A Constraint Solver for PHP Arrays

Ivan Enderlin[1], Alain Giorgetti, Fabrice Bouquet

March 22th, 2013
CSTVA, Luxembourg

[1]Funded by Squash

Context

Web

- Many mixed technologies
- Strings and arrays are the most used and manipulated data
- They can be complex

PHP

- Language that powers more than 75% of the Web
- No tool for automatic unit tests generation
- Highly dynamic: no type
- Interpreted: source code 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
  - callee 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

Praspel, a Specification Language for Contract-Based Testing (ICTSS'11)

- 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

Grammar-Based Testing using Realistic Domains (A-MOST'12)

- Representing and generating complex textual data with grammar
- Grammar-based testing in Praspel
  - PP, a new grammar description language
  - two new realistic domains: `grammar()` and `regex()`

Motivations and contributions

Arrays

- Representing and generating PHP arrays
- PHP arrays cover most of the needs for collections:
  - store key-value pairs of any kinds (hashmap)
  - no specific length or depth
  - efficiently implemented

Contributions

- Study to known the most popular constraints on PHP arrays
- Formalize these constraints
- A constraint solver implemented in PHP

Outline

Outline

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

Inheritance and interfaces

Inheritance

Since realistic domains are implemented in PHP as classes, they can:

- inherit from each other
- refine inherited features of the parent
- describe a hierarchical universe

Interfaces

Some realistic domains implement interfaces which characterize them:

- `Constant`, immutable realistic domain with one value (42, `true`, etc.)
- `Interval`, interval with a lower and an upper bound
- `Nonconvex`, discreditable values, i.e. specify a value that no longer belongs to a realistic domains and should therefore not be generated
- `Finite`, count the number of values
- `Enumerable`, iterate over all the values

PHP Realistic Annotation and SPEcification Language

Annotated class

```
class C {

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

    /**
     * @requires x: integer() or string('a', 'z', 1);
     * @ensures \result: boolean();
     */
    public function f ( $x ) { ... }
}
```

Expresses contracts using formal constraints, called clauses.

PHP Realistic Annotation and SPEcification Language

Annotated class

```
class C {

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

    /**
     * @requires x: integer() or string('a', 'z', 1);
     * @ensures \result: boolean();
     */
    public function f ( $x ) { ... }
}
```

`@invariant`: class invariant on class attributes

PHP Realistic Annotation and SPEcification Language

Annotated class

```
class C {

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

    /**
     * @requires x:  integer() or string('a', 'z', 1);
     * @ensures \result:  boolean();
     */
    public function f ( $x ) { ... }
}
```

`@requires`: method precondition on class attributes and method arguments

PHP Realistic Annotation and SPEcification Language

Annotated class

```
class C {

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

    /**
     * @requires x:  integer() or string('a', 'z', 1);
     * @ensures \result:  boolean();
     */
    public function f ( $x ) { ... }
}
```

`@ensures`: method postcondition on class attributes, and method arguments and result

PHP Realistic Annotation and SPEcification Language

Annotated class

```
class C {

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

    /**
     * @requires x:  integer() or string('a', 'z', 1);
     * @ensures \result:  boolean();
     */
    public function f ( $x ) { ... }
}
```

Realistic domains can receive parameters: it helps generating structured data such as arrays, objects, graphs, automata, etc.

Outline

Arrays in PHP

Associative array

An array is always an associative array, a collection of key-value pairs:

- keys appear at most once
- keys are reduced to null, integers or strings
- values can be of many kinds

Content

- homogeneous: all keys have the same type, idem for values
- heterogeneous: keys and values can have distinct types

Length

The array length (or size) is its number of elements:

- no predefined length
- stored internally by the PHP engine
- can be retrieved with the PHP function `count()`

Arrays in Praspel

$$\texttt{array}(D,\ L)$$

- D, a comma-separated list, between `[` and `]`, of *array descriptions* of the form `from` K `to` V
  - K and V are realistic domain disjunctions, respectively for keys and values
- L, a disjunction of realistic domains of non-negative integers

Examples of array declaration

```
a1: array([to boolean()], 7..42)
a2: array([from  0..5 or 10 to integer()], 7)
a3: array([from  0..10 to boolean(),
           from 20..30 to float()], 7)
a4: array([from  0..10 or 20..30 to boolean() or float()], 7)
```

Normal form

Implicit domain:

$$\texttt{to}\ T_1\ \texttt{or}\ T_2 \quad\rightarrow\quad \texttt{from natural(0, 1) to}\ T_1\ \texttt{or}\ T_2$$

Disjunction removal:

$$\texttt{from}\ F_1\ \texttt{or}\ F_2\ \texttt{to}\ T_1\ \texttt{or}\ T_2 \quad\rightarrow\quad \begin{array}{l}\texttt{from}\ F_1\ \texttt{to}\ T_1\texttt{,}\\ \texttt{from}\ F_1\ \texttt{to}\ T_2\texttt{,}\\ \texttt{from}\ F_2\ \texttt{to}\ T_1\texttt{,}\\ \texttt{from}\ F_2\ \texttt{to}\ T_2\end{array}$$

a4 in normal form

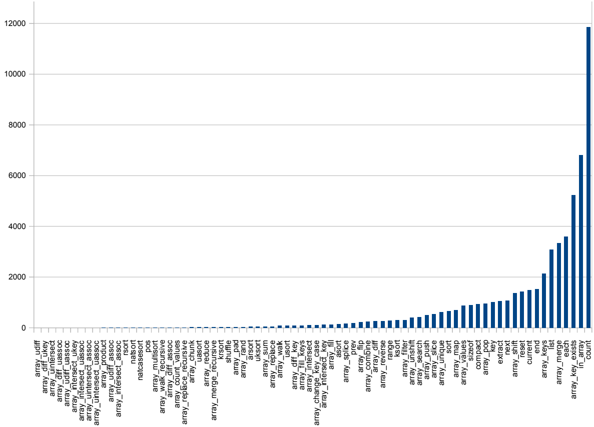
```
a4: array([from  0..10 or 20..30 to boolean() or float()], 7)

      ↓

a4: array([from  0..10 to boolean(),
           from  0..10 to float(),
           from 20..30 to boolean(),
           from 20..30 to float()], 7)
```

Frequently used array functions

- selected 61 PHP projects (for their popularity, impact on the industry and complexity)
- represent 28 066 files and 5 220 547 lines of code
- three most used functions are: `count()`, `in_array()` and `array_key_exists()`



Array Conditions

Extend syntax of declaration with array *conditions*:

Pair condition

$$\texttt{a[}K\texttt{]:}\ V$$

- K and V are realistic domain disjunctions
- all the keys in K are keys of the array `a`
- all the values associated to keys in K are in V
- K only accepts realistic domains that implements at least the `Constant`, `Interval` and `Enumerable` interfaces
- equivalent to use `array_key_exists()` and `in_array()` combined

Array Conditions

Key condition

$$\texttt{a}[K]\texttt{:}\ \_$$

- all the keys from K must be present in the array a
- equivalent to use `array_key_exists()` with all values in K in conjunction

Value condition

$$\texttt{a}[\_]\texttt{:}\ V$$

- all the values in V must be present in the array a
- equivalent to use `in_array()` with all values in V in conjunction

Array Conditions

Pair negation condition

$$\texttt{a[}K\texttt{]!:}\ V$$

All the keys in K have a value in the array a. None of this value is in V.

Key and value negation condition

$$\texttt{a[}K\texttt{]!:}\ \_$$

No key in K appears in a.

$$\texttt{a[\_]!:}\ V$$

No value in V appears in a.

Array Conditions

Unicity of values

```
a is unique
```

expresses unicity condition of values. We cannot have the same value twice in the array a.

Reminder: keys are always unique.

Running example, system = conjunction of array conditions

```
length: 0..5 or 10
a     : array([to string('a', 'e', 1)], length)
a[0]  : 'b' or 'd'
a is unique
```

Outline

Conditions to constraints

The solver constructs an array satisfying all the conditions.

$$\texttt{a:}\quad \texttt{array(}D\texttt{,}\ L\texttt{)}$$

- D is `[from` F_1 `to` T_1`, ..., from` F_p `to` T_p`]` with $1 \leq p$, in normal form
- L is L_1 `or ... or` L_m with $1 \leq m$ and are realistic domains that inherit from the `Integer` realistic domain and that are non-negatives

Example

```
length: 0..5 or 10
a     : array([to string('a', 'e', 1)], length)
```

- $p = 1$ and $m = 2$
- $L_1 = [0..5]$ and $L_2 = \{10\}$
- $F_1 =$ `natural(0, 1)` and $T_1 =$ `string('a', 'e', 1)`

Variables

Constraint variables are:

- array size S, which is a non-negative integer
- sets X and Y, which respectively are the array domain (set of keys) and codomain (set of values)
- array content H, which is a total function from X to Y
- realistic domains $X_1, \ldots, X_p$ (resp. $Y_1, \ldots, Y_p$), which are subsets of the realistic domains $F_1, \ldots, F_p$ (resp. $T_1, \ldots, T_p$) compatible with all the array conditions

Goal: finding a value of H, X_i and Y_i.

Cardinality Constraints

Let $\text{card}(E)$ denotes the cardinality of the finite set E.

$$\begin{array}{rcll} \text{card}(X) & = & S & \text{size is the number of keys} \\ S & \geq & 0 & \text{size is non-negative} \\ \text{card}(X) & \geq & \text{card}(Y) & \text{no unicity on values (by default)} \\ \text{card}(X) & = & \text{card}(Y) & \texttt{a is unique} \end{array}$$

The array size S should be in one of the sets $L_1, \dots, L_m$, so:

$$S \in L_1 \cup \dots \cup L_m$$

Example

`length: 0..5 or 10`

$L_1 \subseteq [0..5]$ and $L_2 \subseteq \{10\}$. The size S is constrained by $S \in L_1 \cup L_2$.

Constraints on Pairs

When $x \in X$ holds, $H(x) = y$ means that the key-value pair (x, y) is in the array. We extend H to any subset E of X with H' defined by:

$$H'(E) = \{H(x) \text{ s.t. } x \in E\}$$

For each array condition $\mathtt{a}[K]:\ V$, where K and V are domain disjunctions, we introduce the constraints:

$$K \subseteq X \qquad H'(K) \subseteq V$$

Translation of a pair condition

`a[0]: 'b' or 'd'`

$K = \{0\}$, $V = \{\text{'b'}, \text{'d'}\}$. The constraints are $\{0\} \subseteq X$ and $H'(\{0\}) \subseteq \{\text{'b'}, \text{'d'}\}$.

Propagation, Consistency and Labelling

Propagation and consistency

- Propagation uses an AC3 algorithm implemented in PHP
- All kinds of realistic domains (`Constant`, `Interval`, `Nonconvex`, etc.) have a *revise* method to reduce the domain
- The consistency is checking there is no empty domain for the four variables S, H, X and Y

Labelling

Heuristic to converge quickly: chosing a value for the variable S at first. Then, the solver tries to compute the sets X and Y:

- use a random generator to generate a value in a domain disjunction
- the generated value is then propagated
- if an inconsistency is detected, a new constraint is added (if $S = 5$ leads to an inconsistency, we add the constraint $S \neq 5$)

When all variables are labelled, the solver returns the solution.

Outline

Evaluate the solver efficiency

Its capability to avoid or reduce rejection when generating data from systems (i.e. conjunction of array conditions).
Measures:

- the number of backtracks in the solver
- the time to generate data from satisfiable systems
- how many unsatisfiable systems are detected

Process

The experimentation is composed of three steps:

- system generation
- data generation (i.e. system solving)
- measuring step

We generate systems on arrays containing strings and integers, and of length 5 to 20.

System and data generation

System generation

- Use the Praspel grammar, including array conditions
- Apply previous work on grammar-based testing
- Use the bounded-exhaustive generation algorithm:
  - lot of data
  - all branches are covered
  - all tokens are covered

Data generation

- Call the solver on each produced system to generate an array satisfying it.
- Every generated array is evaluated by the predicability feature associated to the system of array conditions, to check the solver soundness.

Measures

During data generation:

- count only the array generation time
- measure the number of backtracks in the solver
- measure the number of rejected systems

n	generated systems	backtracks	backtracks per system	rejected systems	generation time (ms)
10	14	0	0	0	6.484
15	86	34	0.40	0	42.167
18	210	91	0.43	0	141.694
19	275	103	0.37	0	229.001
20	492	114	0.23	0	372.241

What have we seen?

- An extension of the Praspel language to specify usual conditions on PHP arrays
- Semantics of conditions expressed by constraints
- Designed and implemented a constraint solver in PHP to generate test data
  - integrated in realistic domains
  - can be used within the Praspel framework
- Rejection has been totally removed

Future works

- Extend the experimentation:
  - generate inconsistencies and observe the solver behavior
- Formalize more constraints and extend the solver
- Transform constraints into the formalism of MiniZinc:
  - compare to other solvers regarding performances and capabilities to find solutions
- Extend solver to strings (because a string is an array of characters)

Thanks!

Thank you for your attention! Any questions?

References

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Fabrice Bouquet, *Grammar-Based Testing Using Realistic Domains in PHP*, ICST (Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, eds.), IEEE, 2012, pp. 509–518.

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman, *Praspel: A specification language for contract-based testing in php*, ICTSS (Burkhart Wolff and Fatiha Zaïdi, eds.), Lecture Notes in Computer Science, vol. 7019, Springer, 2011, pp. 64–79.