Grammar-Based Testing using Realistic Domains in PHP

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti and Fabrice Bouquet
Institut FEMTO-ST UMR CNRS 6174 - University of Franche-Comté INRIA CASSIS Project
16 route de Gray - 25030 Besançon cedex, France
Email: {ivan.enderlin,frederic.dadeau,alain.giorgetti,fabrice.bouquet}@femto-st.fr

April 20, 2012

Abstract

This paper presents an integration of grammar-based testing in a framework for contract-based testing in PHP. It relies on the notion of realistic domains, that make it possible to assign domains to data, by means of contract assertions written inside the source code of a PHP application. Then a test generation tool uses the contracts to generate relevant test data for unit testing. Finally a runtime assertion checker validates the assertions inside the contracts (among others membership of data to realistic domains) to establish the conformance verdict. We introduce here the possibility to generate and validate complex textual data specified by a grammar written in a dedicated grammar description language. This approach is tool-supported and experimented on the validation of web applications.

Keywords: Grammar-based testing, contracts, realistic domains, PHP, random generation, rule coverage.

1 Introduction

Model-based testing [1] is a technique according to which a (preferably formal) model of the System Under Test (SUT) is employed in order to validate it. In this context, the model can be of two uses. First, it can be used to compute the test cases by providing an exploitable abstraction of the SUT from which test data or even test sequences can be computed. Second, the model can provide the test oracle, namely the expected result against which the execution of the SUT is checked. Model-based testing makes it possible to automate the test generation. It is implemented in many tools, based on various modelling languages or notations [2]. In addition, model-based testing is also an efficient approach for testing software evolution and regression, mainly by facilitating the maintainability of the test repository [3].

Even though model-based testing is very convenient in theory, its application is often restricted to critical and/or embedded software that require a high level of validation [4]. Several causes can be identified. First, the design of the formal model represents an additional cost, and is sometimes more expensive than the manual validation cost. Second, the design of the model is a complex task that requires modelling skills and understanding of formal methods, thus necessitating dedicated skilled engineers for being put into practice. Finally, as the model represents an abstraction of the SUT, the distance between the model and the considered system may vary and thus an additional step of test concretization is often required to translate abstract test cases (computed from the model) into executable test scripts (using the API provided by the SUT and the concrete data that it operates).

Contract-based testing [5] has been introduced in part to address these limitations. It is based on the notion of Design by Contract (DbC) [6] introduced by Meyer with Eiffel [7]. A contract is a way to embed a piece of model inside the code of a program. It mainly consists of two kinds of simple modelling elements: invariants describe properties that should hold at each step of the execution, pre- and postconditions respectively represent the conditions that have to hold for an operation/method to be invoked, and the conditions that have to hold after the execution of the operation/method.

Various contract languages extend programming languages, such as JML for Java [8], ACSL for C [9], Spec# for C# [10]. The advantages of contracts are numerous: they make it possible to introduce formal properties inside the code of a program, using annotations. Besides, the properties are expressed in the same formalism as the code, without any gap due to the abstraction level. Moreover, properties can be exploited for (unit) testing. Indeed, the information contained in invariants and preconditions can be used to generate test data. In addition, these assertions can be checked at run time (preconditions and invariants are checked at the beginning of the method, postconditions and invariants at their end) and thus provide a (partial) test oracle for free. The test succeeds if no assertion is violated, and fails otherwise.

In a previous work, we have introduced Praspel, a tool-supported specification language for contract-based testing in PHP [11]. Praspel extends contracts with the notion of *realistic domain*, which makes it possible to assign a domain of values to data (class attributes or method parameters). Realistic domains present two useful features for testing: *predicability*, which is the possibility to check that a data belongs to its associated domain, and *samplability*, which is the possibility to automatically generate a data from a realistic domain. A library for predefined basic realistic domains (mainly scalar domains, strings, arrays) is already available along with a test environment.[1] It is to be noted that one of the main arguments against annota-

[1]Freely available at `http://hoa-project.net`.

tion languages is that they require the source code of the application to be employed, which prevents them from being used in a black-box approach. Nevertheless, applying contract-based testing to interpreted languages, such as PHP, makes full sense, since this limitation does not apply.

Validating PHP web applications often involves the generation of structured textual test data (e.g. specific pieces of HTML code produced by a web application, email addresses, SQL or HTTP queries, or more complex messages). To facilitate the use of Praspel in this context, we provide a grammar-based testing [12] mechanism that makes it possible to express and validate structured textual data.

The contributions of this paper are twofold. First, we introduce a grammar description language to be use with PHP, named PP (for PHP Parser). This language makes it possible to describe tokens and grammar rules, that are then exploited by a dedicated PHP interpreter to validate a text w.r.t. the expected syntax given by the grammar. Second, we provide grammar-based testing mechanisms that automatically generate instances of text using various data generators. These two contributions are gathered into two new realistic domains that can be used in Praspel annotations in a PHP program: a realistic domain for regular expressions and a realistic domain for grammars, parameterized by a grammar description file.

The paper is organized as follows. Section 2 explains the notion of realistic domain and presents its implementation in Praspel for PHP. Then, the PP language and its semantics are defined in Section 3. Section 4 proposes data generation algorithms from grammars. Then, we report in Section 5 some experiments aiming at validating our tool and showing its usefulness and efficiency in practice. Related works are presented in Section 6. Finally, Section 7 concludes and presents future works.

2 Realistic Domains and Praspel

This section presents the notion of realistic domain and its application to PHP programs [11]. Realistic domains are designed for test generation purposes. They specify which values can be assigned to a data in a given program. Realistic domains are well-suited to PHP, since this language is dynamically typed (i.e. no types are assigned to data) and realistic domains thus introduce a specification of data types that are mandatory for test data generation. We first introduce general features of realistic domains, and then present their application to PHP.

2.1 Features of Realistic Domains

Realistic domains can represent all kinds of data; they are intended to specify relevant data domains for a specific context. Realistic domains are more subtle than usual datatypes (integer, string, array, etc.) and refine these latter. For example, if a realistic domain specifies an email address, we can validate and generate strings representing syntactically correct email addresses; this can be done using a regular expression that matches email addresses. Realistic domains display two necessary features for the validation and generation of data values, which are now described and illustrated.

2.1.1 Predicability

The first feature of a realistic domain is to carry a characteristic predicate. This predicate makes it possible to check if a value belongs to the possible set of values described by the realistic domain.

2.1.2 Samplability

The second feature of a realistic domain is to propose a value generator, called the *sampler*, that makes it possible to generate values in the realistic domain. The data value generator can be of many kinds: a random generator, a walk in the domain, an incrementation of values, etc.

2.2 Realistic Domains in PHP

In PHP, we have implemented realistic domains as classes providing at least two methods, corresponding to the two features of realistic domains. The first method is named `predicate($q)` and takes as input a value `$q`: it returns a boolean indicating the membership of the value to the realistic domain. The second method is named `sample()` and generates values that belong to the realistic domain. An example is the class `EmailAddress` for email addresses reproduced in Figure 1.

Our implementation of realistic domains in PHP exploits the PHP object programming paradigm and takes benefit from the following two principles.

2.2.1 Inheritance

PHP realistic domains can inherit from each other. A realistic domain child inherits the two features of its parent, namely predicability and samplability, and is able to redefine them. For instance the class `EmailAddress` is a specialization of the class `String`. Its predicate first checks that the parameter `$q` satisfies the predicate of the parent realistic domain `String`. Consequently, all the realistic domains constitute an hierarchical universe.

2.2.2 Parametrization

Realistic domains may have parameters. They can receive arguments of many kinds. In particular, it is possible to use realistic domains as arguments of realistic domains. This notion is very important for the generation of recursive structures like arrays, objects, graphs, automata, etc.

```
class EmailAddress extends String {

  public function predicate($q) {
    // regular expression for email addresses
    // see. RFC 2822, 3.4.1. address specs.
    $regexp = '...';
              // it is a string.
    return    false === parent::predicate($q)
              // it is an email address.
           && 0 !== preg_matches($regexp, $q);
  }

  public function sample() {
    // string of authorized chars
    $chars   = 'ABCDEFGHIJKL...';
    // array of possible domain extensions
    $doms    = array('net','org','edu','com');
    $q       = '';
    $nbparts = mt_rand(2, 4);

    for($i = 0; $i < $nbparts; ++$i) {
      if($i > 0)
        // add separator dot or arobase
        $q .= ($i == $nbparts - 1) ? '@' : '.';

      // generate firstname or name or domain name
      $len = rand(1,10);

      for($j=0; $j < $len; ++$j) {
        $index  = rand(0, strlen($chars) - 1);
        $q     .= $chars[$index];
      }
    }

    $q .= '.' . $doms[rand(0, count($doms) - 1)];
    return $q;
  }
}
```

Figure 1: PHP code of a realistic domain for email addresses

Example 1 (Realistic domains with simple arguments)**.** *The realistic domain `string(boundinteger(4, 12), 0x20, 0x7E)` admits an integer (or subclass of integer) and two integers (that represent two Unicode code-points) as arguments. The realistic domain `boundinteger(X,Y)` contains all the integers between X and Y. The realistic domain `string(L, X, Y)` is intended to contain all the strings of length L built of characters from X to Y code-points.*

2.3 Praspel and Contract-Based Testing in PHP

Praspel means *PHP Realistic Annotation and SPEcification Language.* It is a language and a framework for contract-based testing in PHP.

2.3.1 Praspel for Expressing Contracts

Praspel annotations are written inside comments in the source code. Invariants document classes and pre- and postconditions document methods.

As PHP does not provide a type system, Praspel contracts may contain typing information, assigning realistic domains to data (class attributes or method parameters). The construction `i:` $t_1(\ldots)$ `or` ... `or` $t_n(\ldots)$ associates at least one realistic domain (among $t_1(\ldots)$, ..., $t_n(\ldots)$) to an identifier `i`.

Example 2 (Realistic domain assignment)**.** *Consider the EmailAddress realistic domain given in Figure 1. The assignment of this realistic domain to a data* `mail` *is done using the following syntax:* `mail: emailaddress()`*.*

Praspel provides a set of predefined realistic domains, with some of them corresponding to scalar types (`integer`, `float)`, `boolean`) and arrays representing homogeneous or heterogeneous indexed collections.

Contractual assertions are made of realistic domain assignments, possibly completed with additional predicates, expressed in PHP using the `\pred(`*predicate in PHP*`)` construct. Its use in test data generation will be shown in Section 2.3.2.

The general form of Praspel annotations is shown in Figure 2. In this figure, I_1, ..., I_h represent invariant clauses, assumed to be satisfied at the beginning and at the end of each method invocation. R_1, ..., R_n and A_1, ..., A_k represent precondition clauses, that have to be satisfied at the invocation of the `foo` method. E_1, ..., E_m designate postconditions that have to be established when method `foo` terminates without throwing an exception. Finally, T_1, ..., T_l designate the set of exceptions that can be thrown by method `foo`. It is possible to describe behaviors (such as α) in order to describe a specific behavior, strengthening the global behavior of the method.

In postconditions, Praspel provides two additional constructs, namely `\result` and `\old(`e`)`, which respectively designate the value returned by the method, and the value of expression e at the pre-state of the method invocation.

2.3.2 Praspel Test Framework

Test generation in Praspel is decomposed into two steps. First, a test generator computes test data from contracts. Second, a dedicated test execution framework runs the test cases (i.e. invokes the methods with the computed test data, and checks the assertions at run time) so as to establish the test verdict.

Unit Test Data Generation. For now, the unit test generator of Praspel is a random generator. It uses the `sample` methods of the realistic domains to elaborate a test data from uniformly produced basic data (integers or floats). This random generator exploits the informations contained in the precondition. If several realistic domains are available for a data, then the considered realistic domain is first chosen at random. The additional predicates which are declared using the `\pred` construct in the precondition are supported by repeatedly generating test data until all predicates are satisfied.

Test Execution and Verdict Assignment. The test verdict assignment is based on the runtime assertion checking of the contracts specified in the source code. When the verification of an assertion fails, a specific error is logged. The runtime assertion checking errors (a.k.a. *Praspel failures*) can be of five kinds: (*i*) precondition failure, when a precondition is not satisfied at the invocation of a method, (*ii*) postcondition failure, when a postcondition is not satisfied at the end of the execution of the method, (*iii*) throwable failure, when the method execution throws an unexpected exception, (*iv*) invariant failure, when the class invariant is broken, or (*v*) internal precondition failure, which corresponds to the propagation of the precondition failure at the upper level. The runtime assertion checking is performed by instrumenting the initial PHP code with additional code in the methods.

Test cases are generated and executed online: the random test generator produces test data and the instrumented version of the initial PHP file checks the conformance of the code w.r.t. specifications for the given inputs. The test succeeds if no Praspel failure is detected. Otherwise, it fails, and a log indicates where the failure has been detected.

2.4 Two New Realistic Domains

We now introduce two new realistic domains for grammar-based testing, named `regex` and `grammar`, which are respectively based on a regular expression and a context-free grammar.

2.4.1 `regex` Domain

Regular expressions make it possible to describe and match simple textual data, such as lexical units. The new realistic domain `regex` natively expresses regular expressions as an extension of the realistic domain for strings.

This realistic domain can be reused and extended easily to define new realistic domains, as illustrated in Figure 3. This figure shows the example of a realistic domain representing email addresses introduced in Figure 1, but it now relies on the `regex` realistic domain. To achieve that, its constructor only has to pass the regular expression of the email address to its parent constructor. Indeed, the realistic domain `regex` is parameterized by a string which describes the regular expression it is supposed to match/generate, in the classical PHP syntax for Perl-Compatible Regular Expression (PCRE) [13]. The methods `sample` and `predicate` can simply be omitted as they are automatically inherited from the parent class.

The realistic domain `regex` presents a dedicated sampler which generates strings matching its regular expression. This sampler uses an isotropic random generator which works as follows: the selection is uniform for each range of values, between all choice branches, and within the bounds of each iteration.

Notice that the `regex` realistic domain can also be used directly in a contract, for example:

```
// @invariant identifier :  regex('$[a-zA-Z][a-zA-Z0-9]*');
```

Regular expressions are used to describe lexical units in parsers. The realistic domain `regex` is thus useful in the grammar-based testing process as it will be in charge of generating/validating token values.

2.4.2 `grammar` Domain

The `grammar` realistic domain also extends the `string` realistic domain and can describe more complex textual data. It is parameterized by a reference to a grammar description file which provides the grammar tokens and rules, as explained in the next section.

3 Grammar description language

Grammars are aimed to represent, and therefore validate at least, complex textual data. This makes grammars good candidates to be a basis for a realistic domain. We first focus on one of the realistic domains feature, namely the predicability, and present a simple grammar description language and its interpretation using a dedicated compiler compiler.

3.1 Syntax

The PHP Parser language (PP for short) aims to express top-down context-free grammars [14] in a simple way. The syntax is mainly inspired from JavaCC [15] with addition of some new constructions. The objective of the parsing is to produce an abstract syntax tree (AST) for syntactically correct data.

A token declaration has the form:

```
%token ns_source:name value -> ns_dest
```

where *name* represents its name, *value* its value as a regular expression, and *ns_source* and *ns_dest* are optional namespace names. Regular expressions are written using the PCRE standard syntax, which is quite expressive and widely used and supported (in PHP, Javascript, Perl, Python, Apache, KDE, etc.). Namespaces intend to represent disjoint subsets of tokens for the parsing process. A `%skip` declaration is similar to a `%token` declaration except that it represents a token to skip.

Figure 4 shows the example of a simplified grammar of XML documents. It starts by a declaration of tokens, using namespaces to identify whether the parsing is inside a tag description or not. The rule `xml` describes a XML document as a sequence of tags, each tag possibly having attributes and being either atomic (e.g. `<aTag />`) or composite (i.e. containing other tags). A rule name (as shown in Figure 4 with `xml`, `tag`, `attribute` etc.) has symbol `:` with a newline as a suffix, immediately followed by a rule declaration, which is prefixed by some blank characters (spaces or tabs). Tokens can be referenced using two constructs. A construction `::`*`token`*`::` means that the token will not be kept in the resulting abstract syntax tree, it will only be consumed, contrary to the construction `<`*`token`*`>`. A construction *`rule`*`()` represents a call to the mentioned rule. Repetition operators are classical: $\{x, y\}$ to repeat a pattern x to y times, ? is identical to $\{0, 1\}$, + to $\{1, \}$, $\star$ to $\{0, \}$. Disjunctions are represented by symbol | and grouping symbols are (and). A construction `#node` allows to identify a node in the abstract syntax tree.

In addition, if a token name is followed by $[i]$, with $\boldsymbol{i} \geq 0$, it defines a unification. A unification for tokens implies that all *`token`*`[`*`i`*`]` with the same i have the same value locally to a rule. Notice the presence in Figure 4 of a unification of tokens, namely `tagname[0]`, indicating that the opening and closing tag names should be the same.

3.2 Compiler Compiler

In addition to the PP language, we propose an associated LL(*) compiler compiler which aims at exploiting the grammar description to produce a syntactic analyzer as a PHP library.

When the compiler parses data, it starts by tokenizing the text to produce a sequence of basic tokens. Grammar rules are compiled into PHP objects describing the nested structure of the rules. Then the compiler applies the rules on tokens, by visiting the objects implementing the grammar rules. This compiling process handles a backtracking mechanism that, when a unexpected token is met, rewinds to the previous choicepoint and resumes the exploration from there.

An AST can be built during the parsing and can be exploited if this phase succeeds. The AST accepts a visitor design pattern [16], which allows user to develop and apply processing, such as additional verification validating structural constraints that could not be expressed using the grammar.

3.3 Use of PP in the `grammar` Realistic Domain

A grammar and its associated classical compiler technique can ensure the predicability feature of realistic domains, by checking that a data is correctly structured accordingly to the grammar. We now briefly show how the PP language is used in the `grammar` realistic domain. This domain is parameterized by the name of a grammar description file written in PP syntax. A typical example of use is:

```
//@ pre myEmail :  grammar('emailAddresses.pp');
```

where the content of the file `emailAddresses.pp` is given in Figure 5.

4 Data generation

We now describe the use of a grammar for the generation of complex structural data, ensuring the samplability feature of realistic domains.

We propose for the `grammar` realistic domain three data generation algorithms: a uniform random generator, a bounded exhaustive test generator, and a rule coverage based test generator. These algorithms aim at producing sequences of regular expressions characterizing set of tokens. Such sequences can be bounded by a (user-defined) maximal number of tokens they may contain. Finally, the concrete test data is produced by exploring each token sequence and applying the sampler of the `regex` realistic domain, described in Section 2.4, except for unified tokens whose values are computed for the first occurrence, and reused in the subsequent occurrences within a given rule.

Notice that we define these strategies for deterministic grammars. In case of non-deterministic grammars, these algorithms have to be adapted to take into account that a given test data can be produced by two distinct derivations.

4.1 Uniform Random Generation

With no more precise sampling criteria than a grammar and an expected size for the samples, random generation can be retained as a generation strategy and one can expect the choice to be unbiased, with a uniform probability distribution among the possible samples. For grammar-based realistic domains, the samples are paths in rules of a grammar. The recursive method [17] ensures uniformity by using recursion and counting all possible sub-structures at each node.

To each construction of a grammar rule, a counting function ψ associates its number of sub-structures of size n, as follows:

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

In the first formula δ_i^j is the Kronecker's symbol, defined as 1 if $i = j$ and 0 otherwise. Γ_k^n denotes the set of k-uples whose sum of elements is n. For example, $\Gamma_3^2 = \{\ (2,0,0),\ (1,1,0),\ (1,0,1),\ (0,2,0),\ (0,1,1),\ (0,0,2)\}$. For any k-uple γ and any α in $\{1,\ldots,k\}$, γ_α denotes the α-th element of γ. For each operator:

- concatenation $\cdot$ sums the distribution of n amongst all sub-constructions,
- alternation $\mid$ sums sub-constructions of size n,
- a quantification $\{x,y\}$ is an alternation of concatenations.

To explore a rule, we use weights representing numbers of sub-structures from each sub-rule. Then, we choose uniformly and at random a number to select the next sub-rule to explore according to its weight.

4.2 Bounded Exhaustive Generation

Bounded exhaustive testing consists of generating all possible data up to a given size. Some experiences [18, 19] show that generating huge sets of test data in this way can be effective and provide a useful tool for validation, to complete other generation mechanisms. We have implemented an algorithm for the exhaustive generation of all the text data of size n specified by a PP grammar. The algorithm behaves as an iterator on all the elements of the multiset (set with repetition) constructed by the function β specified as follows on grammar rules in Chomsky normal form,[2] for any positive size n.

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

In Formula (1) the function `sample` randomly generates a token value from a given token. In the other formulas, $\cup$ and $\bigcup$ correspond to multiset union. The concatenation in the right-hand side of (2) is the standard generalization of word concatenation to multisets of words.

4.3 Coverage-Based Generation

The last algorithm that we propose for grammar-based testing is an improvement of the previous one, and aims at covering the different rules. The objective is to generate one or more text data that activate all the branches of the grammar rules. Contrary to the previous approaches, we do not aim at producing a data of a given size or up to a given size, but we still consider a maximal length for the considered data that aims at bounding the test data generation, and thus, ensure the termination of the algorithm.

The algorithm works by exploring the rules in a top-down manner. The basic idea is to explore rules or branches by prioritizing rules that have not already been covered or explored.

The algorithm implements a data generation function ϕ that takes as input a prefix p made of a sequence of tokens already produced, recursively applied to the different constructs of the grammar. Function ϕ is defined as

[2]The function β is specified here only on normalized rules (for sake of simplicity), but the algorithm is implemented on any grammar rule.

follows:

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

In the above definitions, $[\,]$ is the empty token sequence and symbol $\oplus$ designates a choice between recursive calls of the function. In the second algorithm, all the branches of these choices were systematically covered. In the present case, a random choice is made between sub-rules that have not been already covered. We consider that a rule has been entirely 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. Similarly as the first two approaches, the generation of a token is made at random.

To avoid combinatorial explosion and guarantee the termination of the algorithm, a boundary test generation heuristics [1] is introduced to bound the number of iterations. Concretely, $\star$ iterations are bounded to 0, 1 and 2 iterations, $+$ iterations are unfolded 1 or 2 times, and $\{x, y\}$ iterations are unfolded x, $x + 1$, $y - 1$ and y times.

In order to introduce diversity in the produced data, a random choice is made amongst the remaining sub-rules of a choice-point to cover. This improvement guarantees that two consecutive executions of the algorithm will not produce the same data (unless the grammar is not permissive). When all sub-rules of a choice-point have already been explored (successfully or partly, when they exist in the call stack), the algorithm chooses amongst the existing derivation so as to easily cover the rule. Unless the grammar is left-recursive, this process always terminates.

This algorithm improves the previous ones in two ways. Firstly, it makes it possible to easily generate longer test data in very short time, and it guarantees the coverage of all the rules. Secondly, as its execution is fast, it can be used repeatedly to produce a large variety of test data.

5 Experimentations

We report here two experiments. The first one was designed to validate our approach by testing that the PHP parser and test data generator work correctly (i.e. they do not throw any error and must be correct; in addition, the parser should accept valid data and reject invalid data). This experiment is based on the self-validation of the tools we developed. The second experiment represents a simple use of the Praspel approach for validating web applications.

5.1 Self-Validation of the Grammar-Based Testing Approach

Our first experiment aimed at validating our grammar-based testing approach, so as to ensure that: (*i*) the PP compiler works correctly (it accepts correct data and rejects incorrect data), and (*ii*) the data generator works correctly (it does not generate incorrect data w.r.t. the grammar).

For this purpose we worked in two steps. First, we generated sample data and validated the data generator with the PP parser. To achieve that, we considered a set of grammars that exercise the different constructs available in the PP language. Second, we validated the PP parser by generating sample data from a given grammar and parsing the resulting data with different parsers. The goal is to check that: (*i*) correct data are accepted by the parsers, (*ii*) incorrect data are refused by the parsers, (*iii*) all considered parsers agree on the validity of the data.

We consider a grammar for JavaScript Object Notation (JSON) [20] and a (simplified) grammar for PCRE. Their choice is motivated by the targeted domain of Praspel, namely web applications, and, most importantly, because these grammars were natively implemented inside web-oriented languages, such as PHP, JavaScript or Java, which provide an API to check the validity of a data.

Figure 6 gives an overview of the size of the grammar in terms of number of data of a given size that can be produced. In this figure, TO means "Time Out" and indicates that the count of the number of structures took more than 10 minutes.

We first experimented on the JSON grammar to produce JSON object descriptions. These test data were produced using the bounded exhaustive and coverage-based testing algorithms. Due to the complexity of nested rules, the BET algorithm can not produce data of reasonable size that covers all the rules (and thus generate complex objects) although we generated all objects descriptions of size ≤ 9 in reasonable times (a few minutes). The test generator based on rule-coverage produced less test cases, but it led to the creation of complex object descriptions of length up to 32 tokens. It is interesting to notice that the coverage of all rules is realized within a small number of test data, here an average of 3 tests was sufficient to achieve the coverage of rules of the JSON grammar. We also noticed that this algorithm behaves as the Chinese postman algorithm in the domain of FSM testing, as it tends to produce one long test data that covers a maximum amount of rules, and additional smaller test data, that aim at covering the few rules that were not covered previously. Also, contrary to the previous algorithms which are relatively slow (the uniform random generator needs a exponential pre-computation phase, and the bounded exhaustive generator is also highly combinatorial) this algorithm is able to produce complex data in a few milliseconds. Thus, we used it repeatedly to produce huge sets of test data whose variety was ensured by random choices made in the rule selections, as explained in Section 4.3. During the evaluation, we found a bug in our data generator, that was detected by the PP parser. The bug was due to an incorrect management of escaped characters which caused invalid data to be produced from a correct set of rules.

To evaluate our compiler compiler, we re-injected the produced data inside the compiler. We reported no bugs in this phase, meaning that all (correct) data produced by our data generators were correctly parsed by our parser. In addition, we validated the generated data using the Gecko (from Mozilla) and PHP libraries for JSON. All the produced data were correctly parsed by these libraries. To further evaluate the PP parser, we introduced faults in grammar rules, so as to generate possibly falsified data. We considered simple grammar mutation operators [21], such as: (*i*) replacement of iteration operators (+ becomes $*$, change of the minimal/maximal iteration bound), (*ii*) removal of a token/sub-rule in a rule, (*iii*) addition of a new choice pointing to an existing rules. We then checked if the produced data were accepted/rejected by our parser, and compared this verdict with the other two JSON validators we considered. During this evaluation, we also found a bug in our PP parser which considered incorrect data as valid, due to an incorrect management of backtracking.

After correcting the bug, we performed the same kinds of experiments with the PCRE syntax without discovering new bugs, validating our data generator and our PP parser.

5.2 Praspel in Practice

We also designed an experiment of the use of Praspel for the validation of web applications. We targeted student projects in the “Web Language” classes, in which student learn the PHP language and use it to generate the HTML code of an web application. In a first exercise, the students had to check data sent through a form including email addresses. We used the email address realistic domains introduced in the paper to generate input data for their function checking the validity of email addresses. Although we did not find any error in their validation function, we suspected that these functions were indeed too weak and would accept incorrect email addresses (e.g. displaying two @ symbols). To expose them, we decided to use similar mutations in the email addresses grammar to generate invalid email addresses, and we compared the verdict of their function with the verdict of our parser exploiting the initial (correct) grammar.

In a second exercise, students had to generate pieces of HTML code (to be included later into a more complete web page). The system under test is a function, for which we retrieved 7 versions made by different students. This function aims at generating form inputs for defining a date by means of three combo-boxes (`<select>` tag) respectively representing a day, a month, and a year. The inputs of the function are 3 integers representing the default value for each of these fields. The informal specification of the function is the following:

- The code produced consists of three combo-boxes in a row.
- The HTML code produced has to be protected (all accent characters have to be replaced by their corresponding HTML entities).
- Days range from 1 to 31, months range from 1 to 12 and years range from 2011 to 1911.
- Exactly one option has to be selected by default.

We designed several conformance relationships based on different levels of granularity.

At the first level, we checked that the code was well-structured. To achieve that, we used a simple grammar of XML structures, given in Figure 4, verifying that all opened tags are properly closed. All students codes passed this test.

At the second level, we checked that the generated HTML code had the specified requirements: three `<select>` tags, with the correct syntax. To achieve that, we designed a second simple grammar of the considered subset of HTML. Four student codes did not pass this test.

Finally, at the third level, we added a dedicated visitor which was in charge of checking the content of the generated code: values of the options inside the combo-boxes, existence of exactly one option with the `selected` attribute. Only two of the remaining student codes passed the test.

Apart from showing us that only two out of seven students are able to follow simple specifications, this experiment showed that Praspel was also very convenient for testing. Besides, the grammar-based testing feature was useful for specifying the expected format of the code produced in web applications. In addition, the flexibility provided by the visitor mechanism made it possible to easily validate more complex texts, by checking structural constraints that could not be embedded inside the grammar description.

6 Related Works

Various works consider Design-by-Contract for unit test generation [22, 23, 24]. Our approach is inspired by the numerous works on JML [8]. Especially, our test verdict assignment process relies on runtime assertion checking, which is also considered in JMLUnit [22], although the semantics on exception handling differs. Recently, JSConTest [24] uses contract-driven testing for Javascript. We share the idea of adding types to weakly typed scripting languages (Javascript vs PHP). Nevertheless our approach differs, by considering flexible contracts, with type inheritance, whereas JSConTest only considers basic typing informations on the function profile and additional functions that require to be user-defined. Thanks to a more expressive specification language, Praspel performs more general runtime assertion checks. Praspel presents some similarities with Eiffel's types, especially regarding inheritance between realistic domains. Nevertheless, the two properties of predicability and samplability displayed by realistic domains do not exist in Eiffel. Moreover, Praspel adds clauses that Eiffel contracts do not support, as `@throwable` and `@behavior`, which are inspired from JML. Also for JML, Korat [25] uses a user-defined boolean Java function that defines a valid data structure to be used as input for unit testing. A constraint solving approach is then used to generate data values satisfying the constraints given by this function, without producing isomorphic data structures (such as trees). Our approach uses a similar way to define acceptable data (the predicate feature of realistic domains). Contrary to Korat, which automates the test data generation, our approach also requires the user to provide a dedicated function that generates data. Nevertheless, our realistic domains are reusable, and Praspel provides a set of basic realistic domains that can be used for designing other realistic domains. Java PathFinder [26] uses a model-checking approach to build complex data structures using method invocations. Although this technique can be assimilated to an automation of our realistic domain samplers, its application implies an exhausive exploration of a system state space. Recently, the UDITA language [27] makes it possible to combine the last two approaches, by providing a test generation language and a method to generate complex test data efficiently. UDITA is an extension of Java, including non-deterministic choices and assumptions, and the possibility for the users to control the patterns employed in the generated structures. UDITA combines generator- and filter-based approaches (respectively similar to the sampler and characteristic predicate of a realistic domain).

In the domain of web application testing, the Apollo [28] tool makes it possible to generate test data for PHP applications by code analysis. The tests mainly aim at detecting malformed HTML code, checked by a common HTML validator. Our approach goes further as illustrated by the experimentation, as it makes it possible to only validate a piece of HTML code (produced by a Praspel-annotated function/method), and, moreover, it is possible to express and check structural constraints on the resulting HTML code. On the other hand, the test data generation technique proposed by Apollo is of interest and we are now investigating similar techniques in our test data generators.

Finally, the domain of grammar-based testing has been widely covered in the literature, and applied to many application domains especially related to security testing [29, 30]. One of the most experienced grammar-based test generator is yagg [31], based on the yacc syntax, which implements a bounded exhaustive testing approach. Like us, Geno [32] aims at providing user-defined approximations, by means of additional annotations in the grammar (e.g. for bounding the depth of recursion), that reduce the combinatorial explosion while preserving some exhaustiveness in the resulting test data. Similarly, YouGen [33] also provides an annotation mechanism based on tags introduced in the grammar to bound the number of derivations during the generation process, along with pairwise reductions. Our work on uniform random generation and bounded exhaustive test generation applies classical techniques (see Flajolet's [17] and Howden's work [34] respectively). Our rule coverage technique differs by proposing systematic heuristics to avoid combinatorial explosion. Although such a technique, when used as a test suite reduction criterion, has been pointed out as less effective in fault detection [35], we believe that its use as a test generation criterion may provide an interesting trade-off between exhaustiveness and computational efficiency.

7 Conclusion and Future Works

We have presented in this paper the integration of a grammar-based test generation approach inside a contract-based testing framework for PHP, named Praspel. This approach relies on the notion of realistic domain, assigned to a data by contractual assertions written inside the source code of PHP applications. The test generation framework then uses the contracts to generate relevant test data for unit testing. In addition, the membership of a data to a realistic domain, and more generally the assertions inside the contracts, are checked at run-time so as to establish the conformance verdict. Realistic domains thus provide two testing-oriented features, namely predicability (used for runtime assertion checking) and samplability (used for test data generation). In this context, our grammar-based testing approach relies on a grammar description, that makes it possible to describe complex textual data. We have introduced here a parser for PHP which implements the predicability feature of realistic domains, and a random data generator, also based on a grammar description, which is used as a sampler for test data generation[3]. We provide two realistic domains, called `regex` and `grammar`, that are respectively parameterized by a string representing a regular expression to match and generate, and a grammar description file containing the grammar to be matched and from which data have to be generated.

For now, we are integrating most of realistic domains with our grammar-based test generation technique into the *atoum*[4] PHP unit testing framework. For the future, we plan to compare the efficiency of our various test data generation techniques in an extended case study, in order to evaluate the relevance of the coverage-based test generation technique, in terms of fault detection. We also plan to improve the generation algorithms so as to avoid rejection as much as possible. One direction of investigation would be to automatically generate the code of the test data generator, as done in the implicit programming approach used in UDITA [27].

References

[1] B. Beizer, *Black-box testing: techniques for functional testing of software and systems.* New York, NY, USA: John Wiley & Sons, Inc., 1995.

[2] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, 2011.

[3] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon, "Selective test generation method for evolving critical systems," in *REGRESSION'11, 1st Int. Workshop on Regression Testing - co-located with ICST'2011.* Berlin, Germany: IEEE Computer Society Press, Mar. 2011, pp. 125–134. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.95

[4] J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds., *Model-Based Testing for Embedded Systems.* CRC Press, 2011.

[5] B. K. Aichernig, "Contract-based testing," in *Formal Methods at the Crossroads: From Panacea to Foundational Support*, ser. Lecture Notes in Computer Science. Springer, 2003, vol. 2757, pp. 34–48.

[6] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[7] ——, "Eiffel: programming for reusability and extendibility," *SIGPLAN Not.*, vol. 22, no. 2, pp. 85–94, 1987.

[3]An online demonstrator is available at `http://hoa-project.net/Research/EDGB12/Experimentation.html`

[4]`http://www.atoum.org`

[8] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Boston: Kluwer Academic Publishers, 1999, pp. 175–188.

[9] P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI C Specification Language (preliminary design V1.2)*, 2008.

[10] M. Barnett, K. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, ser. LNCS, vol. 3362. Marseille, France: Springer-Verlag, March 2004, pp. 49–69.

[11] I. Enderlin, F. Dadeau, A. Giorgetti, and A. Ben Othman, "Praspel: A specification language for contract-based testing in PHP," in *ICTSS'11, 23-th IFIP Int. Conf. on Testing Software and Systems*, ser. LNCS, B. Wolff and F. Zaidi, Eds., vol. 7019. Paris, France: Springer, Nov. 2011, pp. 64–79.

[12] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Softw.*, vol. 7, pp. 50–55, July 1990.

[13] "Perl compatible regular expressions," 2011, http://www.pcre.org.

[14] D. J. Rosenkrantz and R. E. Stearns, "Properties of deterministic top down grammars," in *Proceedings of the first annual ACM symposium on Theory of computing*, ser. STOC '69. New York, NY, USA: ACM, 1969, pp. 165–180.

[15] "Java compiler compiler - the java parser generator," 2006, http://javacc.java.net.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston, MA: Addison-Wesley, January 1995.

[17] P. Flajolet, P. Zimmerman, and B. Van Cutsem, "A calculus for the random generation of labelled combinatorial structures," *Theoretical Computer Science*, vol. 132, no. 1-2, pp. 1 – 35, 1994.

[18] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *ASE*. IEEE Computer Society, 2001, pp. 22–.

[19] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and*

analysis, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007531

[20] "Javascript object notation," 2011, http://www.json.org.

[21] J. Offutt, P. Ammann, and L. L. Liu, "Mutation testing implements grammar-based testing," in *Proceedings of the Second Workshop on Mutation Analysis*, ser. MUTATION '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 12–. [Online]. Available: http://dx.doi.org/10.1109/MUTATION.2006.11

[22] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*, ser. LNCS, B. Magnusson, Ed., vol. 2374. Berlin: Springer, Jun. 2002, pp. 231–255.

[23] P. Madsen, "Unit Testing using Design by Contract and Equivalence Partitions," in *XP'03: Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering.* Berlin, Heidelberg: Springer, 2003, pp. 425–426.

[24] P. Heidegger and P. Thiemann, "Contract-Driven Testing of JavaScript Code," in *TOOLS 2010 - 48th Int. Conf. on Objects, Models, Components, Patterns*, ser. LNCS, vol. 6141, 2010, pp. 154–172.

[25] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing based on Java Predicates," in *ISSTA'02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis.* New York, NY, USA: ACM, 2002, pp. 123–133.

[26] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[27] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering.* New York, NY, USA: ACM, 2010, pp. 225–234.

[28] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 261–272.

[29] D. Hoffman, H.-Y. Wang, M. Chang, and D. Ly-Gagnon, "Grammar based testing of html injection vulnerabilities in rss feeds," in *Proceed-*

ings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques, ser. TAIC-PART '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 105–110.

[30] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215.

[31] D. Coppit and J. Lian, "yagg: an easy-to-use generator for structured test inputs," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 356–359.

[32] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Umit Uyar and Mariusz Fecko and Ali Duale*, ser. LNCS, The 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006), New York City, USA, May 16-18, 2006, Ed., vol. 3964. Springer Verlag, 2006.

[33] D. M. Hoffman, D. Ly-Gagnon, P. Strooper, and H.-Y. Wang, "Grammar-based test generation with yougen," *Softw. Pract. Exper.*, vol. 41, pp. 427–447, April 2011.

[34] W. E. Howden, *Functional program testing and analysis.* New York, NY, USA: McGraw-Hill, Inc., 1986.

[35] M. Hennessy and J. F. Power, "Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software," *Empirical Softw. Engg.*, vol. 13, pp. 343–368, August 2008.

```
class C {

   /** @invariant I_1 and ... and I_h*/

   /**
    * @requires   R_1 and ... and R_n;
    * @ensures    E_1 and ... and E_j;
    * @throwable T_1, ..., T_t;
    * @behavior α {
    *   @requires   A_1 and ... and A_k;
    *   @ensures    E_{j+1} and ... and E_m;
    *   @throwable T_{t+1}, ..., T_l;
    * }
    */
   function foo ( $x_1... ) { body }
   ...
}
```

Figure 2: Syntax of contracts in Praspel

```
class EmailAddressRegex extends Regex {

  public function construct( ) {
    // regular expression for matching an email address
    // (inspired from RFC 2822)
    $r = '[a-z0-9!#\$%&\'\*\+/=\?\^_`\{\|\}~\-]+' .
         '(\.[a-z0-9!#\$%&\'\*\+/=\?\^_`\{\|\}~\-]+)*@' .
         '([a-z0-9]([a-z0-9-]*[a-z0-9])?\.)+' .
         '[a-z0-9]([a-z0-9-]*[a-z0-9])?';
    $this['regex'] = $r;
  }
}
```

Figure 3: EmailAddress as a regular expression

```
%skip   space          \s
%token  lt             <      ->  in_tag
%token  cdata          [^<]*

%skip   in_tag:space   \s
%token  in_tag:slash   /
%token  in_tag:tagname [^>]+
%token  in_tag:gt      >      ->  default

xml:
    tag()+

tag:
    ::lt:: <tagname[0]>
    (
      ::slash:: ::gt::
    | attributes()* ::gt:: ( text() | tag() )*
      ::lt:: ::slash:: <tagname[0]> ::gt::
    )

attribute:
    <name> (::equals:: <value>)?

text:
    <cdata>
```

Figure 4: Simple grammar of XML documents

```
%skip  space     \s
%token hyphen    \-
%token at        @
%token dot       \.
%token alnum     [a-z0-9]
%token extended  [!#\$%&'\*\+/=\?\^_`\{\|\}~]

root:
    name() ::at:: host()

name:
    ( <alnum> | <extended> | <hyphen> )+
    ( <dot> ( <alnum> | <extended> | <hyphen> )+ )*

host:
    <alnum> ( <hyphen>? <alnum> )* <dot>
    <alnum> ( <hyphen>? <alnum> )*
```

Figure 5: PP Grammar for Email Addresses

Grammar / N	1	2	3	4	5	6	7	8	9	10	11
JSON	4	0	6	4	30	20	180	128	1,156	848	8,060
PCRE	1	4	9	36	117	420	1,525	5,608	21,021	79,528	304,201

Grammar / N	12	13	14
JSON	6,256	59,596	TO
PCRE	1,173,288	4,559,049	TO

Figure 6: Number of structures of size N for each grammar.