based Testing
   - A new grammar description language
   - Grammar-based realistic domain
   - Experimentation
4. Conclusion

Features of the PP language

We propose the PP (*PHP Parser*) language as a new grammar description language because none exists before in PHP

Token

- Lexical unit
- Represented by the PCRE (*Perl Compatible Regular Expression*)
- Namespaces (operator -> to change namespace)

Rule

- Identified by a name
- Sequence of tokens is based on the following operators:
    - repetition: $e\{x,y\}$, $e?$, $e+$, $e*$
    - concatenation: $e_1 \dots e_i \dots e_n$
    - disjunction and grouping: $e_1 \mid \dots \mid e_i \mid \dots \mid e_n$ and (e)
    - token: <t> or ::t::
    - call a rule: $r()$
    - add a marker: #n

Syntax

Simplified XML grammar expressed with PP

```
%skip   s          \s
%token  lt         <       -> tag
%token  tag:skip   \s
%token  tag:name   \w+
%token  tag:slash  /
%token  tag:gt     >       -> default
%token  content    [^<]+

xml:
    ::lt:: <name>
  ( ::slash:: ::gt:: #fold
  | ::gt:: ( <content> | xml()+ )? ::lt:: ::slash:: ::name:: ::gt:: #unfold )
```

Valid: `<a> <b>foo</b> <b>bar</b> <c /> </a>`,
but also: `<a>foo</z>`

Unification

Principle

Unification expresses another constraint in grammar.
All tokens $t[i]$ with the same i have the same value locally to a rule instance

Unified XML tag names

```
xml:
    ::lt::  <name[0]>
  ( ::slash::  ::gt::  #fold
  | ::gt::  ( <content> | xml()+ )?  ::lt::  ::slash::  ::name[0]::  ::gt::
#unfold )
```

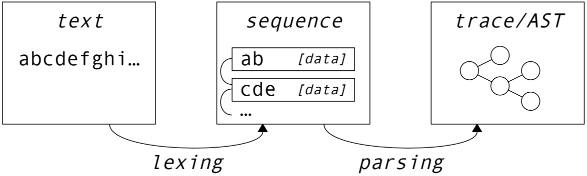
Invalid: `<a>foo</z>`

Grammar-based realistic domain

Such a realistic domain has also two features:

- **Predicability**, checks the conformance between the data and the grammar
- **Samplability**, will generate a data matching the grammar

Predicability process

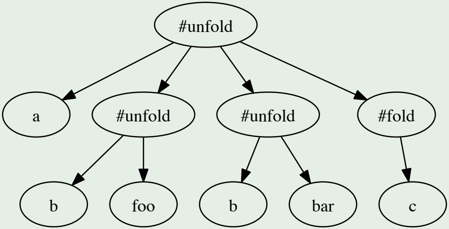


- Lexing: transform a given data into a sequence of tokens
- Parsing: analyze this sequence according to the rules
  - derive from left to right and top to bottom
  - grammar is ambiguous: $LL(*)$, implies backtracks

Abstract Syntax Tree

Classic compilation process ends by building an AST, which accepts visitors (design-pattern)

<a> foo bar <c /> associated AST



Useful for additional verifications that can not be expressed in the grammar

Grammar-based realistic domain

Such a realistic domain has also two features:

- ✓ **Predicability**, checks the conformance between the data and the grammar
  - ensured by the parsing process
- □ **Samplability**, will generate a data matching the grammar
  1. generate tokens
  2. generate sequences of tokens

Generate tokens

Example

```
([ae]+|[x-z]!){1,3}
  → ([ae]+|[x-z]!)([ae]+|[x-z]!)
  → ([ae]+)([ae]+|[x-z]!)
  → [ae][ae]([ae]+|[x-z]!)
  → e[ae]([ae]+|[x-z]!)
  → ea([ae]+|[x-z]!)
  → ea([x-z]!)
  → eay!
```

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and $*$ is set to a predefined integer

Generate tokens

Example

```
([ae]+|[x-z]!){1,3}
  → ([ae]+|[x-z]!)([ae]+|[x-z]!)
  → ([ae]+)([ae]+|[x-z]!)
  → [ae][ae]([ae]+|[x-z]!)
  → e[ae]([ae]+|[x-z]!)
  → ea([ae]+|[x-z]!)
  → ea([x-z]!)
  → eay!
```

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and $*$ is set to a predefined integer

Generate tokens

Example

```
([ae]+|[x-z]!){1,3}
  → ([ae]+|[x-z]!)([ae]+|[x-z]!)
  → ([ae]+)([ae]+|[x-z]!)
  → [ae][ae]([ae]+|[x-z]!)
  → e[ae]([ae]+|[x-z]!)
  → ea([ae]+|[x-z]!)
  → ea([x-z]!)
  → eay!
```

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and $*$ is set to a predefined integer

Generate tokens

Example

([ae]+|[x-z]!){1,3}

→ ([ae]+|[x-z]!)([ae]+|[x-z]!)

→ ([ae]+)([ae]+|[x-z]!)

→ [ae][ae]([ae]+|[x-z]!)

→ e[ae]([ae]+|[x-z]!)

→ ea([ae]+|[x-z]!)

→ ea([x-z]!)

→ eay!

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and $*$ is set to a predefined integer

Generate tokens

Example

```
([ae]+|[x-z]!){1,3}
  → ([ae]+|[x-z]!)([ae]+|[x-z]!)
  → ([ae]+)([ae]+|[x-z]!)
  → [ae][ae]([ae]+|[x-z]!)
  → e[ae]([ae]+|[x-z]!)
  → ea([ae]+|[x-z]!)
  → ea([x-z]!)
  → eay!
```

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and $*$ is set to a predefined integer

Generate tokens

Example

```
([ae]+|[x-z]!){1,3}
  → ([ae]+|[x-z]!)([ae]+|[x-z]!)
  → ([ae]+)([ae]+|[x-z]!)
  → [ae][ae]([ae]+|[x-z]!)
  → e[ae]([ae]+|[x-z]!)
  → ea([ae]+|[x-z]!)
  → ea([x-z]!)
  → eay!
```

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and ∗ is set to a predefined integer

Generate tokens

Example

([ae]+|[x-z]!){1,3}
→ ([ae]+|[x-z]!)([ae]+|[x-z]!)
→ ([ae]+)([ae]+|[x-z]!)
→ [ae][ae]([ae]+|[x-z]!)
→ e[ae]([ae]+|[x-z]!)
→ ea([ae]+|[x-z]!)
→ ea([x-z]!)
→ eay!

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and $*$ is set to a predefined integer

Generate tokens

Example

([ae]+|[x-z]!){1,3}

→ ([ae]+|[x-z]!)([ae]+|[x-z]!)

→ ([ae]+)([ae]+|[x-z]!)

→ [ae][ae]([ae]+|[x-z]!)

→ e[ae]([ae]+|[x-z]!)

→ ea([ae]+|[x-z]!)

→ ea([x-z]!)

→ eay!

Approach

- Random and uniform choices
- Isotropic exploration
- Naive but tokens are not important here, sequences of tokens are important

Repetition unfolding

Upper bound of + and * is set to a predefined integer

Rules to generate sequences of tokens

We propose 3 differents algorithms

Why?

- because we can (thanks to Praspel)
- because an algorithm to rule them all does not exist
  - to each context of use is associated an algorithm
  - we retain 3 algorithms from the literature

Algorithms

- Random and uniform generation
- Bounded exhaustive generation
- Coverage-based generation

Random and uniform generation

Example: f(), g() and $n = 5$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

$$
\begin{aligned}
f(5) &= 1.g(4) \\
g(4) &= (0+0+f(4)) \\
&\quad +(0+0+f(3)).(0+1+f(1)) \\
&\quad +(1+0+f(2)).(1+0+f(2)) \\
&\quad +(0+1+f(1)).(0+0+f(3)) \\
&\quad +(1+0+f(2)).(0+1+f(1)).(0+1+f(1)) \\
&\quad +(0+1+f(1)).(1+0+f(2)).(0+1+f(1)) \\
&\quad +(0+1+f(1)).(0+1+f(1)).(1+0+f(2)) \\
f(4) &= 1.g(3) \\
g(3) &= \ldots
\end{aligned}
$$

Approach

- An expected sequence size n and uniform probability distribution among all the possible sequences
- Recursive method to count all possible sub-structures of size n
- Counting helps to compute cumulative distribution functions, which guide exploration

Repetition unfolding

Upper bound of + and * is set to n

Random and uniform generation

Example: f(), g() and $n = 5$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

example:

	5	4	3	2	1
f	24	3	3	1	0
g	-	24	3	3	1

choice-point and probability:

```
h(3) = 20
i(2) = 6
j(2) = 14
```

$$\texttt{h:}\quad \texttt{<x>}\ (\ \underbrace{\texttt{i()}}_{\frac{6}{20}}\ \mid\ \underbrace{\texttt{j()}}_{\frac{14}{20}}\ )$$

Approach

- An expected sequence size n and uniform probability distribution among all the possible sequences
- Recursive method to count all possible sub-structures of size n
- Counting helps to compute cumulative distribution functions, which guide exploration

Repetition unfolding

Upper bound of + and $*$ is set to n

Random and uniform generation

Counting function ψ

$$
\begin{aligned}
\psi(n, e) &= \delta_n^1 \quad \text{if } e \text{ is a token} \\
\psi(n, e_1 \cdot \ldots \cdot e_k) &= \sum_{\gamma \in \Gamma_k^n} \prod_{\alpha=1}^{k} \psi(\gamma_\alpha, e_\alpha) \\
\psi(n, e_1 \mid \ldots \mid e_k) &= \sum_{\alpha=1}^{k} \psi(n, e_\alpha) \\
\psi(n, e^{\{x,y\}}) &= \sum_{\alpha=x}^{y} \sum_{\gamma \in \Gamma_\alpha^n} \prod_{\beta=1}^{\alpha} \psi(\gamma_\beta, e) \\
&\quad \text{with } 0 \leq x \leq y
\end{aligned}
$$

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size *n* (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and * is set to *n*

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of $+$ and $*$ is set to n

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and $*$ is set to n

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and $*$ is set to n

Bounded exhaustive generation

Example: f(), g() and $n = 10$

f: <a> g()
g: (<b> <c> | <d> | f()){1,3}

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and * is set to n

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and * is set to n

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  &lt;a&gt; g()
g:  ( &lt;b&gt; &lt;c&gt; | &lt;d&gt; | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and ∗ is set to n

Bounded exhaustive generation

Example: f(), g() and $n = 10$

```
f:  <a> g()
g:  ( <b> <c> | <d> | f() ){1,3}
```

1. <a> <b> <c>
2. <a> <d>
3. <a> <a> <b> <c>
4. <a> <a> <d>
5. <a> <a> <a> <b> <c>
6. <a> <a> <a> <d>
7. <a> <a> <a> <a> <a> <b> <c>
8. ...

more than 10000 solutions

Approach

- Generate all possible sequences (exhaustive) up to a given size n (bounded)
- The algorithm behaves like an iterator
- Based on multiset (set with repetition)

Repetition unfolding

Upper bound of + and $*$ is set to n

Bounded exhaustive generation

Function β for bounded exaustive generation

$$
\begin{aligned}
\beta(1, e) &= \{\texttt{sample}(e)\} \quad \text{if } e \text{ is a token} \\
\beta(n, e) &= \{\} \quad \text{if } n \neq 1 \\
\beta(n, e_1 \mid e_2) &= \beta(n, e_1) \cup \beta(n, e_2) \\
\beta(n, e_1 \cdot e_2) &= \bigcup_{p=1}^{n-1} \beta(p, e_1) \cdot \beta(n-p, e_2) \\
\beta(n, e^{\{x,y\}}) &= \bigcup_{p=x}^{y} \beta(n, e^p) \\
\beta(n, e^\star) &= \bigcup_{p=0}^{n} \beta(n, e^p) \\
\beta(n, e^+) &= \beta(n, e \cdot e^\star) \\
\beta(n, e^0) &= \{\} \\
\beta(n, e^1) &= \beta(n, e) \\
\beta(n, e^p) &= \beta(n, e \cdot e^{p-1}) \quad \text{if } p \geq 2
\end{aligned}
$$

Coverage-based generation

Example: f(), g()

```
f:  <a> g()
g:  ( <b> <c> | <d> | f()
){1,3}
```

1. <a> <d> <b> <c> <a> <d>

1 solution

Repetition unfolding

- * is bounded to 0, 1 or 2
- + is unfolded 1 or 2 times
- {x,y} is unfolded x, $x+1$, $y-1$ and y times

Approach

- Reduce the combinatorial explosion
- Aims at producing data that activate all the branches of the grammar rules
- A rule is said to be covered if and only if its sub-rules have all been covered
- A token is said to be covered if it has been successfully used in a data generation
- To ensure diversity, a random choice is made amongst the remaining sub-rules of a choice-point to cover
- Boundary test generation heuristics to avoid combinatorial explosion and guarantee the termination

Coverage-based generation

Coverage-based function ϕ

$$
\begin{aligned}
\phi(p, e) &= [\texttt{sample}(e)] \quad \text{when } e \text{ is a token} \\
\phi(p, e_1 \cdot e_2) &= \phi(\phi(p, e_1), e_2) \\
\phi(p, e_1 \mid \ldots \mid e_k) &= \phi(p, e_1) \oplus \ldots \oplus \phi(p, e_k) \\
\phi(p, e^?) &= [] \oplus \phi(p, e) \\
\phi(p, e^\star) &= [] \oplus \bigoplus_{i=1}^{\infty} \phi(p, \underbrace{e \cdot \ldots \cdot e}_{i}) \\
\phi(p, e^+) &= \bigoplus_{i=1}^{\infty} \phi(p, \underbrace{e \cdot \ldots \cdot e}_{i}) \\
\phi(p, e^{\{x,y\}}) &= \bigoplus_{i=x}^{y} \phi(p, \underbrace{e \cdot \ldots \cdot e}_{i})
\end{aligned}
$$

Pros and cons

Complex textual data generation

- Random and uniform generation:
  - ☺ fast for small data, diversity of data and fixed size
  - ☹ counting phase is exponential ($n > 10$ needs at least 2 hours), despite that generation is fast
- Bounded exhaustive generation:
  - ☺ fast for small data and exhaustiveness is efficient
  - ☹ exponential number of data
- Coverage-based generation:
  - ☺ fast for medium and big data and diversity of data
  - ☹ do not consider size of data

Grammar as a realistic domain

Such a realistic domain has also two features:

- ✓ **Predicability**, checks the conformance between the data and the grammar
  - ensured by the parsing process
- ✓ **Samplability**, will generate a data matching the grammar
  - ensured by one of the three algorithms

Experimentation

Self-validation

- Generate and validate data with PP and other parsers
- Considered grammars: JSON and PCRE, with other parsers from Mozilla Gecko and PHP
- All produced data were parsed correctly

Mutation

- Then, we consider simple grammar mutation operators
- Some generated data were parsed correctly by PP but not by other parsers (due to the backtracking)
- After fixing the bug, we performed the same kinds of experiments with PP and other parsers

Outline

Conclusion

What have we seen?

- Realistic domains specifying data and providing two useful features for automated test generation: predicability and samplability
- Praspel, a new Design-by-Contract language implementing realistic domains
- Praspel tool: a test generation and execution framework to automate unit testing in PHP
- Grammar-based Testing is introduced in Praspel
- PP, a new grammar description language
- Two new realistic domains join the standard library

Conclusion

Future works

- Extend case studies in order to evaluate the relevance of the coverage-based test generation, in terms of fault detection and code coverage of the system under test
- Improve the generation algorithms so as to avoid rejection as much as possible (look at UDITA)
- Implement Praspel into other languages (e.g. Java, C, Javascript)

Thanks!

Thank you for your attention! Any questions?