A Constraint Solver for PHP Arrays

Ivan Enderlin, Alain Giorgetti, Fabrice Bouquet

A Constraint Solver for PHP Arrays

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

***Abstract*—In previous works, we have proposed Praspel, a framework for contract-based testing in PHP. Among others, it includes a specification language and a unit test generator which automatically generates test data from formal preconditions. The generator sometimes rejects data, when they do not satisfy parts of the preconditions. In many cases, generation with rejection is not efficient enough. Thus we investigate practical contexts where more efficient generation algorithms can be designed and we extend Praspel with their implementation.**

After strings, that we have already considered, the most frequent data type in PHP is arrays. They 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. In this paper, we report on a study to know what are the most popular constraints on PHP arrays. Then we formalize these constraints and we present an implementation in PHP of a constraint solver for these constraints. In this context, the constraint-based approach removes all the rejections.

***Keywords*-array, constraint, solver, php, realistic domain, praspel**

I. Introduction

Contract-based testing [1] is a promising approach to increase software safety and security. It is based on the notion of Design by Contract, introduced by Meyer [2] with the Eiffel language [3]. A contract is a formal specification associated to the code of a program. It mainly consists of invariants, pre- and postconditions. Invariants describe properties that should hold at each step of the execution. Pre- and postconditions respectively represent conditions that have to hold for a method to be invoked, and conditions that have to hold after execution of the method.

Various contract languages extend programming languages with contracts: JML extends Java [4], ACSL extends C [5], Spec# extends C# [6], etc. Among the numerous advantages of contracts, formal properties can be exploited for (unit) testing. Indeed, the information contained in invariants and preconditions can be used to generate test data. In addition, these assertions can be checked at run time and thus provide a (partial) test oracle for free.

In previous works, we have introduced Praspel [7], a tool-supported specification language for contract-based testing in PHP [8]. Praspel extends contracts with the notion of *realistic domain*, which makes it possible to assign a domain of values to data (class attributes or method parameters). A library of predefined realistic domains is already available along with a test environment [9].

Validating PHP web applications often involves to manipulate strings and arrays. Strings have been already addressed [10] by introducing grammar-based testing in Praspel. This paper focus on PHP arrays, which are always *associative arrays*. They cover most of the needs for collections, because they can store key-value pairs of any kind (they can be homogeneous and heterogeneous), they do not have a specific length or depth, and they are efficiently implemented. When generating arrays for testing, the main difficulty is to satisfy their properties, which are formalized by predicates in their specification. We address this issue in the context of PHP and Praspel, by designing a specific constraint solver.

Our contributions are twofold. First, after studying the most popular conditions on PHP arrays, we propose a syntax to specify them in Praspel and a semantics of these array specifications. Second, we propose a new constraint solver in PHP for PHP, to generate arrays satisfying these conditions. These two contributions are embedded in realistic domains and can be used in Praspel annotations of a PHP program.

The paper is organized as follows. Section II briefly introduces the notion of realistic domain and its implementation in Praspel for PHP. Section III presents a language of array conditions in Praspel, inspired by a study to know what are the most popular conditions expressed on arrays in popular PHP code. Section IV presents the constraint solver. Then, Section V reports on an experimentation validating our approach and tool, and showing its usefulness and efficiency in practice. Related works are presented in Section VI. Finally, Section VII concludes and presents future works.

II. Realistic domains and Praspel

This section is a reminder of [7]. It shortly presents the notion of realistic domain and its application to PHP programs. Realistic domains are designed for test generation purposes. They specify which values can be assigned to a data in a given program. They are well-suited to PHP, since this language is dynamically typed (i.e. no types are syntactically assigned to data) and realistic domains thus introduce a specification of data types mandatory for test data generation. We first introduce general features of realistic domains. Then we present their implementation in PHP and the Praspel framework.

A. Features of Realistic Domains

Realistic domains refine usual datatypes (integer, string, array, etc.). They are intended to specify data domains relevant for specific application contexts. For example, email addresses constitute a realistic domain: many applications identify a user by her email address, so we need to validate and generate such

data, and an email address is more than a string: it obeys to a specific syntax that makes it not obvious to generate.

The first feature of a realistic domain (named its *predicability*) is to carry a characteristic predicate of its values, used to check if a value belongs to the possible set of values described by the realistic domain. The second feature of a realistic domain (named its *samplability*) is to propose a value generator, called the *sampler*, that generates values in the realistic domain. For instance, a sampler for the realistic domain of email addresses can generate strings representing syntactically correct email addresses defined by a regular expression.

B. Realistic Domains in PHP

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

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

1) Inheritance: PHP realistic domains can inherit from each other. A realistic domain child inherits the two features of its parent, namely predicability and samplability, and is able to redefine them. Consequently, we say that all the realistic domains constitute a hierarchical universe.

2) Interfaces: All data are represented by realistic domains. Some of them implement interfaces which characterize them. Useful interfaces for our current concerns are:

- `Constant`: represents an immutable realistic domain with only one value, such as `42`, `true`, etc.
- `Interval`: represents an interval by its lower and upper bounds, that can be dynamically reduced,
- `Nonconvex`: allows to discredit values, i.e. to specify that a value no longer belongs to a realistic domain and should therefore not be generated,
- `Finite`: allows to count the number of values,
- `Enumerable`: allows to iterate over all the values.

Thus, a realistic domain that implements the `Interval` and `Nonconvex` interfaces is an interval with "holes". Counting and exhaustive generation of a finite realistic domain take discredited values into account.

3) Parameterization: Realistic domains may have *parameters*, like a function does. Data given to a realistic domain are called *arguments*. This feature helps generating structured data such as arrays, objects, graphs, automata, etc.

Example 1 (Realistic domains with arguments)**.** *The realistic domain `string(0x61, 0x7a, boundinteger(4, 12))` admits as arguments two integers (that represent two Unicode code-points) and a domain of integers to specify its possible length. The realistic domain `boundinteger(`X`, `Y`)` contains all the integers between X and Y. We can also write X`..`Y as syntactic sugar for this domain. The realistic domain `string(`X`, `Y`, `L`)` is intended to contain all the strings of length (in the domain) L built of characters from X to Y code-points.*

When describing parameters, because all data are realistic domains, we can multi-*type hint* the parameters, i.e. filter with multiple realistic domain names. For instance, the first two parameters of the realistic domain `string()` are described as `Constinteger | Conststring`, so can be a constant integer or a constant string. Then, the syntax `string('a', 'z', 4..12)` is strictly equivalent to the previous one. If we write `string(true, 'z', 4..12)`, an error will be thrown. The realistic domain itself handles the cast.

C. Praspel and Contract-Based Testing in PHP

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

Praspel annotations are written inside comments in the source code. Invariants document classes and pre- and post-conditions document methods. The general form of Praspel annotations is shown in Figure 1. For lack of space, named behaviors and specification of exceptions are not presented. In this specification, I_1, ..., I_h are invariant clauses, assumed to be satisfied at the beginning and at the end of each method invocation. Formulas R_1, ..., R_n and A_1, ..., A_k are precondition clauses, that have to hold at the invocation of the `foo` method. Formulas $E_1, \ldots, E_m$ are postconditions that have to be established when method `foo` terminates without throwing an exception. In postconditions, Praspel provides the two additional constructs `\result` and `\old(`e`)`, which respectively designate the value returned by the method, and the value of expression e at the pre-state of the method invocation.

PHP does not provide a type system, but Praspel contracts make it possible to give typing informations, assigning realistic domains to data (class attributes or method parameters). The construction `i:` t_1`(...) or ... or` t_n`(...)` associates at least one realistic domain, among $t_1(\ldots), \ldots, t_n(\ldots)$, to the identifier `i`. Every identifier holds a realistic domain *disjunction*, thanks to the `or` keyword. Identifiers can also be passed into arguments of realistic domains.

Example 2 (Identifiers in Praspel)**.** *The declarations:*

```
length: 4..12
str   : string('a', 'z', length)
```

show a dependency between two identifiers: The third argument of the realistic domain of the second identifier `str` is the first identifier `length`.

Praspel provides a set of more than 30 predefined realistic domains, called the standard library. Some of them correspond to scalar types (`integer`, `float`, `boolean`...), other ones to classes and arrays (detailed in Section III).

Contractual assertions are made of realistic domain declarations, possibly completed with additional predicates, expressed

in PHP using the \pred construct.

```
class C {
   /** @invariant I_1 and ... and I_h */
   /** @requires  R_1 and ... and R_n;
    *  @ensures   E_1 and ... and E_j; */
   function foo ( $x_1... ) { body } }
```

Figure 1. Syntax of contracts in Praspel

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

III. Arrays in Praspel

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 (casted into integers), integers, floats (reduced to their integer parts) or strings. 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.

A. Array Description

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 3 (Homogeneous and heterogeneous arrays)**.** *This syntax 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. This length is a realistic domain that implements the `Interval` interface. 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. Its length is a realistic domain that implements the `Constant` interface, the domain for its keys is the disjunction of two realistic domains (`0..5` and `10..23`) which implement the `Interval` interface, etc. The identifiers `a3` and `a4` are declared as heterogeneous arrays. Both arrays can contain the pairs $(5, \texttt{true})$ *and* $(15, \texttt{4.2})$*, but `a4` can contain the pair* $(5, \texttt{4.2})$*, whereas `a3` can not contain it.*

Actually, we introduce a *normal form* that removes disjunctions in array descriptions (in `from ...to ...` constructs). An array description is in normal form when it can not be reduced by 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).

Example 4 (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)
```

B. Collecting Information

When generating test data, the Praspel testing tool calls the `sample()` method on a realistic domain and then checks whether the predicates declared with `\pred()` hold. For an array with conditions, a random generator may produce a lot of rejected data before getting a valid one. The idea is to determine the most popular conditions on arrays expressed in PHP (usually written in the `\pred()` construct), to allow them inside Praspel, and to use a solver to satisfy these conditions.

To achieve this, 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 count the number of occurrences of each array function available in the PHP standard library. It appeared that the three most 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. This study suggests that we could consider these side-effect-free Boolean functions as the most frequent conditions on arrays.

C. Array Conditions

We extend the syntax of the array declaration `a :` `array(`D`,`L`)` 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 5 (Array Conditions)**.** *To illustrate all kinds of conditions, we will use the following example that uses* `a`*:*

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

IV. Constraint Solver

Given a conjunction of array conditions on an array `a`, we propose to invoke a constraint solver to construct an array satisfying all these conditions. This section explains how array conditions are transformed into constraints for the solver. One of these conditions is assumed to be an array declaration of the form `a: array(D,L)`, where D is assumed to be in normal form, without loss of generality. In other words, D is a list of p constructs `from` F_i `to` T_i with $1 \leq i \leq p$. We also assume that L is L_1 `or ... or` L_m (with $m \geq 1$), where L_1, ..., L_m are realistic domains that inherit from the `Integer` realistic domain and that are non-negatives.

In Example 5, $p = 1$, $m = 2$, $L_1 = [0..5]$ and $L_2 = \{10\}$. In the array description, no domain is declared. In this case the default realistic domain `integerpp(0,1)` is used (an auto-incremented integer: 0, 1, 2, 3, etc). In this example $F_1 =$ `integerpp(0,1)` and $T_1 =$ `string('a','e',1)`.

Without risk of confusion, a domain disjunction D_1 or ... or D_n will often be identified with the set $D_1 \cup \cdots \cup D_n$.

A. Variables

The constraint variables are: (i) the array size –noted S– which is a non-negative integer, (ii) the sets X and Y, which respectively are the array domain (set of keys) and codomain (set of values), (iii) the array content noted H, which is a total function from X to Y, since keys are unique in a PHP array, (iv) the realistic domains[1] X_1, ..., X_p (resp. Y_1, ..., Y_p), which are subsets of the realistic domains F_1, ..., F_p (resp. T_1, ..., T_p) compatible with all the array conditions.

We are essentially interested in finding the content of X and the values of the function H, i.e. the content of H considered as a hashtable, possibly also the values of the X_is and Y_is for checking purposes. The other variables are introduced only to simplify the expression of constraints.

When $x \in X$ holds, $H(x) = y$ means that the key-value pair (x, y) is in the array. We extend H to subsets of X by the function $\hat{H}$ defined by $\hat{H}(E) = \{H(x) \text{ s.t. } x \in E\}$ for any subset E of X.

B. Cardinality Constraints

Let $\mathrm{card}(E)$ denote the cardinality of the finite set E. The constraints $\mathrm{card}(X) = S$ and $S \geq 0$ say that the array size is its number of keys and is non-negative.

By default, there is no unicity constraint on the codomain, so we only have the constraint $\mathrm{card}(Y) \leq \mathrm{card}(X)$ However, in presence of the array condition `a is unique`, this constraint becomes $\mathrm{card}(Y) = \mathrm{card}(X)$.

C. Constraints on the Array Size

When propagating constraints, the solver may refine the domains L_1, ..., L_m of possible values for the array size S for i in $\{1, \ldots, m\}$, and the array size S should be in one of these sets, i.e. the constraint to define domain of S is:

$$S \in L_1 \cup \cdots \cup L_m$$

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

D. Constraints on Domains and Codomains

The domain X and codomain Y of H are related by the constraint $Y = \hat{H}(X)$.

We expect that the constraint solver proposes us the array domain X (resp. codomain Y) as a disjunction X_1 `or ... or` X_p (resp. Y_1 `or ... or` Y_p) of realistic domains compatible with all the array conditions. We should have the equalities $X = \bigcup_{1 \leq i \leq p} X_i$ and $Y = \bigcup_{1 \leq i \leq p} Y_i$, and the inclusions: $X_i \subseteq F_i$ and $Y_i \subseteq T_i$ for i in $\{1, \ldots, p\}$. The pair (X_i, Y_i) with $1 \leq i \leq p$ should also satisfy the constraint $\hat{H}(X_i) = Y_i$ meaning that Y_i is the codomain of the restriction of H to X_i ($\subseteq X$).

E. Constraints on Pairs

For each array condition `a[K]: V` where K and V are domain disjunctions, we introduce the constraints: $K \subseteq X$ and $\hat{H}(K) \subseteq V$. A negated pair condition `a[K]!: V` is translated into the constraints: $K \subseteq X$ and $\hat{H}(K) \cap V = \emptyset$. For the condition `a[0]: 'b' or 'd'` in Example 5, we have $K = \{0\}$ and $V = \{'b', 'd'\}$. The constraints are $\{0\} \subseteq X$ and $\hat{H}(\{0\}) \subseteq \{'b', 'd'\}$.

[1] In fact, the solver will not handle realistic domains, but only sets.

F. Constraints on Keys or Values

The condition `a[`K`]: _` is translated into the constraint $K \subseteq X$, and its negation `a[`K`]!: _` into the constraint $K \cap X = \emptyset$. The condition `a[_]: `V is translated into the constraint $V \subseteq Y$, and its negation `a[_]!: `V into the constraint $V \cap Y = \emptyset$.

G. Propagation and consistency

Propagation of constraints uses an AC3 algorithm [11] 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 II-B2). 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.

H. Labelling

The labelling is the process of finding a value for each variable. In order to make the solver converge quickly to a solution, we use a heuristic that consists in chosing a value for the variable S at first. This helps to unfold the $\forall$ and $\exists$ quantifiers (because F_i and T_i are enumerables, we manipulate finite sets). Then, the solver tries to compute the sets X_i and Y_i. 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 the solution.

V. Experimentation

In all that follows, the word *system* stands for the expression "conjunction of array conditions". This section presents an experimentation evaluating the solver efficiency, i.e. its capability to avoid or reduce rejection when generating data from systems. We measure the number of backtracks in the solver, the time to generate data from satisfiable systems of array conditions, and how many unsatisfiable systems are detected. The experimentation is composed of three steps: system generation, then data generation (i.e. system solving) and finally a measuring step. We generate systems on arrays containing strings and integers, and of length 5 to 20.

The Praspel language (and in particular its sublanguage of array conditions) is described by a grammar. In order to generate systems, we re-use a previous work in grammar-based testing [10] proposing three algorithms generating data from grammars: a uniform random generator, a bounded exhaustive test generator, and a rule-coverage-based test generator. Since the grammar of array conditions is small, the last generator does not generate a sufficiently wide collection of systems. Bounded exhaustive testing is more costly than random testing, but it is more precise and well adapted to small grammars. We retain the bounded exhaustive test generator: for increasing values of n it enumerates all the systems composed of a sequence of n tokens: In this generator, a single token value of each variable token (i.e. interval bounds and particular array lengths, keys and values) is generated at random.

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

Table I
Experimentation results.

With $n = 3$, we can generate conditions of the form `arr is unique`. With $n = 6$, we can generate constraints of the form `arr[0]: 0`. With $n = 8$, we can generate constraints of the form `arr[0]: 0 or 1`. With $n = 11$, we can generate for instance the system in Example 5.

The second step calls the solver with each produced system to generate an array satisfying it. Every generated array is evaluated by the predicate associated to the system of array conditions, to check the solver soundness. During the data generation step, some marks are placed to count only the array generation time, without counting the compiling time. We also measure the number of backtracks in the solver and the number of rejected systems. When a system is rejected, its backtracks are not counted.

Since our grammar-based testing algorithms use an isotropic random generation to generate token values, these values can differ from one system to another. It may lead to different generation times and numbers of backtracks. To avoid peaks in the results, we report the average of 100 generations of systems sharing the same pattern.

Table I shows our experimentation results. In the first column, n is the length of the generated sequences of tokens representing systems of array conditions. Column 2 gives the number of distinct system patterns with this length. Columns 3, 5 and 6 respectively give average numbers of backtracks and rejected systems for 100 runs, and an average generation time. Column 4 gives the rate of backtracks per system pattern.

For $n \leq 20$, we observe that no system is rejected, which is a great improvement by comparison with random generation. All generated data satisfy their specification. The solver successfully and quickly generates data with a low number of backtracks. For $n = 20$, which represents approximatively 3 constraints with disjunctions, the exhaustive generation algorithm generates 492 distinct systems of array conditions and the execution produces only 114 backtracks, so approximatically 1 backtrack for 4 systems. This is a good result. Nevertheless, we were not able to characterize the number of backtracks with the number of constraints. Finally, this experimentation allowed us to find a bug in our solver. This bug always led to a rejection when only a certain constraint was analyzed. Thus, it shows that the validation

process also helps finding bugs.

VI. Related Works

Various works consider Design-by-Contract for unit test generation [12], [13]. Our specification language is based on JML [4] and ACSL [5]. Thanks to an expressive specification language, Praspel performs general runtime assertion checks. Realistic domains present some similarities with Eiffel's types [3], 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.

Euclide [14] is a constraint-based testing tool that could take as input additional safety properties defined in ACSL [15]. 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 [16] 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 [17] 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.

VII. Conclusion and Future Works

We have presented in this paper an extension of the Praspel language to specify usual conditions on PHP arrays. We have expressed its semantics by constraints. We have designed and implemented in PHP a constraint solver to generate test data from these constraints. It uses a random generator to ensure a diversity of generated solutions. This solver is integrated in realistic domains and can be used within the Praspel framework. A first validation shows cases where rejection has been totally removed. It also shows that the solver dramatically increases the generation speed.

In a near future, we plan to lead a more complete experimentation. Then, we plan to formalize more constraints and extend our solver. We also plan to transform constraints into the formalism proposed by MiniZinc [18] in order to compare our solver to other ones regarding performances and capabilities to find solutions. As a string is an array of characters, we would like to apply the same process on strings with the help of existing and promising solving techniques [19].

References

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

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

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

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

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

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

[7] I. Enderlin, F. Dadeau, A. Giorgetti, and A. B. Othman, "Praspel: A specification language for contract-based testing in php," in *ICTSS*, ser. Lecture Notes in Computer Science, B. Wolff and F. Zaïdi, Eds., vol. 7019. Springer, 2011, pp. 64–79.

[8] PHP Group, "The PHP website," 2010, URL: http://php.net.

[9] I. Enderlin, "Hoa project, a set of PHP libraries," 2010, URL: http://hoa-project.net.

[10] I. Enderlin, F. Dadeau, A. Giorgetti, and F. Bouquet, "Grammar-Based Testing Using Realistic Domains in PHP," in *ICST*, G. Antoniol, A. Bertolino, and Y. Labiche, Eds. IEEE, 2012, pp. 509–518.

[11] A. Macworth, "Consistency in network of relations," *Journal of Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[12] Y. Cheon and G. T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," in *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*, ser. LNCS, B. Magnusson, Ed., vol. 2374. Berlin: Springer, Jun. 2002, pp. 231–255.

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

[14] A. Gotlieb, "Euclide: A Constraint-Based Testing Framework for Critical C Programs," in *ICST*. IEEE Computer Society, 2009, pp. 151–160.

[15] ——, "Tcas software verification using constraint programming," *Knowledge Eng. Review*, vol. 27, no. 3, pp. 343–360, 2012.

[16] A. Gotlieb, B. Botella, and M. Rueher, "A CLP Framework for Computing Structural Test Data," in *Computational Logic*, ser. Lecture Notes in Computer Science, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds., vol. 1861. Springer, 2000, pp. 399–413.

[17] S. Bardin and A. Gotlieb, "FDCC: A Combined Approach for Solving Constraints over Finite Domains and Arrays," in *CPAIOR*, ser. Lecture Notes in Computer Science, N. Beldiceanu, N. Jussien, and E. Pinson, Eds., vol. 7298. Springer, 2012, pp. 17–33.

[18] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language," in *CP*, ser. Lecture Notes in Computer Science, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 529–543.

[19] V. Ganesh, A. Kiezun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A string solver for testing, analysis and vulnerability detection," in *CAV*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 1–19.