Praspel: A Specification Language for Contract-Driven Testing in PHP

Ivan Enderlin Frédéric Dadeau Alain Giorgetti
Abdallah Ben Othman

November 9th, 2011
ICTSS, Paris

Motivations

Context

- Types are the main verification mechanism massively adopted
- Annotation is more and more a common processus for developers
- Unit testing is used to maintain large softwares (with a thousand of manual tests)

Initial ideas

- An easy language to express contracts
- Automate tests (data) generation and execution
- A complete, clean and modular test framework

Design-by-Contract

Definition

- Invented by B. Meyer in 1992 with Eiffel language
- Describes a model using annotations
- Expresses formal constraints: pre-, postconditions, invariants. . .
- Included directly in the source code: classes attributes, methods arguments. . .

Semantics of contracts

- Contractual agreement:
  - caller: “I commit to satisfy your pre-condition when I’m calling you”
  - called: “In this case, I commit to establish my post-condition”
- Invariants must be satisfied before and after the execution of the methods

LIFC

UFC

Design-by-Contract

Existing contract-based specification languages

- Spec#: for C# language, contracts are written in C#
- JML: Java Modeling Language, contracts are expressed with logic formulæ
- ACSL: ANSI/C Specification Language, adds algebraic structures
- Nothing for PHP

Initially designed for verification (static or dynamic)

Contract-Driven Testing

Definition

Exploits the contract for generating tests:

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

Issue

- How to describe realistic data for being able to generate them?

Contributions

- Realistic domains
  - overlay/refinement of types
  - structures to automate the validation and the generation for 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
  - uses Praspel to perform Contract-Driven Testing

Outline

1. Realistic domains for PHP
2. Implementation in Praspel
3. Automated unit test generator
4. Experimentation
5. Conclusion

About of realistic domains

Definition and goal

- Specify a set of relevant values that can be assigned to a data for a specific context (e.g. an email address) in a given program
- Come with properties for the validation and generation of data values
- Realistic domains are intended to be used for test generation purposes

Two important properties

- **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, a walk in the domain, an incrementation of values etc.

Properties are implemented by the end-user.

Realistic domains in PHP

Implementation

In PHP, we have implemented realistic domains as classes providing at least two methods, corresponding to the two features of realistic domains:

- `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`

Skeleton of a realistic domain

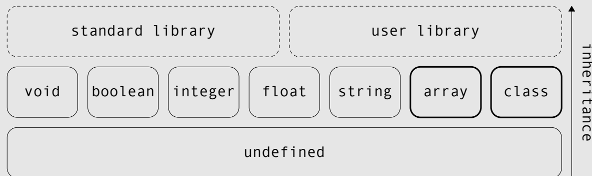
```
class MyOwnRealdom extends ... {

    public function predicate ( $q                ) { return $aBoolean = ...; }
    public function sample    ( Sampler $sampler ) { return $data     = ...; }
}
```

Hierarchy

Our implementation exploits the PHP object programming paradigm:

Hierarchical inheritance

PHP realistic domains can inherit from each other



What does it imply?

The predicate of a child realistic domain can refine its parent predicate by adding new constraints. Default sampler: parent sampler and rejection.

Parameters

Parameterizable

Realistic domains may have parameters. They can receive arguments of many kinds: constants or realistic domains themselves

Basic realistic domain with constant arguments

`boundinteger(7, 42)` contains all the integers between 7 and 42

Realistic domain with constants and realistic domains as arguments

`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)

User-defined realistic domain

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

Outline

1. Realistic domains for PHP
2. Implementation in Praspel
   - Assigning realistic domains to data
   - Designing contracts in Praspel
3. Automated unit test generator
4. Experimentation
5. Conclusion

Presentation of Praspel

Signification and format

- PHP Realistic Annotation and SPEcification Language
- Written in the API documentation (`/** ... */`)
- Makes it possible to express formal constraints, called *clauses* (starting with the standard `@` symbol)

Operator :

Syntax and semantics

The syntactic construction:

$$i: \quad t_1\texttt{(...)} \quad \texttt{or} \quad \texttt{...} \quad \texttt{or} \quad t_n\texttt{(...)}$$

associates at least one realistic domain among $t_1(\ldots)$, ..., $t_n(\ldots)$ to an identifier i.

Examples of realistic domains declarations

- `y: integer() or float() or boolean()` means that `y` can either be an integer, a floating-point number or a boolean
- `u: email() or userLogin()` specifies that `u` is either an email address or a user login

Array description

Syntax and semantics

An array *description* has the following form:

$$
\begin{array}{lllllll}
\texttt{array([} & \texttt{from} & T_1^1\texttt{(...)} & \texttt{or ... or} & T_i^1\texttt{(...)} & & \\
 & \texttt{to} & T_{i+1}^1\texttt{(...)} & \texttt{or ... or} & T_n^1\texttt{(...),} & & \\
 & \vdots & & & & & \\
 & \texttt{from} & T_1^k\texttt{(...)} & \texttt{or ... or} & T_j^k\texttt{(...)} & & \\
 & \texttt{to} & T_{j+1}^k\texttt{(...)} & \texttt{or ... or} & T_m^k\texttt{(...)} & \texttt{],} & \mathit{size}\texttt{)}
\end{array}
$$

`from` *domains* `to` *co-domains* (*pairs* separated by `,`), each ones are domains disjunctions.

Array description

Examples of arrays

- a_1: `array([from integer() to boolean(), boundinteger(7, 42))`
- a_2: `array([to boolean(), to float()], 7)`
- a_3: `array([to boolean() or float()], 7)`
- a_4: `array([to integer()], boundinteger(1, 256))`

Defining a contract clause

Contract content

- either the assignment of a realistic domain to a given data (`:`)
- or it is a predicate `\pred(...)` (expressed in the PHP syntax)
- enriched with the `\result` and `\old(`*e*`)` constructs

Praspel clauses

Applied on classes:

- `@invariant`, invariant on class attributes

Applied on methods:

- `@requires`, precondition on class attributes and method arguments
- `@ensures`, postcondition on class attributes, and method arguments and result
- `@throwable`, list of throwable exceptions by the method

Praspel clauses

Example of a short Praspel contract

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

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

    if(null === $matches)
        $matches = array_keys($intersect);

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

Organization in behaviors

Several behaviors per method

- Each `@behavior` clause has a unique name inside a contract
- A behavioral clause contains `@requires`, `@ensures` and `@throwable` clauses
- By default, a global implicit behavior exists

`matches` is no longer undefined

```
/**
 * @behavior get_matches {
 *     @requires matches: void();
 *     @ensures  matches: array(
 *                            [to boundinteger(0, 255)],
 *                            boundinteger(0, 256)
 *                        );
 * }
 */
```

Outline

1. Realistic domains for PHP
2. Implementation in Praspel
3. **Automated unit test generator**
4. Experimentation
5. Conclusion

Unit test generator and test verdict

Contract-Driven Testing

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

- (*samplability*) the sampler is implemented as a random data generator that satisfies the precondition of the method
- (*predicability*) the predicate makes it possible to check the postcondition at runtime after the execution of the method

Random test data generation

- Test data generation are used in the `sample()` method
- `\pred` thus introduces rejection in precondition
- Random generation was a first approach

Unit test generator and test verdict

Contract-Driven Testing

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

- (*samplability*) the sampler is implemented as a random data generator that satisfies the precondition of the method
- (*predicability*) the predicate makes it possible to check the postcondition at runtime after the execution of the method

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

LIFC

UFC

Example of instrumentation

Instrument foo()

```
public function foo ( ... ) {

    $this->foo_contract();
    $this->foo_pre(...);

    try {

        $result = $this->foo_body();
    }
    catch ( \Exception $e ) {

        $this->foo_exception($e);
        throw $e;
    }

    $this->foo_post($result, ...);

    return $result;
}
```

```
public function foo_pre ( ... ) {

    // ...
    return    $contract->verifyInvariants(...)
           && $contract->verifyPreCondition(...);
}

public function foo_post ( $result, ... ) {

    // ...
    return    $contract->verifyPostCondition(
                  $result, ...
              )
           && $contract->verifyInvariants(...);
}

public function foo_exception ( $exception ) {

    return    $contract->verifyException($exception)
           && $contract->verifyInvariants(...);
}
```

Implementation in the Praspel tool

Environment for unit testing

- Clean and modular framework for generating and executing online tests
- Praspel and its tools are freely available in Hoa (`http://hoa-project.net`), a set of libraries for PHP

Outline

1 Realistic domains for PHP

2 Implementation in Praspel

3 Automated unit test generator

4 Experimentation

5 Conclusion

Experimentation

Case study

Validate HTML code produced by second year student programs written in PHP, with different granularities:

- firstly with a general oracle: is the HTML markup well-formed?
- secondly with a refined oracle: do attributes exist, are they well-positionned and are the values right?

Grammar as a dedicated realistic domain

- Grammars can be used to validate or generate data
- It is a new realistic domain
- Then, we have written a grammar of HTML for the first oracle

LIFC

UFC

Observations

Simple and easy

- Experimentation shows that tools are easy to use (2 commands), find bugs quickly and easily
- Simple mechanism to describe an oracle
- Simple mechanism to develop a new realistic domain (here, for grammar)
- It offers a good ratio effort/results (bugs found)

Outline

1. Realistic domains for PHP
2. Implementation in Praspel
3. Automated unit test generator
4. Experimentation
5. Conclusion

What we have seen?

- Realistic domains specifying test data for program variables
  - provide two useful features for automated test generation: *predicability* and *samplability*
  - related work: UDITA has the same approach
- Praspel, a new Design-by-Contract language implementing realistic domains
- A test generation and execution framework to automate unit testing in PHP
- Presently, standard library of 31 realistic domains in Hoa (`date`, `timestamp`, `regex`, `bag`, ...)
- We have introduced Grammar-based Testing into a realistic domain

Future works

- Extend and generalize the concept of realistic domains to other programming languages to illustrate the benefits of this concept
- Improve data generators (Search-based Testing, Constraint-based Testing etc.)
- Extending standard library of realistic domains

⋆ Fill the box!

UFC

Thanks!

Thank you for your attention! Any questions?