Praspel: Contract-Driven Testing for PHP using Realistic Domains

Ivan Enderlin, Fabrice Bouquet, Frédéric Dadeau, Alain Giorgetti

Praspel: Contract-Driven Testing for PHP using Realistic Domains

Ivan Enderlin, Fabrice Bouquet, Frédéric Dadeau, Alain Giorgetti

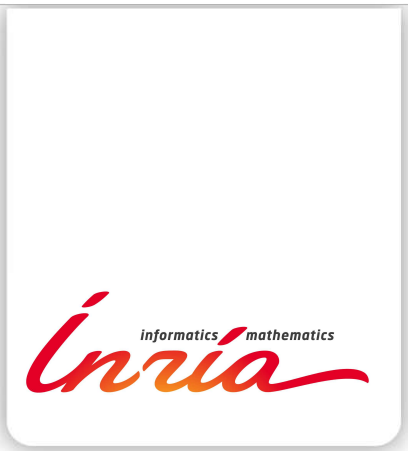
RESEARCH REPORT

N° 8592

September 2014

Project-Team Cassis

ISSN 0249-6399 ISRN INRIA/RR--8592--FR+ENG



Praspel: Contract-Driven Testing for PHP using Realistic Domains

Ivan Enderlin*, Fabrice Bouquet*, Frédéric Dadeau*, Alain Giorgetti*

Project-Team Cassis

Research Report n° 8592 — September 2014 — 36 pages

Abstract: We present an integrated contract-based testing framework for PHP. It relies on a behavioral interface specification language called Praspel, for *PHP Realistic Annotation and Specification Language*. Using Praspel developers can easily annotate their PHP scripts with formal contracts, namely class invariants, and method pre- and postconditions. These contracts describe assertions either by predicates or by assigning *realistic domains* to data. Realistic domains introduce types in PHP and describe complex structures frequently encountered in applications, such as email addresses or SQL queries. Realistic domains display two properties: *predicability*, which allows to check if a data belongs to a given realistic domain, and *samplability*, which allows to generate valid data.
This paper introduces coverage criteria dedicated to contracts, designed to exhibit relevant behaviors of the annotated methods. Test data are then computed to satisfy these coverage criteria, by using dedicated data generators for complex realistic domains, such as arrays or strings. This framework has been implemented and disseminated within the PHP community, which gave us feedback on their usage of the tool and the relevance of this integrated process with respect to their practice of manual testing.
This work is supported by FUI Squash project (contract F1010025 Z).

Key-words: Contract-Driven Testing, Realistic Domains, Praspel, PHP, Coverage Criteria, Grammar-Based Testing, Array Solver.

* E-mail: `firstName.lastName@femto-st.fr`

RESEARCH CENTRE NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Praspel: Contract-Driven Testing for PHP using Realistic Domains

Résumé : Nous présentons un environnement intégré de test à base de contrats pour PHP. Les contrats sont exprimés dans un nouveau langage de spécification formelle appelé Praspel, pour *PHP Realistic Annotation and Specification Language*. Avec Praspel, les développeurs peuvent facilement annoter leurs scripts PHP avec des contrats formels, composés d'invariants de classe et de pré- et post-conditions de méthodes. Ces contrats décrivent des propriétés comportementales, soit par des prédicats, soit en associant des *domaines réalistes* à des données. Les domaines réalistes introduisent des types en PHP et décrivent des structures de données complexes fréquemment rencontrées dans les applications, telles que des adresses électroniques ou des requêtes SQL. Les domaines réalistes ont deux propriétés : la *prédicabilité*, qui permet de valider qu'une donnée appartient au domaine, et la *générabilité*, qui permet de générer une valeur dans le domaine.

Ce document définit aussi des critères de couverture dédiés aux contrats, conçus pour distinguer certains comportements des méthodes annotées. Des données de test sont calculées pour satisfaire ces critères de couverture, en utilisant des générateurs de données dédiés, pour des domaines réalistes complexes, comme les tableaux ou les chaînes de caractères. Cet environnement de test a été implanté et diffusé au sein de la communauté PHP, qui nous a donné des retours d'expérience sur leur utilisation de l'outil et sur la pertinence de ce processus intégré, par rapport à la pratique usuelle du test manuel.

Ce travail est financé par le projet FUI Squash, contrat F1010025 Z.

Mots-clés : Contract-driven testing, domaines réalistes, Praspel, PHP, critères de couverture, grammar-based testing, solveur de tableaux.

1 Introduction

Model-Based Testing (MBT for short) [6] is a technique in which a model of the System Under Test (SUT) is employed in order to validate it using tests. 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 compared. MBT makes it possible to automate the test generation, and partly, the test execution phase. It is implemented in many tools, based on various modelling languages or notations [40].

Model-Based Testing is thus a convenient reason to promote formal methods among developers. However, its application is often restricted to critical and/or embedded software that require a strong level of validation [43]. 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 model design is a complex task that requires modelling skills and understanding of formal methods, thus requiring skilled validation 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, and effort, of test concretization is 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 on which it operates). Therefore, Model-Based Testing is currently not applied at a large scale.

Annotation languages provide an interesting solution to address these issues. Their underlying paradigm consists in inserting the model as annotations (i.e. comments) in the code, expressing formal assertions about it, namely: (*i*) *invariants*, which are properties that should hold at each execution step, and (*ii*) *pre-* and *postconditions*, which are conditions that respectively have to hold for an operation/method to be invoked and after its execution. One can consider that these assertions establish a *contract* between code developers and their users. Therefore, *behavioral interface specification languages* for these annotations are also called *contract languages*. This *Design by Contract* (DbC for short) [37] was first introduced by B. Meyer with Eiffel [36]. Various contract languages extend popular programming languages, such as JML for Java [31], ACSL for C [5], and Spec# for C# [4]. Annotation-based Design by Contract is thus a way to introduce lightweight formal methods in the process of development of an application, and the syntactic proximity between the code and its model makes its writing easier to learn and to practice by developers, even if they are not familiar with formal methods.

One of the main interests, and uses, of contracts is for testing. Indeed, Contract-Based Testing (CBT for short) [1] has been introduced to exploit annotation languages for testing programs. CBT aims to address the following issues of "classical" MBT approaches, especially for unit testing: (*i*) The information contained in invariants and preconditions can be used to generate test data easily. (*ii*) Postconditions provide a (partial) test oracle to perform assertion checking at run-time: The test succeeds if no assertion is violated, and fails otherwise. (*iii*) Due the proximity between the code and the model, there is no abstraction gap to bridge, contrary to other MBT approaches, especially in testing, when it comes to the concretization of the abstract test cases.

However, we identified some issues that motivate our work. First, the current state of the art related to Contract-Based Testing mainly consists in using the contracts to filter out irrelevant test cases, and their dominating usage is to establish the test verdict based on the violation of the assertions. Thus, the contracts are poorly used during the test generation phase. We believe that contracts could be better used for test generation purposes. Secondly, in the case of dynamically typed languages, automated test generation remains a challenge, as data are never formally typed. Thus, it is a hard task to automatically infer their types to produce relevant test

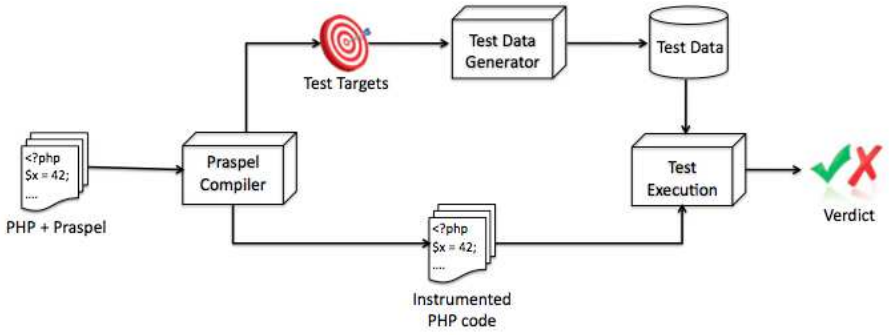


Figure 1: Process of the Praspel Contract-Based Testing Framework

data. In particular, the well-established language PHP, widely used to design web applications, lacks tool support for test generation. It has surprisingly been very little targeted by the scientific community, while it presents a widespread community of users who really need assistance for knowing if their programs work correctly or not.

The motivations of our approach can be summarized by the following three research questions:

RQ1: To what extend is it possible to use the annotations written in contracts for generating model-based tests?

RQ2: To what extend is it possible to automate the test generation for dynamically-typed languages, such as PHP?

RQ3: To what extend is it useful for developers to dispose of an integrated contract-based testing framework?

We summarize in this paper an original Contract-Based Testing approach, applied to the PHP programming language, and supported by a complete and integrated tool-set, made available to the PHP community.

In a previous work, we have introduced Praspel, a tool-supported specification language for Contract-Based Testing in PHP [14]. Praspel extends contracts with the notion of *realistic domain*, which makes it possible to assign a domain of values to data, namely class attributes or method parameters. Realistic domains present two useful features for testing: *predicability*, which is the possibility to check whether a data belongs to its associated domain, and *samplability*, which is the possibility to automatically generate a data from a realistic domain. A standard library of predefined realistic domains (mainly scalar domains, strings, arrays) is already available along with an integrated test environment.

The overall process of contract driven testing with Praspel is depicted in Figure 1. Contracts written in Praspel specify the behaviors of methods. These contracts are used by a test generator that computes equivalence classes based on the structure of the contracts to define the test targets (RQ1). Dedicated test data generators, especially for structured data, notably strings and arrays, are then used to automatically compute data values for a given test target (RQ2). The generated tests can then automatically be run on the system under test, namely the PHP program, and the test verdict is automatically established by checking the contract assertions at runtime, using the PHP code instrumented by the verifications of the assertions (pre/postconditions and invariants). This approach is fully integrated into a dedicated framework, called Hoa [13] (RQ3).

The first contribution of this paper is the definition of contract coverage criteria that make it possible to produce test targets, illustrating relevant behaviors formalized in the contracts.

These coverage criteria can be applied at different levels, namely: (*i*) at a structural level, to exhibit various configurations given by the clauses of the contract, (*ii*) at the predicate level, to exhibit various configurations that are nested inside contract predicates, and (*iii*) at the realistic domain level, to combine between the assignments of realistic domains to a given datum.

The second contribution is an experimental evaluation of the proposed approach, involving actors of the PHP community (professional web developers) who tried out our framework and compared its usage and efficiency with their current state of the practice.

This paper is organized as follows. Section 2 explains the notion of realistic domain and presents its implementation in PHP. Then, Section 3 introduces the syntax and semantics of contracts in Praspel, to specify class invariants and method behaviors. Section 4 presents how to use the method contracts to extract test targets, namely test data that aim at satisfying contract coverage criteria that we also define. We then address in Section 5 the problem of test data generation from these test targets which focuses on improving the random generation in presence of strings and arrays. Experimental results are then reported in Section 6. Section 7 presents the related work. Finally, Section 8 concludes and presents future extensions to this work.

2 Realistic domains

One of the most spreaded way to caracterize data is the use of types. Unfortunatly, the majority of interpreted languages, of whom PHP, have no type. More precisely, PHP has a dynamic type system. This means that types are not declared syntactically but deduced at runtime. Moreover, PHP has a weak type system. This means that it is really easy to cast a data from one type to another. Thus, a software frequently manipulates many types for a datum, which is casted at runtime to satisfy the operations. For instance, the addition of the string `'1.2'` to the integer `4` is equal to the float `5.2`.

To fill the gap of the absence of information about data, we propose realistic domains. Roughly speaking, they refine usual types such as integers, strings, arrays, etc., and more complex types. The word "realistic" means they are intended to specify domains of relevant data for a specific context. For example, an email address can be a realistic domain: many software identify their users by their email address. It is more than a string, as it matches a specific syntax.

Realistic domains have been designed in order to be used in a testing context. That is why they have two properties:

- their *predicability* allows to verify that a data belongs to the set of values described by the realistic domain,
- their *samplability* allows to generate a value described by the realistic domain.

2.1 Implementation

In our PHP implementation, a realistic domain is represented by a class, and the properties of predicability and samplability by methods, respectively:

- `predicate($q)`, where `$q` is the value to validate,
- `sample($sampler)`, where `$sampler` is a numerical sampler: it generates only integers and floats, and allows to guide the generation of data; for example we will use this numerical sampler to pick a character within a given range of characters. For now, the sole numerical sampler available is pseudo-random.

```
class Boundinteger extends Integer /* ① */ {

    protected $_arguments = array(
        'Constinteger lower' => PHP_INT_MIN, /* ⑤ */
        'Constinteger upper' => PHP_INT_MAX  /* ⑥ */
    );

    public function predicate ( $q ) {

        return    parent::predicate($q)                    /* ② */
               && $q >= $this['lower']->getConstantValue()  /* ③ */
               && $q <= $this['upper']->getConstantValue(); /* ④ */
    }

    public function sample ( $sampler ) {

        return $sampler->getInteger(
            $this['lower']->sample($sampler),
            $this['upper']->sample($sampler)
        );
    }
}
```

Figure 2: Light implementation of the `Boundinteger` realistic domain. We can see the inheritance, the properties `predicate` and `sample`, the refinement and the parameters declaration.

2.2 Hierarchy

Since realistic domains are implemented as classes and PHP is object-oriented, we have inheritance. Consequently, a child realistic domain can inherit and refine the properties of its parent. For example, `Color` extends `String`, and `Boundinteger` extends `Integer`. All realistic domains must be children of the `Realdom` class.

Figure 2 shows a light implementation of the `Boundinteger` realistic domain. It extends the `Integer` realistic domain in ①. The `predicate` property refines the one of its parent in ②, and adds new constraints in ③ and ④.

2.3 Parameters

Similarly to functions, realistic domains are parameterizable. The data received by a realistic domain are called its arguments. Parameters are very useful when we want to represent complex structures, such as nested arrays, graphs, automata, trees, etc.

For example, the realistic domain `string(0x61, 0x7a, boundinteger(4, 12))` represents a string whose size is between 4 and 12 and whose character code-points are between `0x61` and `0x7a`. We can write `4..12`, which is a syntactic sugar.

When we describe the parameters of a realistic domain, we can indicate their type. This type is expressed through the name of one or many realistic domains. Indeed, a realistic domain can accept many sorts of arguments, and is able to make the translation from one type to another by itself. We stay close to the spirit of PHP. For example, for the realistic domain `String`, its two first parameters can be integers or characters. Then, we are allow to write `string('a', 'z', 4..12)` without any error, but `string(true, 'z', 4..12)` will throw an error.

The Figure 2 shows the declaration of two parameters: `lower` in ⑤, and `upper` in ⑥.

2.4 Classification

Inheritance is one classification. A transversal one is the use of interfaces. They help to characterize the "landscape" of realistic domains. The most interesting interfaces are:

- `Constant`, to represent an immutable realistic domain with only one value, such as `42`, `true`, etc.,
- `Interval`, to represent an interval by a lower and an upper bound, which can be reduce dynamically (at runtime),
- `Finite`, to represent a realistic domain with a countable number of values,
- `Nonconvex`, to represent a realistic domain with holes, i.e. where values have been excluded, and thus, can no longer be generated,
- `Enumerable`, to represent a realistic domain that can be enumerated.

Thus, a realistic domain that implements the `Interval` and `Nonconvex` interfaces represents an interval with holes.

3 Praspel

Praspel means *PHP Realistic Annotation and SPEcification Language*. It is a language and a framework for contract-based testing in PHP. It relies on realistic domains. Contract-Based Testing is explained in Section 4.

Praspel is an *annotation language* because it is written in the source code of the software, more precisely in PHP comments. PHP has three sorts of comments: inline (with `//` or `#`), multi-lines (between `/*` and `*/`) and block (between `/**` and `*/`). Praspel will be located in block comments, since it is culturally the dedicated type of comment for annotations, API documentation etc.

Praspel is a Behavioral Interface Specification Language (BISL) based on contracts. A *contract* is a model of the code behavior, described with formal constraints, called clauses, such as preconditions, postconditions and invariants. These constraints are generally located in the source code around the data. In the case of Praspel, invariants are located on class attributes, and pre- and postconditions are located on method parameters. The semantics of a contract is as follows:

- the callee of a method commits to satisfy the precondition,
- only in this case, the called method commits to establish its postcondition,
- and invariants must be satisfied before and after the method execution.

The rest of this section describes parts of the grammar of Praspel, in normal form.

3.1 Clauses

A contract is composed of clauses, whose syntax is defined in Figures 3, 4, 6 and 7. In these figures, the style `token` expresses a Praspel token, *rule* expresses a syntactic entity in the grammar, and *php-token* expresses a PHP token (to be as close as possible to the language manipulated by the developer). The notation e^r_s means that the pattern e is repeated r times, and separated by s. r can be ?, + or $*$, respectively for 0 or 1 time, 1 or more times and 0 or more times. If s is not empty, then s must be a token.

```
specification      ::=  attribute-clauses | method-clauses
attribute-clauses  ::=  invariant-clause?
method-clauses     ::=  ( description-clause ; )?
                        rbet-clauses
rbet-clauses       ::=  ( requires-clause ; )?
                        (
                            ( behavior-clause+ default-clause? )?
                          | ( ensures-clause ; )? ( throwable-clause ; )?
                        )
```

Figure 3: Praspel grammar in normal form: Top rules.

The rules for *attribute-clauses* and *method-clauses* in Figure 3 define the syntax of the clauses respectively annotating a class and a method. The components of these clauses are detailed in Figure 4. All of them introduce Praspel expressions, detailed in Section 3.2. The syntactic entity *method-clauses* shows the order of method clauses in normal form. The actual Praspel grammar accepts clauses in any order.

Among method clauses, a *description-clause* introduces an informal comment about the method. The other method clauses are formal ones. The syntactic entities *invariant-clause*, *behavior-clause* and *default-clause* are explained through an example in the following paragraphs. The keyword `@requires` introduces a precondition. The keywords `@ensures` and `@throwable` introduce postconditions. A precondition expresses constraints on the pre-state of the system, whereas a postcondition expresses constraints on its post-state. In order to change the state of the system, it has to be executed by calling the method under the contract. The keyword `@ensures` introduces a normal postcondition, that should hold when the system terminates without throwing an exception. The clause `@throwable` T expresses the exceptions that can be thrown and constraints on the exceptional post-state reached when the system throws an exception. The syntax of T is defined by the syntactic entity *exceptional-expression* in Figure 6. T is a disjunction of expressions of the form T_C `with` T_E, where T_C is a list of PHP classnames associated to an identifier, and T_E is an expression, called an *exceptional postcondition*, whose syntax is detailed in Section 3.2. All identifiers declared in T_C can appear in T_E. The semantics is as

```
invariant-clause    ::=  @invariant expression
requires-clause     ::=  @requires expression
behavior-clause     ::=  @behavior php-identifier {
                           ( description-clause ; )?
                           rbet-clauses
                         }
default-clause      ::=  @default {
                           ( description-clause ; )?
                           ( ensures-clause ; )?
                           ( throwable-clause ; )?
                         }
ensures-clause      ::=  @ensures expression
throwable-clause    ::=  @throwable exceptional-expression
description-clause  ::=  @description php-string
```

Figure 4: Praspel grammar in normal form: Clause rules.

$$
\begin{array}{rcl}
\textit{expression} & ::= & (\textit{declaration}^{+}_{\texttt{and}}\ \texttt{and})^{?}\ (\textit{constraint}^{+}_{\texttt{and}}\ \texttt{and})^{?}\ \textit{predicate}^{?}_{\texttt{and}} \\
\textit{exceptional-expression} & ::= & (\ (\ \textit{exception-identifier}\ )^{+}_{\texttt{or}}\ \texttt{with}\ \textit{expression}\ )^{+}_{\texttt{or}} \\
\textit{exception-identifier} & ::= & \underline{\textit{php-classname}}\ \underline{\textit{php-identifier}}
\end{array}
$$

Figure 6: Praspel grammar in normal form: Expression rules.

follows: If the thrown exception is an instance of a class of exceptions in the list T_C, then it is assigned to the associated identifier and the exceptional postcondition T_E has to hold.

Figure 5 shows the typical form of a specification of a class `C` with an attribute `a` and a method `f`. The `@invariant` clause is located just before the class attribute `a`. Invariants are checked before and after the execution of the system. The other clauses are method clauses. They are located just before the method header. A method can have different behaviors according to its arguments and the state of the system. Praspel proposes the keyword `@behavior` to introduce a behavior, identified by a name, and possibly containing an informal description (in a *description-clause*), both useful for giving a feedback to the user. A behavior is defined by a precondition (`@requires` keyword) and either a sequence of normal and exceptional postconditions (`@ensures` and `@throwable` keywords), or a sequence of nested behaviors, again introduced by the keyword `@behavior`, as illustrated in Figure 5 where the behavior β is nested inside the behavior α. We can also describe alternative behaviors by juxtaposing `@behavior` clauses. This is illustrated in Figure 5 with the behavior γ, sibling of the behavior α. Behaviors are, in practice, mutually exclusives, however the syntax allows to describe non-deterministic behaviors. The `@default` clause is strictly equivalent to a `@behavior` clause, with an implicit `@requires` clause describing the conjunction of all the negations of previous sibling `@behavior` clauses.

In order to "activate" a given behavior, its precondition and the ones of all its ancestors have to be satisfied. If the post-state after execution of the system is normal, then the normal postconditions of all the activated behaviors have to hold. If the post-state is exceptional, then the exceptional postconditions of all the activated behaviors have to hold, as defined above.

```
class C {

    /**
     * @invariant I;
     */
    protected $a;

    /**
     * @requires R;
     * @behavior α {
     *     @requires  R_α;
     *     @behavior β {
     *         @requires  R_α.β;
     *         @ensures   E_α.β;
     *         @throwable T_α.β;
     *     }
     * }
     * @behavior γ {
     *     @requires  R_γ;
     *     ...
     * }
     * @default {
     *     @ensures   E_D;
     *     @throwable T_D;
     * }
     */
    public function f ( ) { }
}
```

Figure 5: A Praspel contract with all clauses

3.2 Expressions

Figures 6 and 7 describe Praspel expressions. There are mainly three kinds of expressions: declarations, predicates and constraints, respectively presented in the following paragraphs.

Declarations. A declaration assigns one or many realistic domains to a variable through the operator `:`. A variable is either an attribute, specified in an `@invariant` clause, or a method

declaration ::= `let`$^{?}$ *extended-identifier* `:` *disjunction*
constraint ::= *qualification* | *contains*
qualification ::= *identifier* `is` *php-identifier*$^{+}_{,}$
contains ::= *extended-identifier* `contains` *constant*$^{+}_{\texttt{or}}$
predicate ::= `\pred(` *php-string* `)`
disjunction ::= `(` *constant* | *realdom* | *extended-identifier* `)`$^{+}_{\texttt{or}}$
realdom ::= *php-identifier* `(` *argument*$^{?}_{,}$ `)`
argument ::= `default` | *realdom* | *constant* | *array*
| *extended-identifier*
constant ::= *scalar* | *array*
scalar ::= `null` | *php-boolean* | *number* | *php-string*
number ::= *php-binary* | *php-octal* | *php-hexa* | *php-decimal*
array ::= `[` *pair*$^{?}_{,}$ `]`
pair ::= `from`$^{?}$ *disjunction* `to` *disjunction*
| `to`$^{?}$ *disjunction*
extended-identifier ::= *array-access*
array-access ::= *identifier* `(` `[` *scalar* `]` `)`$^{?}$
identifier ::= *php-identifier*
| `this` `(` `->` *php-identifier* `)`*
| `(` `self` | `static` | `parent` `)` `(` `::` *php-identifier* `)`$^{+}$
| `\old(` *extended-identifier* `)`
| `\result`

Figure 7: Praspel grammar in normal form: Expression construction rules.

argument, specified in other clauses. The value of a variable can belong to different realistic domains if the variable is defined by a *disjunction* of realistic domains, represented by the `or` keyword. Figure 8 shows a short contract containing only a precondition (`@requires`) and a postcondition (`@ensures`). The precondition declares that the domain of the variable `seed` is either an interval from 7 to 42 or an interval from 153 to 256, and that the variable `factor` should be a float greater than or equal to 1.0. A disjunction of realistic domains can contain any kinds of realistic domains: It fits the "dynamic" side of PHP.

If a variable is preceded by the `let` keyword, it means that it is a local variable, which belongs to the model and not to the system. This is helpful when manipulating intermediate data representations.

The `@ensures` clause has one particularity: It can contain the special variable `\result` representing the value returned by the method. The `@ensures` and `@throwable` clauses can refer to the value of variables in the pre-state, thanks to the `\old`(i) construction, where i is the name of a variable.

Predicates. Variables are preferably associated to a realistic domain, because automated validation is based on their predicability and samplability features. But these features are sometimes hard to implement. These variables can however be constrained with the `\pred`(p) construction, where p is PHP code. It allows the user to express arbitrary constraints using PHP itself instead of Praspel. The code p must be a predicate in disjunctive normal form (DNF). For instance, the predicate `\pred('isfemtotime() or isarch(64)')` in Figure 8 means that if the system supports femto-time (a variant of micro-time) or if its architecture supports 64 bits, then this part of the precondition is valid.

Since the `\pred`(p) construction is like a black-box, it introduces a rejection of generated data. In fact, when we have data, we validate them by the predicates. If one predicate is too strong, it invalidates data, and we generate new ones. Such a situation can lead into a infinite-like loop. Fortunately, there is a maximum number of tries (which is parameterizable). When this number is reached, the generator gives up. Then, a recurrent task is to look what developers write the most in this construction and extract it into Praspel in order to be validated and verified. The goal is to move them into the language itself instead of being in a black-box, and thefore, reduce the introduced rejection as much as possible. This processus is illustrated in Section 5.1.

Constraints. Praspel has mainly two kinds of constraints: Either by using the declaration syntax, or by using the `is` keyword. Examples of constraints are presented in Section 5.1.

```
/**
 * @requires seed:   7..42 or 153..256 and
 *           factor: 1.0.. and
 *           \pred('isfemtotime() or isarch(64)');
 * @ensures  \result: array([to float()], 1..5);
 */
public function generate ( $seed, $factor ) { ... }
```

Figure 8: Example of a short contract.

3.3 Array description

Array descriptions are an important part of Praspel, controlling the generation of arrays for testing, presented in Section 5.1.

In PHP, an array is always an *associative array* (or *map*, or *dictionary*), i.e. a collection of key-value pairs, where each key appears at most once. Keys can be null, booleans, integers, floats or strings. PHP accepts these kinds of keys but booleans are casted to an integer and floats are reduced to their integer parts. Values can be of many kinds. An array can be *homogeneous* or *heterogeneous*. In an homogeneous array, all the keys have the same type, and all the values too. In an heterogeneous array, keys may have distinct types, and/or values may have distinct types. Keys can be auto-incremented, by adding 1 to the last integer index starting by 0. The *length* (or *size*) of an array is its number of elements. An array has no predefined length, but its length (stored internally by the PHP engine) can be retrieved thanks to the PHP function `count()`. An array has also no predefined depth, i.e. it can contain arbitrary arrays.

In Praspel, `array(`D`,`L`)` denotes the realistic domain of arrays whose domains and codomains are described by D and whose length is in the disjunction L of realistic domains of non-negative integers. D is a comma-separated list, between `[` and `]`, of *array descriptions* of the form `from` K `to` V, where K and V are realistic domain disjunctions, respectively for keys and values. When the `from` keyword is missing, it is transformed into a realistic domain representing an auto-incremented integer starting at 0 with a step of 1.

Example 1 (Homogeneous and heterogeneous arrays)**.** *The syntax of array descriptions is illustrated by the following array declarations:*

```
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)
```

The identifier `a1` *is declared as a homogeneous array of booleans with a length between 7 and 42. The identifier* `a2` *is declared as a homogeneous array of length 7, whose keys are integers between 0 and 5 or simply 10, and whose values are integers. The identifiers* `a3` *and* `a4` *are declared as heterogeneous arrays. Both arrays can contain the pairs* (`5`, `true`) *and* (`15`, `4.2`)*, but* `a4` *can contain the pair* (`5`, `4.2`)*, whereas* `a3` *can not contain it.*

Actually, we introduce a *normal form* that removes disjunctions in array descriptions (in `from` ... `to` ... constructs), by applying iteratively the rewriting rule (`from` F_1 `or` F_2 `to` T_1 `or` T_2 $\rightarrow$ `from` F_1 `to` T_1, `from` F_1 `to` T_2, `from` F_2 `to` T_1, `from` F_2 `to` T_2). An array description is in normal form when it can not be reduced further by this rule.

Example 2 (Array description in normal form)**.** *The following declaration of* `a4` *is in normal form:*

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

4 Contract-coverage criteria

This section presents contract-based testing for Praspel. Since a contract most of the time expresses several behaviors, it deserves several tests. We propose various criteria of contract coverage, to qualify a set of tests w.r.t. a contract. After associating a graph to each Praspel contract in Section 4.1, we define in Section 4.2 contract-coverage criteria based on the activation of paths in these graphs.

4.1 Contract graph

In this section, we explain how to construct a graph from a Praspel contract, in normal form. This graph reflects the contract clauses through realistic domains assignments and predicates.

In all that follows, we denote by A the set composed of all the Praspel expressions generated by the syntactic entities *expression* and *exception-identifier* of Praspel grammar (see Figure 6), completed by all the expressions of the form $\neg t_1 \wedge \ldots \wedge \neg t_n$ where t_i is generated by the syntactic entity *exception-identifier*, for $1 \leq i \leq n$. We consider two families of graphs, namely *expression graphs* and *contract graphs*, defined as follows.

Definition 1. *An* expression graph *is a tuple* (V, D, i, n) *where* V *is the finite set of* vertices*,* $D \subseteq V \times A \times V$ *is the finite set of* arcs *from* V *to* V *annotated by an expression in* A*,* $i \in V$ *is the* initial vertex*,* $n \in V$ *is the* normal final vertex*.*

Definition 2. *A* contract graph *is a tuple* (V, D, i, N, E, U) *where* V *is the finite set of* vertices*,* $D \subseteq V \times A \times V$ *is the finite set of* arcs *from* V *to* V *annotated by an expression in* A*,* $i \in V$ *is the* initial vertex*,* N *is the set of* normal final vertices*,* E *is the set of* exceptional final vertices *and* U *is the set of* trap vertices*, with* $N \uplus E \uplus U \subseteq V$*.*

Section 4.1.1 transforms any Praspel expression into an expression graph. The other sections define the transformation of any Praspel contract into a contract graph. The simplest case is the one of behavior-free contracts, called *atomic contracts*. They are considered in Sections 4.1.2 and 4.1.3. Then, contract graphs with behaviors are defined by induction. The basic case with only one behavior is treated in Section 4.1.4. The induction step considering one more `@behavior` clause is treated in Section 4.1.5. The case of a contract with a `@default` clause (resp. `@invariant` clause) is considered in Section 4.1.6 (resp. 4.1.7). For the sake of simplicity, we only consider contracts in normal form.

4.1.1 Graph of an expression

Praspel expressions are described in Figure 6 by the syntactic entity *expression*. An expression is a conjunction of declarations, predicates and constraints, respectively described by the syntactic entities *declaration*, *predicate* and *constraint* in Figure 7. We only transform expressions of the first two kinds into an expression graph, since expressions whose kind is *constraint* are not relevant for coverage: they represent only one behavior (for example *constraint* and *declaration* always appear together) and executed.

An expression without constraints in normal form is of the form d `and` p, where d (resp. p) is a conjunction of expressions of kind *declaration* (resp. *predicate*). Its expression graph is the concatenation by an ε transition of the *domain graph* obtained from d and the *predicate graph* obtained from p. Both constructions are detailed below.

A conjunction of declarations has the general form i_1`:` D_1 `and` ... `and` i_m`:` D_m where $D_j = d_{j,1}$ `or` ... `or` d_{j,k_j} is a disjunction of realistic domains ($1 \leq j \leq m$). We also denote by D_j the list $[d_{j,1}, \ldots, d_{j,k_j}]$ of realistic domains in the disjunction. Let $S = [i_1, \ldots, i_m]$ be the list

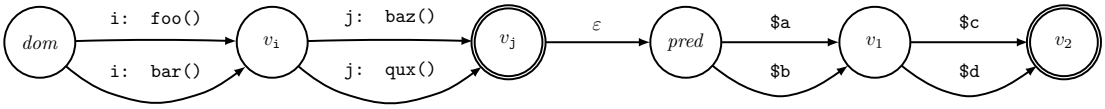


Figure 9: An expression graph, composed of a domain graph and a predicate graph.

of identifiers declared in this expression. Let last(S) denote the last element in the list S. For instance, for the expression:

```
i: foo() or bar() and j: baz() or qux()
```

we have $S = [\mathtt{i}, \mathtt{j}]$, $D_1 = [\mathtt{foo()}, \mathtt{bar()}]$ and $D_2 = [\mathtt{baz()}, \mathtt{qux()}]$.

The *domain graph* associated to the expression i_1`:` D_1 `and` ... `and` i_m`:` D_m is $G = (\{dom\} \cup \{v_s \mid s \in S\}, D, dom, v_{\text{last}(S)})$ where

$$D = \bigcup_{d \in D_1} \{(dom, i_1\texttt{:}\quad d, v_{i_1})\} \cup \bigcup_{2 \leq j \leq m} \bigcup_{d \in D_j} \{(v_{i_{j-1}}, i_j\texttt{:}\quad d, v_{i_j})\}.$$

A *predicate* is of the form `\pred(`p`)`, where p is a string that contains a PHP predicate in disjunctive normal form (DNF). We associate to it a *predicate graph*. The definition is similar to the one of the domain graph, after replacing `&&` by `and` and `||` by `or` (and labelling the disjunctions to get numbered vertices). The initial state is no more *dom* but *pred*. For example, `\pred('($a || $b) && ($c || $d)')` becomes (1 :) `$a or $b and` (2 :) `$c or $d`.

Figure 9 graphically presents the graph associated to the former examples.

4.1.2 Graph of an atomic contract

An *atomic contract* is a contract only composed of the `@requires`, `@ensures` and `@throwable` clauses. It contains neither `@behavior` clauses nor the `@default` clause, i.e. no sub-contract, and is therefore not decomposable into smaller contracts. An atomic contract in normal form takes the general form:

```
@requires  R;
@ensures   E;
@throwable T_C with T_E;
```

where R and E are *expressions* respectively associated to the `@requires` and `@ensures` clauses. The *exceptional-expression* associated to the `@throwable` clause is composed of two parts: T_C represents the classname of the exception and the identifier that holds the exception, and T_E represents the exceptional postcondition, which is also an *expression*. The contract graph associated to this atomic contract is

$$G = (\{v_1, v_2, v_3, v_4, v_5, v_6\}, D, v_1, \{v_3\}, \{v_5\}, \{v_6\})$$

where

$$D = \{(v_1, R, v_2), (v_2, E, v_3), (v_2, T_C, v_4), (v_4, T_E, v_5), (v_2, \neg T_C, v_6)\}.$$

Figure 10 graphically presents this graph. In this figure and all the following ones in this section, normal final vertices are represented by a double circle and exceptional final vertices by a double rectangle.

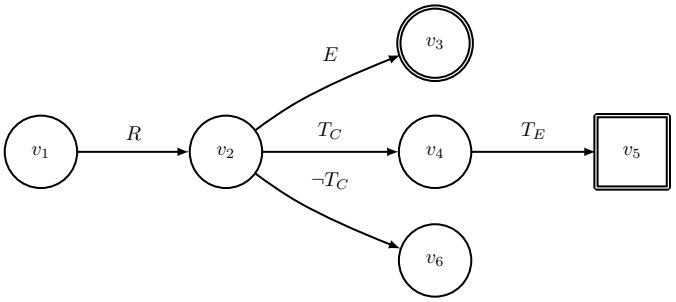


Figure 10: Graph of an atomic contract with one `@throwable` clause.

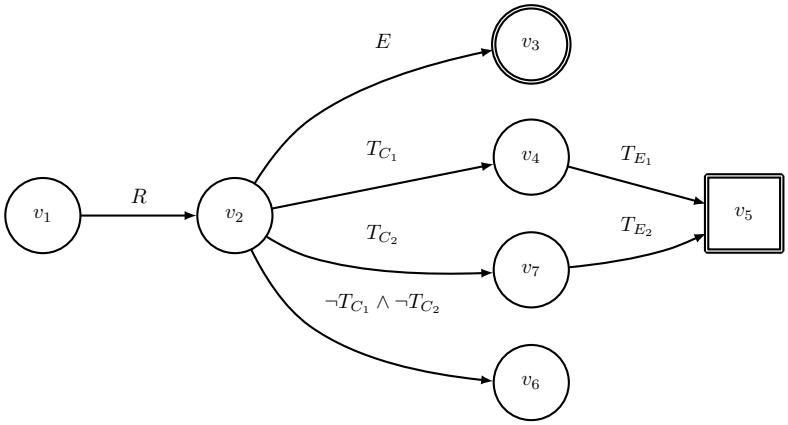


Figure 11: Graph of an atomic contract with two `@throwable` clauses.

4.1.3 Extension of the graph of an atomic contract with one more `@throwable` clause

Section 4.1.2 considered a contract with only one `@throwable` clause. In this section, we see how a contract graph is modified when adding one more clause `@throwable` T_C `with` T_E to a contract whose graph is $G = (V, D, v_1, N, E, U)$. By induction, we can assume that the set of vertices V contains at least the six vertices $v_1, v_2, v_3, v_4, v_5, v_6$ introduced in Section 4.1.2.

The resulting graph is $G' = (V \cup \{v'\}, D', v_1, N, E, U)$ for some new vertex $v' \notin V$, with

$$D' = D \cup \{(v_2, T_C, v'), (v', T_E, v_5)\} \\ - \{(v_2, \varphi, v_6) \mid (v_2, \varphi, v_6) \in D\} \cup \{(v_2, \varphi \wedge \neg T_C, v_6) \mid (v_2, \varphi, v_6) \in D\}.$$

Figure 11 graphically presents this graph when there are only two `@throwable` clauses (in the figure, φ is $\neg T_{C_1}$ and v' and T_C are respectively denoted v_7 and T_{C_2}).

4.1.4 Graph for a contract with one `@behavior` clause

We now consider the case of a contract C containing exactly one `@behavior` clause. As presented in Section 3.1, a contract with a behavior has no external `@ensures` and `@throwable` clauses. So, the most general form of the contract C is:

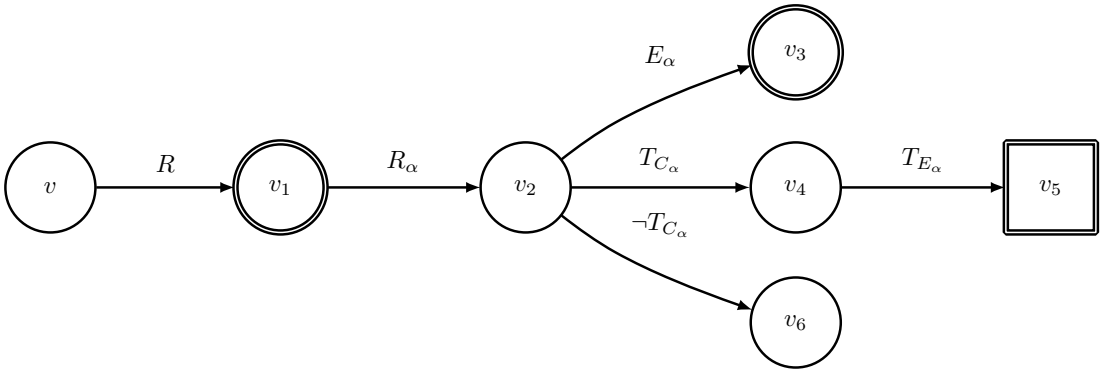


Figure 12: Graph of a contract with one atomic behavior.

```
@requires  R;
@behavior  α { B }
```

where B is the contract of the behavior α. Let $G_\alpha = (V_\alpha, D_\alpha, i_\alpha, N_\alpha, E_\alpha, U_\alpha)$ be the graph of B. Then, the contract graph associated to the contract C is

$$G = (\{v\} \cup V_\alpha, \{(v, R, i_\alpha)\} \cup D_\alpha, v, \{i_\alpha\} \cup N_\alpha, E_\alpha, U_\alpha),$$

for some new vertex $v \notin V_\alpha$. This graph is composed of all the vertices and arcs from G_α, completed with one new arc.

Figure 12 graphically presents this graph when the behavior only contains an atomic subcontract with one `@throwable` clause (in the figure, i_α is denoted v_1).

4.1.5 Extension of the graph of a contract with one more `@behavior` clause

Let us now see how to add one more `@behavior` β clause to a list of behaviors whose contract graph is (V, D, i, N, E, U). Let $G_\beta = (V_\beta, D_\beta, i_\beta, N_\beta, E_\beta, U_\beta)$ be the contract graph associated to the behavior β. Modulo some renaming, assume that V contains a vertex v such that D contains some arc (i, R, v), and that all the vertices of G_β are distinct from the ones of G (i.e., $V \cap V_\beta = \emptyset$). Then, the new contract graph is $G = (V \cup V_\beta - \{i_\beta\}, D \cup D'_\beta, i, N \cup N'_\beta, E \cup E_\beta, U \cup U_\beta)$, where D'_β (resp. N'_β) is obtained from D_β (resp. N_β) by replacing i_β with v.

4.1.6 Case of the `@default` clause

After a group of `@behavior` clauses, we can have a `@default` clause. Remember that a `@default` clause is strictly equivalent to a `@behavior` clause with an implicit `@requires` clause (see Section 3.1). Thus, adding a `@default` clause is a special case of adding a `@behavior` clause, considered in Section 4.1.5.

4.1.7 Case of an invariant clause

In this last section, we see how to integrate a clause `@invariant` I (located on class attributes) in the contract graph of a method whose graph is $G_M = (V_M, D_M, i_M, N_M, E_M, U_M)$. We assume that the set V_M does not contain the vertices v_1, v_2, and v_3. The resulting graph is

$$G = (\{v_1, v_2, v_3\} \cup V_M, D, v_1, \{v_2\}, \{v_3\}, U_M)$$

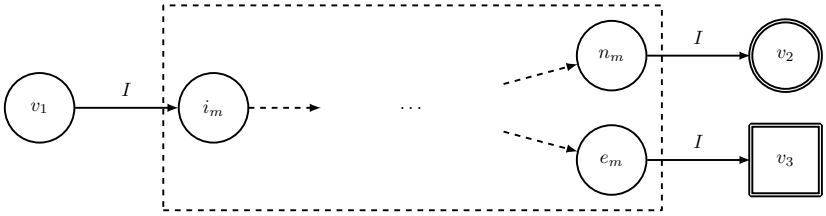


Figure 13: Graph of a contract with an invariant I.

with

$$D = \{(v_1, I, i_M)\} \cup \{(n, I, v_2) \mid n \in N_M\} \cup \{(e, I, v_3) \mid e \in E_M\}.$$

It contains three additional vertices and several new arcs, whose meaning is that the invariant I should hold before the method invocation and after it, in all the cases of normal and exceptional termination.

Figure 13 illustrates this principle of extension of a contract graph with an invariant. For the sake of simplicity, it represents only one normal final vertex and one exceptional final vertex.

We note that all the constructed graphs are acyclic.

4.2 Coverage criteria

We now define contract-coverage criteria as properties of sets of paths in the graph associated to a given PHP method specified with Praspel. Before that, we recall the definitions of path, test case and test suite in the present context.

Throughout this section, we consider a given PHP method `f` and denote by $G(\texttt{f})$ or G its contract graph (V, D, i, N, E, U), as defined in Section 4.1.

4.2.1 Definitions

A *path* is a (dot-separated) sequence of consecutive transitions (two transitions (v, α, w) and (x, β, y) are *consecutive* iff $w = x$). We say that a vertex v *is in* a path p if the path p contains some transition (v, α, w) or (w, α, v).

We associate two kinds of paths to the contract graph $G = (V, D, i, N, E, U)$. A *road* of G is a path whose transitions are in D. Now, consider any road r of G and replace any transition (v, e, w) in r, where e is an expression, with the path $(v, \varepsilon, i_H) \cdot q \cdot (n_H, \varepsilon, w)$, where q is a path from i_H to n_H of the expression graph $H = (V_H, D_H, i_H, n_H)$ associated to e. The result is called the *trail* of r. A *trail* of the contract graph G is any trail obtained so from a road of G.

A (unit) *test* for a PHP method `f` is a set of data (aka inputs), i.e. a set of values for the arguments of `f`. A *test case* is a pair (s, t) composed of a state s (the context from which the test is executed) and a (unit) test t (namely, the method invocation with a given set of parameter values). To build state s, we have two approaches: (i) create an object and invoke its methods in order to change its state, or (ii) instrument the code to shunt encapsulation and set the state of the object directly. A *test suite* is a set of test cases.

When running a test case, we are able to compute a set of paths in the contract graph G. We denote by s' the state after the execution of the test, r the returned value of the invoked method, and e the raised exception. Notice that, if the method throws an exception, then r is undefined. On the opposite, if the method terminates normally (i.e., without throwing any exception), e is considered as undefined.

In order to compute the paths (roads or trails) of the contract that are activated by a given test (s, t), we explore the contract graph $G(\texttt{f})$ and collect the paths by evaluating the transition labels w.r.t. the test, as follows:

- If the transition is labelled by an expression R (resp. I) coming from a precondition (resp. invariant evaluated in the pre-state) the transition is activated if and only if R (resp. I) holds in the pre-state s of the test.
- If the transition is labelled by an expression E coming from a postcondition (normal or exceptional), the transition is activated if and only if E holds, when the `\result` is replaced by the returned value, the `\old` subexpressions are replaced by their value in the pre-state s, and the other subexpressions are evaluated in the post-state s'.
- If the transition is labelled by an expression I coming from an invariant evaluated in the post-state, the transition is activated if and only if I holds in the post-state s'.
- If the transition is labelled by a domain assignment i `:` d, the transition is activated if and only if the current value of i belongs to the realistic domain d (checked using the predicability feature of the realistic domain). When the domain assignment originates from a precondition (i.e. is in the expression graph of a precondition), the value of i is taken in the pre-state s of the test if i is a class attribute, or in the parameters if i is a parameter of method. When the domain assignment is expressed in a postcondition (normal or exceptional), the value of i is taken from the post-state s'.
- If the transition is labelled by a predicate, the transition is activated if and only if the predicate is satisfied by the current state of the system (s in the case of a precondition, s' for a before-after predicate contained in a postcondition in which `\old` expressions are evaluated in the pre-state s).
- If the transition is labelled by an exception class T_C, then the transition is activated if and only if the returned exception e is not undefined and is an instance of T_C.

From now on, we only consider paths of $G(\texttt{f})$ (roads or trails) starting from the initial state of $G(\texttt{f})$ and ending in one of its final states (either normal or exceptional). For example, in Figure 11, we see that it is possible to activate several paths at the same time, when T_{C_1} extends T_{C_2} and an exception of class T_{C_1} is thrown. We denote by $R_{\texttt{f}}(S)$ the set of roads of $G(\texttt{f})$ activated by a test suite S. Similarly, we denote by $T_{\texttt{f}}(S)$ the trails of $G(\texttt{f})$ activated by a test suite S, traversing the expression graphs nested in $G(\texttt{f})$.

We now define the contract coverage criteria that we propose, based on the contract graphs.

The following examples are based on the contract in Figure 14 for a method `compute` with two arguments: `server` which is a class of kind `\Irc\Server`, and `buffer` which contains an IRC message. We have two behaviors, one for a private or public message (characterized by a regular expression through the `regex` realistic domain) and one for a "ping". In this last behavior, the precondition also has a predicate to specify that the network is in a specific state. Finally, the default behavior specifies that an exception of kind `\Irc\Exception\MalformedMessage` must be thrown if the buffer is neither a message nor a ping. Moreover, the code of this exception must be an integer between 400 and 491.

4.2.2 Clause Criterion

The Clause Criterion aims at covering the structure of a contract. Thus, the Clause Criterion is satisfied by the test suite S of a method `f` if all the states $V \backslash U$ of the contract graph $G(\texttt{f}) = (V, D, i, N, E, U)$ of method `f` are in the roads of $R_{\texttt{f}}(S)$.

```
/**
 * @requires server: class('\Irc\Server');
 * @behavior message {
 *     @requires buffer: regex('/^privmessage .+/') or regex('/^message .+/');
 *     @ensures  \result: 1..;
 * }
 * @behavior ping {
 *     @requires buffer: regex('/^ping$/') and
 *               \pred('$server->bufferState >= 0 or
 *                      network_buffer_state() > 0');
 *     @ensures  \result: 1..;
 * }
 * @default {
 *     @throwable \Irc\Exception\MalformedMessage e with e->code: 400..491;
 * }
 */
public static function compute ( $server, $buffer ) { ... }
```

Figure 14: Example of a contract and a method that manipulates an IRC stream.

Example 3 (Test suite satisfying the Clause Criterion)**.** *The following test cases cover all the clauses. The syntax* `object(c)` *represents a valid instance of a class c. The test suite* $\{t_1, t_2, t_3\}$ *with*

(t_1) `compute(object(\Irc\Server), 'message foobar')`,

(t_2) `compute(object(\Irc\Server), 'ping')` *and*

(t_3) `compute(object(\Irc\Server), 'xyz')`

satisfies the Clause Coverage Criterion.

4.2.3 Domain Criterion

The Domain Criterion refines the Clause Criterion by considering the domain assignments inside the expression graphs. More formally, the Domain Criterion is satisfied by a test suite S for a method `f` if all the transitions of the form i: d in the contract graph $G(\texttt{f})$ (which are contained in expression graphs labelling transitions of $G(\texttt{f})$ and containing the *dom* node) appear in at least one trail in $T_{\texttt{f}}(S)$.

Example 4 (Test suite satisfying the Domain Criterion)**.** *The test suite* $\{t_1, t_2, t_4\}$ *with*

(t_4) `compute(object(\Irc\Server), 'privmessage foobar')`

satisfies the Domain Coverage Criterion: Both domain assignments of the variable `server` *are covered.*

4.2.4 Predicate Criterion

Similarly to the Domain Criterion, the Predicate Criterion applies to the predicates in the expression graphs, aiming at covering all of them. More formally, the Predicate Criterion is satisfied

by a test suite S for a method `f` if all the transitions not of the form i: d in the contract graph $G(\texttt{f})$ (which are contained in expression graphs labelling transitions of $G(\texttt{f})$ and containing the *pred* node) appear in at least one trail in $T_{\texttt{f}}(S)$.

Example 5 (Test suite satisfying the Predicate Criterion)**.** *The test suite* $\{t_5, t_6\}$ *with*

(t_5) `compute(object(\Irc\Server), 'ping')` *with* `\pred('$server->bufferState >= 0')` *and*

(t_6) `compute(object(\Irc\Server), 'ping')` *with* `\pred('network_buffer_state() > 0')`

satisfies the Predicate Coverage Criterion.

4.2.5 Combinations

Having such coverage criteria is not enough to produce or qualify *realistic* test cases. Indeed, covering all the clauses of a contract does not ensure that all the domains have been covered. On the contrary, knowing that all domains have been covered does not ensure that all the clauses of the contract have been exploited. Thus, in order to have more realistic test cases, these contract coverage criteria can be combined together. We propose the following combinations:

- the Clause + Domain (CD, for short) Criterion aims at covering the structure and domain data of a contract,
- the Clause + Predicate (CP, for short) Criterion aims at covering the structure and predicate data of a contract, and
- the Clause + Domain + Predicate (CDP, for short) Criterion aims at satisfying the Clause, Domain and Predicate criteria at the same time.

This lattice of contract coverage criteria is summarized in Figure 15.

Example 6 (Test suites satisfying combinations of contract coverage criteria)**.** *The test suite* $\{t_1, t_2, t_3, t_4\}$ *satisfies the Clause + Domain Coverage Criterion. The test suite* $\{t_1, t_3, t_5, t_6\}$ *satisfies the Clause + Predicate Coverage Criterion. The test suite* $\{t_1, t_3, t_4, t_5, t_6\}$ *satisfies the CDP Coverage Criterion.*

5 Test data generation

Praspel uses random generation to generate test data. This approach is efficient when manipulating integers. However, for other types, it can be laboreous. After several studies, we have found that the most used types in PHP are arrays and strings. The following sections address the problem of efficient arrays and strings generation by using respectively a dedicated array solver, and a grammar-based approach. These two sections summarize our previous work [16, 15], for the sake of being self-contained. Finally, we complete these descriptions by explaining, in a third section, how object data are generated.

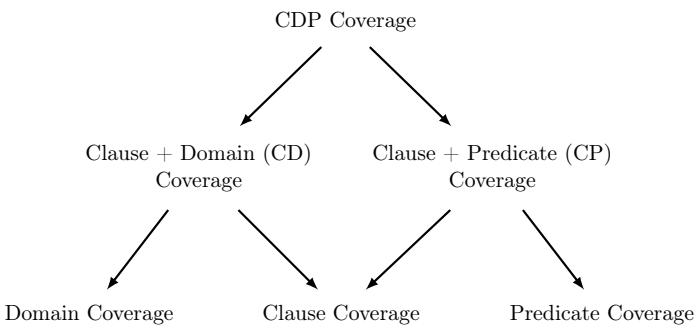


Figure 15: The hierarchy of criteria. $A \longrightarrow B$ means that if criterion A is satisfied criterion B is also satisfied.

5.1 Solver-based generation for arrays

In PHP, arrays cover most of the needs for collections, because they can store key-value pairs of any kind, they do not have a specific length or depth, and they are efficiently implemented. After exploring many popular PHP projects in search for the most used functions on arrays, we have extended Praspel with specifications of the corresponding conditions on arrays. Then, we have formalized their semantics by constraints, for which we have implemented a constraint solver in PHP, fully integrated into the test generation framework [15]. This section summarizes these array conditions, their semantics and describes the constraint solver.

5.1.1 Collecting information

In order to determine the most popular conditions on arrays, we have selected 61 PHP projects, from Github and SourceForge, for their popularity, impact on the industry and complexity. All these projects represent 28 066 files and 5 220 547 lines of code. In this code, we have counted the number of occurrences of each array function available in the PHP standard library. It appeared that the three most frequently used functions are `count()`, `array_key_exists()` and `in_array()`. The `count()` function counts the number of values in an array, the `array_key_exists()` function checks whether a key is present in an array (independently of its associated value, e.g. it returns true even if the value is null), and finally, the `in_array()` function checks whether a value is present in an array. All these functions work on one array at a time. According to this study, it is most often needed to specify the presence of given keys and/or values in an array, and bounds for its size. The following section proposes a syntax for these specifications.

5.1.2 Array conditions

We extend the syntax of the array declaration `a: array(`D`, `L`)`, introduced in Section 3.3, in Praspel with the following conditions on arrays.

A *pair condition* is of the form `a[`K`]:` V where K and V are realistic domain disjunctions. The condition means that the pairs constituted of all the keys in K and at least one value in V are present in the array `a`. K only accepts realistic domains that implements the `Constant`, `Interval` and `Enumerable` interfaces. This is equivalent to use the `array_key_exists()` and `in_array()` functions combined.

If we would like to express a constraint only on K, we can use the symbol `_`. The condition `a[`K`]: _` means that all the keys from K must be present in the array `a`. It is equivalent to use

only the `array_key_exists` function with all values in K in conjunction. The condition `a[_]:` V means that all the values in V must be present in the array `a`. It is equivalent to use the `in_array` function with all the values in V in conjunction.

Instead of the `:` symbol, we can use the symbol `!:` to express a negation. The condition `a[`K`]!:` V means that all the keys in K have a value in the array `a` and that this value is not in V. It works similarly with the symbol `_`. For example, the condition `a[`K`]!: _` means that no key in K appears in `a`.

Keys of an array are always unique, but not its values. We can express a unicity condition on values by writing the condition `a is unique`. In this case, we cannot have the same value twice in the array `a`.

Example 7 (Array conditions)**.** *The following Praspel specification shows three array conditions on a PHP array* `a`*, accompanying its description:*

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

5.1.3 Constraints

This section describes the semantics of array conditions by constraints. Generating an array satisfying these conditions will finally consist in solving these constraints. One of these conditions is assumed to be an array description of the form `a: array(`D`,`L`)`, where D specifies the array content by a list of p constructs of the form `from` F_i `to` T_i with $1 \leq i \leq p$, and L specifies the array content by a disjunction L_1 `or` ... `or` L_m (with $m \geq 1$) of realistic domains $L_1, \ldots, L_m$ inheriting from the `Integer` realistic domain and representing non-negative values. Without risk of confusion, a domain disjunction D_1 `or` ... `or` D_n will often be identified with the set $D_1 \cup \cdots \cup D_n$.

In Example 7, $p = 1$, $m = 2$, $\mathtt{L}_1 = [\mathtt{0..5}]$ and $\mathtt{L}_2 = \{\mathtt{10}\}$. In the array description, no domain is declared. In this case the default realistic domain `integerpp(0,1)` is used (an auto-incremented integer starting at `0` with a step of 1). In this example F_1 `= integerpp(0, 1)` and T_1 `= string('a', 'e', 1)`.

In the constraints associated to an array, the array is described by the following variables:

- two sets X and Y, respectively for the array domain (set of keys) and codomain, with $X = X_1 \cup \ldots \cup X_p$ (resp. $Y = Y_1 \cup \ldots \cup Y_p$), where X_i (resp. Y_i) is a subset of the realistic domain F_i (resp. T_i), for $1 \leq i \leq p$,
- a total function H from X to Y representing the array content, since keys are unique in a PHP array; when $x \in X$ holds, $H(x) = y$ means that the key-value pair (x, y) is in the array,
- a non-negative integer S for the array size.

Let $\mathrm{card}(E)$ denote the cardinality of the finite set E. The constraints $\mathrm{card}(X) = S$, $S \geq 0$ and $S \in L_1 \cup \cdots \cup L_m$ respectively say that the array size is its number of keys, is non-negative and satisfies all the length conditions. The constraint $\mathrm{card}(Y) \leq \mathrm{card}(X)$ should always hold. In presence of the array condition `a is unique`, this constraint becomes $\mathrm{card}(Y) = \mathrm{card}(X)$.

The constraints associated to conditions on keys and/or values are:

- $K \subseteq X$ and $H(K) \subseteq V$ for the pair condition `a[K]:` V, where K and V are domain disjunctions,
- $K \subseteq X$ and $H(K) \cap V = \emptyset$ for the negated pair condition `a[K]!:` V,
- $K \subseteq X$ for `a[K]:` `_`,
- $K \cap X = \emptyset$ for `a[K]!:` `_`,
- $V \subseteq Y$ for `a[_]:` V, and
- $V \cap Y = \emptyset$ for `a[_]!:` V.

In Example 7 and for the condition `a[0]: 'b' or 'd'`, we have $K = \{0\}$ and $V = \{'b', 'd'\}$. The constraints are $\{0\} \subseteq X$ and $H(\{0\}) \subseteq \{'b', 'd'\}$.

5.1.4 Constraint solver

Classically, our constraint solver is based on a propagation and a labelling mechanism.

Propagation of constraints uses an AC3 algorithm [32] implemented in PHP. So, we use five kinds of domains associated to five kinds of realistic domains: `Constant`, `Interval`, `Nonconvex`, `Finite` and `Enumerable` (see Section 2.4). For each kind of domain, we have implemented a *revise* method to allow the domain reduction. So, the consistency is also checking that there is no empty domain for the four variables S, H, X and Y but not for X_i and Y_i. The goal is to detect inconsistencies as soon as possible.

During the labelling process, we use a heuristic to find at first a value to the variable S. We can use this value to unfold the $\forall$ and $\exists$ quantifiers associated to the variables F_i and T_i because in this case they are enumerable (they become finite sets). Then, the solver tries to compute the sets X_i and Y_i. After the propagation, we use a random generator to generate a value in a realistic domain, to select a realistic domain in a disjunction, etc. The generated value is then propagated. If an inconsistency is detected, we add a new constraint to discredit the value, and then generate another one. For instance, if $S = 5$ leads to an inconsistency, we add the constraint $S \neq 5$. The added constraint is removed during the backtracking step.

When all variables are labelled, i.e. each one has a valid value, the solver returns a solution.

5.2 Grammar-based generation for strings

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 [35] mechanism that makes it possible to express and validate structured textual data.

To achieve that, we have introduced the `grammar` realistic domain which makes it possible to express a data using a specific grammar [15]. To support this realistic domain, we consider 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 tokens, possibly bounded by a (user-defined) maximal number of tokens they may contain. Tokens are defined by regular expressions. We have also introduced the `regex` realistic domain which makes it possible to express a data using a regular expression instead of a grammar.

```
%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 16: Simple grammar of XML documents

5.2.1 The PP Grammar Language

The PHP Parser language (PP for short) aims to express top-down context-free grammars in a simple way. The syntax is mainly inspired from JavaCC [29] with the addition of some new constructions. The objective of parsing is to produce an abstract syntax tree for syntactically correct data.

Figure 16 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 grammar continues with four grammatical rules, named `xml`, `tag`, `attribute` and `text`. The main rule `xml` describes an 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 is composed of at least two lines. The first line contains the rule name (such as `xml` or `tag` in Figure 16) and a colon (`:`). The other lines contain the rule declaration, 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: $\{x, y\}$ (resp. $\{x, \}$) denotes a pattern repetition from x to y times (resp. x times or more), ? is a shorcut for $\{0, 1\}$, + for $\{1, \}$, and $\star$ for $\{0, \}$. Disjunctions are represented by the symbol | and symbols are grouped by left and right parentheses (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 $i \geq 0$, it defines a *unification*. A unification for tokens implies that all *`token`*`[`i`]` with the same i have the same value locally inside the rule. Notice the presence in Figure 16 of a unification of tokens, namely `tagname[0]`, indicating that the opening and closing tag names should be the same.

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:

```
@requires xmlText: grammar('xml.pp');
```

where the content of the file `xml.pp` is given in Figure 16.

Each grammar is processed using a dedicated compiler compiler, which generates a compiler, able to recognize a given string/text. This ensures the predicability feature of realistic domains, by checking that a data is correctly structured accordingly to the grammar. The samplability feature is ensured by three possible grammar-based data generators, described in the following paragraphs. These test data generation algorithms are shipped with the `grammar` realistic domain and can be selected by the user.

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.

Bounded Exhaustive Generation. Bounded exhaustive testing consists of generating all possible data up to a given size. Some experiences [34, 39] 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.

Rule Coverage Generation. For grammar-based testing we propose a new generation algorithm less costly than the previous two and aiming 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 in order to bound 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.

To avoid combinatorial explosion and guarantee the termination of the algorithm, a boundary test generation heuristics [6] 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.

5.3 Object generation

The last two sections explained how we generate scalar types, such as booleans, integers, floats and strings, and also more complex types, such as arrays. We now describe how to generate object data, as PHP is mainly employed using an object oriented paradigm.

An object is an aggregation of attributes and methods. Both are annotated by Praspel contracts, respectively by `@invariant` and by other clauses (see the *attribute-clauses* and *method-clauses* syntactic entities in Figure 3, and the Figure 5 for an example). Then, the strategy to generate an object is to instantiate it by calling its constructor, and set a value to all its attributes. The values for the arguments of the constructor are generated thanks to its precondition. The values for the attributes are generated thanks to the invariants. If an argument of the constructor or an attribute is itself an object, this process is repeated recursively. We shunt the encapsulation to access to protected or private attributes or methods by instrumenting the code source.

In order to deal with circular references (e.g. when a first object has an attribute that points to a second object, which has an attribute that points to the first object), we apply a specific strategy, inspired from [38]. This strategy consists in storing created objects in a pool. When an object is needed, a choice is made between creating a new object or picking one from the pool. This choice is dynamically made according to the state of the store: the more existing objects, the lesser object creations.

6 Experimentations

This section presents experiments performed to validate the generation of test cases based on contract coverage criteria. Section 6.1 presents the mechanism of test execution and its integration in an existing testing framework, used to run the experimentations. Section 6.2 considers the case study of a robot, and compared the manual and automated testing processes, to evaluate the capabilities and benefits of Praspel. Section 6.3 makes the same comparison with a panel of engineers who volunteered to learn Praspel and apply it in their developments.

6.1 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.

We have written a bridge between Praspel and Atoum[1], an external tool, which is the current integrated unit testing framework for test execution in the industry. The bridge generates test cases written with the API of Atoum. This approach has two main advantages: (*i*) there is a large community and user-base (including important industrial companies) around Atoum which

[1] See `http://atoum.org/`.

are very active and pleased to take a part in new test techniques, approaches and tools, (*ii*) the framework is integrated into major industrial tools, platforms, IDE, etc.

6.2 Case study: the robot

We first present an evaluation that we made of our approach. The case study we considered is a robot that was presented during the *JDev* sessions[2]. During this event, we have presented a case study to a group a 16 students for a practical training session on unit testing. This case study is a simple robot that can move, using relative or absolute coordinates. When it is moving, it consumes energy, that is regained permanently (e.g. using solar panels), but slowly; it eventually runs out of energy if it stays in motion.

The program is composed of 5 classes describing:

- the *robot* itself,
- *vector*s to represent sequences of data,
- *coordinates* manipulated by a GPS of the robot,
- a *land sensor*, and
- a *clock*.

These 5 classes count 21 methods and represent around 340 lines of code. This case study illustrates the major aspects of unit testing, namely, writing a preamble, defining assertions, or mocking objects.

We have re-used this robot example and the training session results to validate the generation of test cases based on contract coverage criteria. We have applied the following experimental protocol: (*i*) manually write a test suite, (*ii*) annotate all methods with Praspel contracts, (*iii*) generate automatically a test suite satisfying the CDP contract coverage criterion (defined in Section 4.2), and execute them, (*iv*) compare the manual tests (MT) with the automatically-generated tests (AGT).

For step (*i*), we considered the tests produced by the students for our experimental data. Steps (*ii*), (*iii*) and to (*iv*) were done separately, as the purpose of the JDev training session was not to learn Praspel. The comparison criterion between the MT and the AGT is the code coverage score, along the *all-nodes* code coverage criterion proposed by Atoum. In both cases, the score is 100%. However, the average number of MT is 53, while the number of AGT is 29: It represents a reduction of 45%. Moreover, the students took an average time of 1h30 to write the complete test suite of MT, while 15 minutes were sufficient to write all the contracts. The test generation phase took less than a second. It involves the automatic generation of test data, namely integers, floats, strings, specified by regular expressions (grammar-based algorithms) and objects, especially for full robot instances (robots with their associated components).

Thus, we obtain the same code coverage score with almost half the tests and 6 times less time. These first results show the efficiency of the automated testing approach, and the gain that can be obtained, as writing a contract is less difficult, and less time-consuming, than writing a full test suite. However, this experiment was biased as we are Praspel experts. In order to evaluate our approach more objectively, we asked external test engineers to do a similar experiment on their code, to evaluate the expressiveness of Praspel, their understanding of the language, and the gain that they can expect from this approach.

[2] A French national meeting focused on software quality. See `http://devlog.cnrs.fr/`.

6.3 Real-world experiment

We have selected a panel of three test engineers, having an experience ranging from 5 to 8 years in PHP and (unit) testing techniques, and working in different industrial domains, namely: hospital, financial and bookkeeping. We gave them the following experimental protocol:

(i) *select groups of methods that you have already manually tested: One group g_1 of methods that manipulate numbers (integers or floats), one group g_2 of methods that manipulate arrays, one group g_3 for strings and one group g_4 for objects,*

(ii) *annotate those methods with Praspel contracts,*

(iii) *generate automatically test suites satisfying the CDP contract coverage criterion (defined in Section 4.2), and execute them,*

(iv) *finally compare your manual tests (MT) with the automatically-generated tests (AGT).*

The comparison criteria and the results of this experimentation are given below. We asked our panel a set of questions, aiming to evaluate our approach.

6.3.1 *How does code coverage compare between AGT and MT?*

This section presents the feedback about the quality of the AGT. We have asked the panel to compare the AGT to the MT, along the *all-nodes* code coverage criterion proposed by Atoum. We have asked the test engineers to study the AGT in order to give an informal feedback on its quality, based on their experiences. The answers to this first question are summarized in Figure 17.

For the group of methods g_1 (manipulating integers and floats), both MT and AGT achieve a 100% code coverage. However, the AGT were not always able to find the bugs that were detected by the MT. Most of the time, this was due to boundary values. As Praspel uses a pseudo-random generation, such boundary values is (almost) never sampled. Also, we see that the numbers of MT and AGT are very similar, there is only -16.26% less AGT than MT. This gap corresponds to specific tests for boundary values.

	g_1 integers, floats			g_2 arrays			g_3 strings			g_4 objects		
	1	2	3	1	2	3	1	2	3	1	2	3
Code coverage												
MT	100%	100%	100%	100%	100%	100%	76%	95%	100%	100%	100%	100%
ATG	100%	100%	100%	96%	93%	100%	95%	98%	100%	92%	79%	100%
Δ	0%			-3.66%			+7.33%			-9.66%		
Number of tests												
MT	12	16	15	11	9	10	10	11	15	10	8	11
ATG	11	13	12	4	4	2	3	5	8	7	7	8
Δ	-16.26%			-66.66%			-55%			-24.13%		

Figure 17: Code coverage score and number of tests for the MT and the AGT for each group of methods.

For the group of methods g_2 that manipulate arrays, the MT have a 100% code coverage, whereas the AGT have only 96.33%, but there is an important difference between the number of tests: -66.66%. According to the panel, the error detection capabilities of both test suites are similar.

For the group of methods g_3 that manipulate strings, the MT have 90.33% code coverage, whereas the AGT have 97.66%. According to the panel, this is due to the `regex` realistic domain that uses the grammar-based testing with the rule coverage generation algorithm (see Section 5.2). This algorithm produces textual data that cover all the rules and all the tokens in a grammar. Consequently, the number of AGT is twice less than the number of MT, since the test data are more accurate.

For the group of methods g_4 that manipulate objects, the MT have a 100% code coverage, whereas the AGT only have 90.33%. The difference between the number of MT and AGT is -24.13%. In order to reach 100% of code coverage for the AGT, Praspel should be able to generate mocked objects automatically, since it was not able to instantiate some complex object models (like based on a network, web-services, etc.).

6.3.2 *What is the most time-consuming, writing manual tests or writing a contract in Praspel?*

The syntax of Praspel is far from usual specification languages, such as B or Z. The syntactic proximity of Praspel with PHP was design to allow developers to quickly learn the language.

Most of the people involved in this experimentation were new to Praspel. However, they understood the syntax and semantics of the language, in a couple of minutes, and were quickly able to write their firsts contracts. In general, an engineer in the panel needs around 3 minutes to write tests for a method in groups g_1 or g_3, 4 minutes to write tests for a method in group g_2 and 3 to 15 minutes for a method in group g_4 (depending of the complexity of the object model). With Praspel, an engineer needs less than 1 minute to write a contract for the group g_1, between 1 to 2 minutes for the groups g_2 and g_3, and around 5 minutes for the group g_4. On average, an engineer needs less than half the time to write contracts and generate tests than to write manual tests.

However, after obtaining a test suite automatically, the engineer sometimes has to write some tests manually to increase the code coverage score, if he considers that the current score is not satisfactory. The contract (resp. code) coverage criteria gives some information about the parts of the contract (resp. code) that have not been covered (e.g. for the contract: a predicate, for the code: a condition). It thus provides a way to identify the remaining targets.

Moreover, one engineer of the panel noticed that writing a contract leads to more reflection about the code and the design of the software. It fits the TDD (Test-Driven Development) software development process: One first writes the contract, then generates tests covering it, and finally writes an implementation in order to satisfy the contract and pass all the generated tests.

6.3.3 Does the test data generator help the manual testing process?

When Praspel is not able to generate enough tests to achieve a suitable coverage, the validation engineer has to design the tests by himself. Since Atoum has nothing to generate data automatically, the only way is to use fixtures, such as a collection of static pre-computed data. This is a laborious and hard task. To help the user on this task, we exposed the data generators API, so that he can use them to generate test data that he has in mind.

We asked to the panel the following questions: *Do you use the automatic data generators?*, and *Does it improve the quality of your manual tests?*. The overall answer was positive. The automatic data generation is used for both manual and automatically generated tests. The most

used realistic domain is `regex`, with the grammar-based algorithms. As expected, the second most used is `array`.

Finally, in Section 2.1, we said that realistic domains use a numeric sampler, which is currently implemented as a pseudo-random generator. Even if some realistic domains have specific generation strategies, we asked the engineers the following question: *Is the (pseudo-)random aspect of the automatically generated data suitable for your usages?*. The answer was also positive. They explained that there exists no tool to automatically generate data. With data generators, such a hard task disappears and fits the current needs. However, in few cases, it has been noticed the random aspect is not always suitable for their kinds of manipulated data. When random is not desired, they used a dictionary of values.

In conclusion, the CDP contract coverage criterion is able to be as good as what experimented test engineers do, in half the time. This result is similar in our internal experimentation with the robot case study. In addition, one can use the test data generator to produce complex values such as arrays and strings from a lightweight data specification.

7 Related Work

While many works consider PHP as a target for security test generation, only a few focus on functional testing, as in our approach. One notable exception is the Apollo tool [2] which is able 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 it makes it possible not only to validate a piece of HTML code (produced by a Praspel-annotated function/method), but also to express and check more general properties using server-side assertions. Since Praspel embraces different aspects of the test domain, this section follows two axes.

7.1 Contract Driven Testing

Various works consider Design-by-Contract for unit test generation [1]. JML, the main annotation language for Java, has been widely targeted during the last decade [9, 38, 42, 10]. Especially, our test verdict assignment process relies on runtime assertion checking, which is also considered in JMLUnit [9], 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 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.

These approaches aim at generating tests for Java (or, more generally, for object-oriented programs) and use the contract both as a filter, to rule out irrelevant test cases (i.e. tests that do not fulfil the method precondition), and as an oracle, to establish the test verdict. Madsen [33] proposes an extension to Java aiming to describe equivalence classes inside contracts and introduces partition coverage as test objectives. However, the partitioning is done by the validation engineer/developer. It is not extracted automatically from the structure of the contract as in our approach. To the best of our knowledge, the idea of generating tests in order to satisfy

a given criterion of contract coverage is original and has never been developed before, except in our former work on JML [7, 12].

7.2 Test Data Generation

The domain of grammar-based testing has been widely covered in the literature, and applied to many application domains especially related to security testing [26, 20]. One of the most experienced grammar-based test generator is yagg [11], based on the yacc syntax, which implements a bounded exhaustive testing approach. Geno [30] 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 [27] 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 [28] 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 [25], we believe that its use as a test generation criterion may provide an interesting trade-off between exhaustiveness and computational efficiency.

Korat [8] 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 predicability 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 [41] 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 exhaustive exploration of a system state space. Recently, the UDITA language [19] 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).

Euclide [21] is a constraint-based testing tool that could take as input additional safety properties defined in ACSL [22]. Our approach differs by handling conditions directly in the contract. All constructions present in Praspel are well-handled for both aspects: validation and generation. The CLP framework InKA [23] helps computing structural test data from a C program. It transforms the problem of automatic test data generation into a CLP problem over finite domains. In the same way, FDCC [3] is a combined approach for solving constraints over finite domains and arrays. The tricky part of FDCC lies in a bi-directional communication mechanism between two solvers. Our constraints are more specific but we use only one solver.

8 Conclusion and Future Works

This paper has presented a contract-driven testing approach for PHP. It relies on a contract description language, called Praspel, to write formal assertions and typing information inside

classical contract clauses (invariant, pre- and postconditions). These contracts are then used to generate test data using dedicated test generators (e.g. for arrays and strings). The approach is driven by original contract coverage criteria introduced in the article. It is implemented into an integrated framework, distributed to the PHP community of users, which experimented the use of Praspel and its relevance for their daily validation activities.

Notice that one of the main arguments against annotation 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 the source code of the application is always available.

The research questions we introduced are thus answered as follows.

1. *To what extend is it possible to use the annotations written in contracts for generating model-based tests?*
   We introduced dedicated contract coverage criteria that drive the test generation in a black-box manner (i.e. without considering the code of the program). This approach goes further than the usual use of contracts for testing, only to filter irrelevant tests. Here tests should additionally fulfil method preconditions. In this work, we use the contracts to define equivalence classes that are taken from the structure and semantics of the Praspel language. Thus, the coverage criteria that we defined here could be used for other BISL, such as JML or ACSL.

2. *To what extend is it possible to automate the test generation for dynamically-typed languages, such as PHP?*
   The notion of realistic domain that we presented can be used to replace and refine usual data typing information. It improves the accuracy of the test data (e.g. an email address is a refinement of a string). In this work, we have presented Praspel as an implementation of realistic domains for PHP. However, the underlying principles are more general, and could be adapted to any other dynamically typed language (e.g. JavaScript, Python, etc.).

3. *To what extend is it useful for developers to dispose of a integrated contract-based testing framework?*
   We have proposed an integrated framework for Praspel [13] that has been distributed to the PHP community. Praspel is also currently being integrated into Atoum, the current state-of-the-art unit testing framework for PHP. Our experimentations with a group of developers show the interest of Praspel, and the efficiency with which they appropriated the tool and its underlying formal concepts, employed in a transparent manner.

For the future, we plan to extend the Praspel language with the possibility to specify temporal aspects of classes. The idea is to provide a high-level temporal language that would be easy to use by the developers, again by hiding formal concepts behind a simple and expressive language. To achieve that, we propose to revisit the JTPL principles, a temporal language based on Dwyer's property patterns, that was applied to Java [18].

Acknowledgements. *We are very grateful to Frédéric Hardy, Julien Bianchi, and the Hoa and Atoum communities for their help and patience.*

References

[1] B. K. Aichernig. Contract-based testing. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA'08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 261–272, New York, NY, USA, 2008. ACM.

[3] S. Bardin and A. Gotlieb. FDCC: A Combined Approach for Solving Constraints over Finite Domains and Arrays. In N. Beldiceanu, N. Jussien, and E. Pinson, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June 1, 2012, Proceedings*, volume 7298 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2012.

[4] M. Barnett, K. R. M. 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)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Marseille, France, March 2004. Springer.

[5] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language Version 1.7*, 2013. `http://frama-c.com/download/acsl-implementation-Fluorine-20130601.pdf`.

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

[7] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In T. Nipkow and J. Misra, editors, *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 428–443, Hamilton, Canada, August 2006. Springer.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing based on Java Predicates. In *ISSTA'02: Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[9] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer.

[10] Y. Cheon and C. E. Rubio-Medrano. Random Test Data Generation for Java Classes Annotated with JML Specifications. In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 385–391. CSREA Press, 2007.

[11] 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*, ASE '05, pages 356–359, New York, NY, USA, 2005. ACM.

[12] F. Dadeau, Y. Ledru, and L. du Bousquet. Measuring the Coverage of a Java Test Suite using JML Specifications. In B. Finkbeiner, Y. Gurevich, and A.K. Petrenko, editors, *MBT'07, 3rd Int. Workshop on Model-Based Testing, co-located with ETAPS'2007*, volume 190 of *ENTCS, Electronic Notes in Theoretical Computer Science*, pages 21–32, Braga, Portugal, April 2007.

[13] I. Enderlin. Hoa project, 2010. `http://hoa-project.net`.

[14] I. Enderlin, F. Dadeau, A. Giorgetti, and A. Ben Othman. Praspel: A Specification Language for Contract-Based Testing in PHP. In B. Wolff and F. Zaïdi, editors, *Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*, volume 7019 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2011.

[15] I. Enderlin, F. Dadeau, A. Giorgetti, and F. Bouquet. Grammar-Based Testing Using Realistic Domains in PHP. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation Workshops, Montreal, QC, Canada, April 17-21, 2012*, pages 509–518. IEEE, 2012.

[16] I. Enderlin, A. Giorgetti, and F. Bouquet. A Constraint Solver for PHP Arrays. In A. Orso, B. Baudry, and Y. Le Traon, editors, *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, March 18-22, 2013*. IEEE, 2013.

[17] P. Flajolet, P. Zimmerman, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1-2):1 – 35, 1994.

[18] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java modeling language. *IET Software*, 2(6):500–514, December 2008.

[19] 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*, pages 225–234, New York, NY, USA, 2010. ACM.

[20] 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*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[21] A. Gotlieb. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *ICST 2009, Second International Conference on Software Testing Verification and Validation, 1-4 April 2009, Denver, Colorado, USA*, pages 151–160. IEEE Computer Society, 2009.

[22] A. Gotlieb. Tcas software verification using constraint programming. *Knowledge Eng. Review*, 27(3):343–360, 2012.

[23] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Moniz Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2000.

[24] P. Heidegger and P. Thiemann. Contract-Driven Testing of JavaScript Code. In *TOOLS 2010 - 48th Int. Conf. on Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010.

[25] 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.*, 13:343–368, August 2008.

[26] D. Hoffman, H.-Y. Wang, M. Chang, and D. Ly-Gagnon. Grammar based testing of html injection vulnerabilities in rss feeds. In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, TAIC-PART '09, pages 105–110, Washington, DC, USA, 2009. IEEE Computer Society.

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

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

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

[30] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In M. U. Uyar, A. Y. Duale, and M. A. Fecko, editors, *The 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006), New York City, USA, May 16-18, 2006*, volume 3964 of *Lecture Notes in Computer Science*. Springer, 2006.

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

[32] A. Macworth. Consistency in network of relations. *Journal of Artificial Intelligence*, 8(1):99–118, 1977.

[33] Per Madsen. Enhancing design by contract with knowledge of equivalence partitions. *Journal of Object Technology*, 3(4), 2004.

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

[35] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Softw.*, 7:50–55, July 1990.

[36] B. Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22(2):85–94, 1987.

[37] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[38] C. Oriat. Jartege: A tool for random generation of unit tests for java classes. In R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2005.

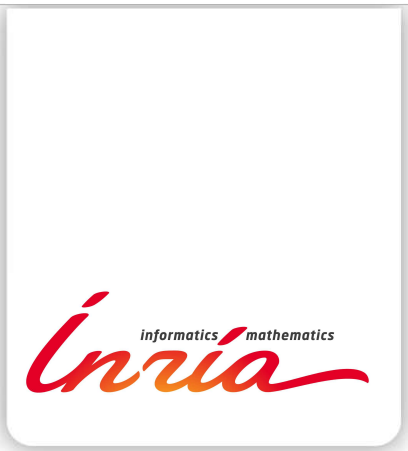
[39] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA'04: Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 133–142, New York, NY, USA, 2004. ACM.

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

[41] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[42] G. Xu and Z. Yang. Jmlautotest: A novel automated testing framework based on jml and junit. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2003.

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



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399