Eris Documentation

Release 0.7.0

Giorgio Sironi

Aug 20, 2017

Contents

Eris is a porting of QuickCheck and property-based testing tools to the PHP and PHPUnit ecosystem.

In property-based testing, several properties that the System Under Test must respect are defined, and a large sample of generated inputs is sent to it in an attempt to break the properties. With a few lines of code, hundreds of test cases can be generated and run.

> "Don't write tests. Generate them." – John Hughes

Eris is the Greek goddess of chaos, strife, and discord. It tries to break your code with the most random and chaotic input and actions.

CHAPTER 1

Installation

You can install Eris through Composer by running the following command in your terminal:

```
composer require --dev giorgiosironi/eris
```

You can run some of Eris example tests with `vendor/bin/phpunit vendor/giorgiosironi/eris/examples`.

Here is an empty sample project installing Eris.

CHAPTER 2

Getting started

This test tries to verify that natural numbers from 0 to 1000 are all smaller than 42. It's a failing test designed to show you an example of error message.

```php
<?php
use Eris\Generator;

class ReadmeTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testNaturalNumbersMagnitude()
    {
        $this->forAll(
            Generator\choose(0, 1000)
        )
            ->then(function ($number) {
                $this->assertTrue(
                    $number < 42,
                    "$number is not less than 42 apparently"
                );
            });
    }
}
```

Eris generates a sample of elements from the required domain (here the integers from 0 to 1000) and verifies a property on each of them, stopping at the first failure. Its functionalities are exported trough a `TestTrait` you can insert into your PHPUnit tests and through a series of functions in the `Eris\Generator` and `Eris\Listener` namespaces.

Generators implement the `Eris\Generator` interface, and provide random generation of values conforming to some types or domains. By combining them, your System Under Test can receive hundreds of different inputs with only a few lines of code.

Given that the input is unknown when writing the test, we have to test predicates over the result or the state of the System Under Test instead of writing equality assertions over the output. Properties should always be true, so that their violation indicates a bug and hence a failing test.

```
[10:34:32][giorgio@Bipbip:~/code/eris]$ vendor/bin/phpunit examples/ReadmeTest.php
PHPUnit 4.3.5 by Sebastian Bergmann.

Configuration read from /home/giorgio/code/eris/phpunit.xml

F

Time: 234 ms, Memory: 3.25Mb

There was 1 failure:

1) ReadmeTest::testNaturalNumbersMagnitude
42 is not less than 42 apparently
Failed asserting that false is true.

/home/giorgio/code/eris/examples/ReadmeTest.php:15
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:48
/home/giorgio/code/eris/src/Eris/Quantifier/RoundRobinShrinking.php:45
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:69
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:50
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:71
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:87
/home/giorgio/code/eris/examples/ReadmeTest.php:16
/home/giorgio/code/eris/examples/ReadmeTest.php:16

FAILURES!
Tests: 1, Assertions: 826, Failures: 1.
```

Eris also tries to *shrink* the input after a failure, giving you the simplest input that still fails the test. In this example, the original input was probably something like `562`, but Eris tries to make it smaller until the test became green again. The smallest value that still fails the test is the one presented to you.

CHAPTER 3

Shrinking

When one of the generated examples makes a test fail, it is useful for debugging purposes to try and generate the simplest possible input that still triggers this failure.

Eris, like all QuickCheck implementations, performs a process called shrinking which:

- stops each test at the first failure
- asks the Generator to turn the currently generated value to another, simpler value
- perform the test with the new value.

Shrinking repeats this process until the test does not fail anymore, or the value cannot be simplified further. The last input in the shrinking sequence that still makes the test fail is the one reported to the user, while all other values are regarded as more complex and thrown away.

Simplest example

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class ShrinkingTest extends \PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testShrinkingAString()
    {
        $this->forAll(
                Generator\string()
            )
            ->then(function ($string) {
                var_dump($string);
                $this->assertNotContains('B', $string);
            });
```

```
    }

    public function testShrinkingRespectsAntecedents()
    {
        $this->forAll(
                Generator\choose(0, 20)
            )
            ->when(function ($number) {
                return $number > 10;
            })
            ->then(function ($number) {
                $this->assertTrue($number % 29 == 0, "The number $number is not␣
↪multiple of 29");
            });
    }
}
```

`testShrinkingAString` is the simplest shrinking example. Each iteration generates random strings and test them to check that they do not contain the letter `B`. This is an example sequence of generated values (which by default will change at every run):

```
string(0) ""
string(1) "K"
string(2) "g,"
string(3) "=%,"
string(7) "jGHr38i"
string(15) "L(uw^K)/&hf!mQK"
string(9) ":W}W[+<GR"
string(20) ":e|$dI,[Bj(Kx-4`-"3X"
string(19) ":e|$dI,[Bj(Kx-4`-"3"
string(18) ":e|$dI,[Bj(Kx-4`-""
string(17) ":e|$dI,[Bj(Kx-4`-"
string(16) ":e|$dI,[Bj(Kx-4`"
string(15) ":e|$dI,[Bj(Kx-4"
string(14) ":e|$dI,[Bj(Kx-"
string(13) ":e|$dI,[Bj(Kx"
string(12) ":e|$dI,[Bj(K"
string(11) ":e|$dI,[Bj("
string(10) ":e|$dI,[Bj"
string(9) ":e|$dI,[B"
string(8) ":e|$dI,["
```

All the values up to `string(9) ":W}W[+<GR"` pass the test. The value `string(20) ":e|$dI,[Bj(Kx-4`-"3X"` is the first to fail.

From there, the value is shrunk by chopping away a single character at the end of the string. The value `string(8) ":e|$dI,["` is the first one in the shrinking sequence that does not fail the test, so the process stops there. The last simplified value to still fail the test is `string(9) ":e|$dI,[B"` and it's the one presented to the user:

```
1) ShrinkingTest::testShrinkingAString
Failed asserting that ':e|$dI,[B' does not contain "B".

/home/giorgio/code/eris/examples/ShrinkingTest.php:16
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
```

```
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/ShrinkingTest.php:17
/home/giorgio/code/eris/examples/ShrinkingTest.php:17

FAILURES!
Tests: 1, Assertions: 119, Failures: 1.
```

Shrinking and preconditions

`testShrinkingRespectsAntecedents` generates a random number from 0 to 20 and tries to check that it is multiple of 29. All generated numbers will fail this test, but shrinking will try to present the lowest possible number; still, the `when()` antecedent has to be satisfied and so the number cannot decrease down to 0 but has to stop at 11:

```
1) ShrinkingTest::testShrinkingRespectsAntecedents
The number 11 is not multiple of 29
Failed asserting that false is true.

/home/giorgio/code/eris/examples/ShrinkingTest.php:18
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/ShrinkingTest.php:19
/home/giorgio/code/eris/examples/ShrinkingTest.php:19

FAILURES!
Tests: 1, Assertions: 4, Failures: 1.
```

Shrinking is only performed when assertions fail: generic exceptions bubbling up out of the `then()` will just interrupt the test.

Shrinking time limit

You can set a time limit for shrinking if you prefer to be presented with more complex examples with respect to spending test suite running time:

```php
<?php
use Eris\Generator;
use Eris\TestTrait;

class ShrinkingTest extends \PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testShrinkingAString()
    {
        $this->forAll(
                Generator\string()
            )
            ->then(function ($string) {
                var_dump($string);
```

```
                $this->assertNotContains('B', $string);
            });
    }

    public function testShrinkingRespectsAntecedents()
    {
        $this->forAll(
                Generator\choose(0, 20)
            )
            ->when(function ($number) {
                return $number > 10;
            })
            ->then(function ($number) {
                $this->assertTrue($number % 29 == 0, "The number $number is not␣
↪multiple of 29");
            });
    }
}
```

The shrinking for this test will not run for more than 2 seconds (although the test as a whole may take more):

```
1) ShrinkingTimeLimitTest::testLengthPreservation
RuntimeException: Eris has reached the time limit for shrinking (2s elapsed of 2s),␣
↪here it is presenting the simplest failure case.
If you can afford to spend more time to find a simpler failing input, increase it␣
↪with $this->shrinkingTimeLimit($seconds).

/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:71
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/ShrinkingTimeLimitTest.php:32
/home/giorgio/code/eris/examples/ShrinkingTimeLimitTest.php:32

Caused by
PHPUnit_Framework_ExpectationFailedException: Concatenating 'hW4N*:fD0&%+D_' to 'p:\(,
↪N\7A6' gives 'hW4N*:fD0&%+D_p:\(,N\7A6ERROR'

Failed asserting that 29 matches expected 24.

/home/giorgio/code/eris/examples/ShrinkingTimeLimitTest.php:31
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/ShrinkingTimeLimitTest.php:32
/home/giorgio/code/eris/examples/ShrinkingTimeLimitTest.php:32

FAILURES!
Tests: 1, Assertions: 8, Errors: 1.
```

Tree-based shrinking

- for some generators what goes on under the hood is not a linear shrinking, write test that demonstrates that with Sample class

– optimistic path – pessimistic path – average path (choose the middle)

Disabling shrinking

In some cases the `then()` method is non-deterministic as it spawns other processes or talks to other services. Moreover, `then()` can be very slow to execute when targeting APIs for end-to-end tests. Finally, if it performs any cleanup executing it for shrinking may clean lods or databases traces from the actual test failure, preventing effective debugging.

Therefore, it is possible to configure Eris to disable the shrinking process. As a result, the first assertion failure will stop the test and let the exception bubble up:

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class DisableShrinkingTest extends \PHPUnit_Framework_TestCase
{
    use TestTrait;

    /**
     * Shrinking may be avoided when then() is slow or non-deterministic.
     */
    public function testThenIsNotCalledMultipleTime()
    {
        $this->calls = 0;
        $this
            ->forAll(
                Generator\nat()
            )
            ->disableShrinking()
            ->then(function ($number) {
                $this->calls++;
                $this->assertTrue(false, "Total calls: {$this->calls}");
            });
    }
}
```

This test will show a failure message containing `Total calls: 1`.

CHAPTER 4

Scalar generators

Integers

Integers can be generated, and by default they can be positive, or negative. You can force the sign of a number with:

- `Generator\nat()` which produces an integer >= 0.
- `Generator\pos()` which produces an integer > 0.
- `Generator\neg()` which produces an integer < 0.
- `Generator\byte()` which produces an integer >= 0 and <= 255.

```php
<?php
use Eris\Generator;
use Eris\TestTrait;

class IntegerTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testSumIsCommutative()
    {
        $this->forAll(
            Generator\int(),
            Generator\int()
        )
            ->then(function ($first, $second) {
                $x = $first + $second;
                $y = $second + $first;
                $this->assertEquals(
                    $x,
                    $y,
                    "Sum between {$first} and {$second} should be commutative"
                );
            });
```

```
    }

    public function testSumIsAssociative()
    {
        $this->forAll(
            Generator\int(),
            Generator\neg(),
            Generator\pos()
        )
            ->then(function ($first, $second, $third) {
                $x = $first + ($second + $third);
                $y = ($first + $second) + $third;
                $this->assertEquals(
                    $x,
                    $y,
                    "Sum between {$first} and {$second} should be associative"
                );
            });
    }

    public function testByteData()
    {
        $this->forAll(
            Generator\byte()
        )
            ->then(function ($byte) {
                $this->assertTrue(
                    $byte >= 0 && $byte <= 255,
                    "$byte is not a valid value for a byte"
                );
            });
    }
}
```

For more precise and custom ranges, the `Generator\choose()` accepts a lower and upper bound for the interval to sample integers from.

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class ChooseTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testSumOfTwoIntegersFromBoundedRangesIsCommutative()
    {
        $this->forAll(
            Generator\choose(-1000, 430),
            Generator\choose(230, -30000)
        )
            ->then(function ($first, $second) {
                $x = $first + $second;
                $y = $second + $first;
                $this->assertEquals(
                    $x,
                    $y,
                    "Sum between {$first} and {$second} should be commutative"
```

```
                );
            });
    }
}
```

Floats

`Generator\float()` will produce a float value, which can be positive or negative. In this example, `testAPropertyHoldingOnlyForPositiveNumbers` fails very quickly.

```
<?php
use Eris\Generator;

class FloatTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testAPropertyHoldingForAllNumbers()
    {
        $this->forAll(Generator\float())
            ->then(function ($number) {
                $this->assertEquals(
                    0.0,
                    abs($number) - abs($number)
                );
            });
    }

    public function testAPropertyHoldingOnlyForPositiveNumbers()
    {
        $this->forAll(Generator\float())
            ->then(function ($number) {
                $this->assertTrue(
                    $number >= 0,
                    "$number is not a (loosely) positive number"
                );
            });
    }
}
```

Booleans

`Generator\bool()` produces a boolean, chosen between `true` and `false`. It is mostly useful in conjunction with other Generators.

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class BooleanTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;
```

```
    public function testBooleanValueIsTrueOrFalse()
    {
        $this->forAll(
            Generator\bool()
        )
            ->then(function ($boolValue) {
                $this->assertTrue(
                    ($boolValue === true || $boolValue === false),
                    "$boolValue is not true nor false"
                );
            });
    }
}
```

Strings

`Generator\string()` produces a string of arbitrary length. Only printable characters can be included in the string, which is UTF-8. Currently only ASCII characters between `0x33` and `0x126` are used.

```
<?php
use Eris\Generator;
use Eris\Listener;

function string_concatenation($first, $second)
{
    if (strlen($second) > 5) {
        $second .= 'ERROR';
    }
    return $first . $second;
}

class StringTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testRightIdentityElement()
    {
        $this
            ->forAll(
                Generator\string()
            )
            ->then(function ($string) {
                $this->assertEquals(
                    $string,
                    string_concatenation($string, ''),
                    "Concatenating '$string' to ''"
                );
            });
    }

    public function testLengthPreservation()
    {
        $this
            ->forAll(
                Generator\string(),
```

```
                Generator\string()
            )
            ->hook(Listener\log('/tmp/eris-string-shrinking.log'))
            ->then(function ($first, $second) {
                $result = string_concatenation($first, $second);
                $this->assertEquals(
                    strlen($first) + strlen($second),
                    strlen($result),
                    "Concatenating '$first' to '$second' gives '$result'" . PHP_EOL
                    . var_export($first, true) . PHP_EOL
                    . "strlen(): " . strlen($first) . PHP_EOL
                    . var_export($second, true) . PHP_EOL
                    . "strlen(): " . strlen($second) . PHP_EOL
                    . var_export($result, true) . PHP_EOL
                    . "strlen(): " . strlen($result) . PHP_EOL
                    . "First hex: " . var_export(bin2hex($first), true) . PHP_EOL
                    . "Second hex: " . var_export(bin2hex($second), true) . PHP_EOL
                    . "Result hex: " . var_export(bin2hex($result), true) . PHP_EOL
                );
            });
    }
}
```

See also:

For more complex use cases, try using a collection generator in conjunction with *char()*.

Characters

`Generator\char()` generates a character from the chosen charset, by default with a `utf-8` encoding. The only supported charset at the time of this writing is `basic-latin`.

```
<?php
use Eris\Generator;
use Eris\Antecedent as is;
use Eris\Antecedent as are;

class CharacterTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testLengthOfAsciiCharactersInPhp()
    {
        $this->forAll(
            Generator\char(['basic-latin'])
        )
            ->then(function ($char) {
                $this->assertLenghtIs1($char);
            });
    }

    public function testLengthOfPrintableAsciiCharacters()
    {
        $this->forAll(
            Generator\char(['basic-latin'])
        )
```

```
            ->when(is\printableCharacter())
            ->then(function ($char) {
                $this->assertFalse(ord($char) < 32);
            });
    }

    public function testMultiplePrintableCharacters()
    {
        $this
            ->minimumEvaluationRatio(0.1)
            ->forAll(
                Generator\char(['basic-latin']),
                Generator\char(['basic-latin'])
            )
            ->when(are\printableCharacters())
            ->then(function ($first, $second) {
                $this->assertFalse(ord($first) < 32);
                $this->assertFalse(ord($second) < 32);
            });
    }

    private function assertLenghtIs1($char)
    {
        $length = strlen($char);
        $this->assertEquals(
            1,
            $length,
            "'$char' is too long: $length"
        );
    }
}
```

`Generator\charPrintableAscii()` can also be used to limit the range of the character to the set of printable characters, from `0x32` to `0x76`.

Constants

`Generator\constant()` produces always the same value, which is the value used to initialize it. This Generator is useful for debugging and simplifying composite Generators in these occasions.

Often, as shown in `testUseConstantGeneratorImplicitly`, constant are automatically boxed in this Generator if used where a `Generator` instance would be required:

```
<?php

use Eris\Generator;

class ConstantTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testUseConstantGeneratorExplicitly()
    {
        $this
            ->forAll(
                Generator\nat(),
```

```
                Generator\constant(2)
            )
            ->then(function ($number, $alwaysTwo) {
                $this->assertTrue(($number * $alwaysTwo % 2) === 0);
            });
    }

    public function testUseConstantGeneratorImplicitly()
    {
        $this
            ->forAll(
                Generator\nat(),
                2
            )
            ->then(function ($number, $alwaysTwo) {
                $this->assertTrue(($number * $alwaysTwo % 2) === 0);
            });
    }
}
```

Elements

`Generator\elements()` produces a value randomly extracted from the specified array. Values can be specified as arguments or with a single, numeric array.

```
<?php
use Eris\Generator;

class ElementsTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testElementsOnlyProducesElementsFromTheGivenArguments()
    {
        $this->forAll(
            Generator\elements(1, 2, 3)
        )
            ->then(function ($number) {
                $this->assertContains(
                    $number,
                    [1, 2, 3]
                );
            });
    }

    /**
     * This means you cannot have a Elements Generator with a single element,
     * which is perfectly fine as if you have a single element this generator
     * is useless. Use Constant Generator instead
     */
    public function testElementsOnlyProducesElementsFromTheGivenArrayDomain()
    {
        $this->forAll(
            Generator\elements([1, 2, 3])
        )
```

```
            ->then(function ($number) {
                $this->assertContains(
                    $number,
                    [1, 2, 3]
                );
            });
    }


    public function testVectorOfElementsGenerators()
    {
        $this->forAll(
            Generator\vector(
                4,
                Generator\elements([2, 4, 6, 8, 10, 12])
            )
        )
            ->then(function ($vector) {
                $sum = array_sum($vector);
                $isEven = function ($number) {
                    return $number % 2 == 0;
                };
                $this->assertTrue(
                    $isEven($sum),
                    "$sum is not even, but it's the sum of the vector " . var_export(
↪$vector, true)
                );
            });
    }
}
```

`testVectorOfElementsGenerators` shows how to compose the Elements Generator into a *vector()* to build a vector of selected, sometimes repeated, elements.

See also:

oneOf() does the same with values instead of Generators.

CHAPTER 5

Collection generators

Collection-oriented Generators produce arrays conforming to different constraints depending on the mathematical definition the reproduce. All these Generators require as an input one or more Generators to be used to produce single elements.

Associative arrays

Associative arrays can be generated composing other generators for each of the keys of the desired array, which will contain the specified fixed set of keys and vary the values.

```php
<?php
use Eris\Generator;

class AssociativeArrayTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testAssociativeArraysGeneratedOnStandardKeys()
    {
        $this->forAll(
            Generator\associative([
                'letter' => Generator\elements("A", "B", "C"),
                'cipher' => Generator\choose(0, 9),
            ])
        )
            ->then(function ($array) {
                $this->assertEquals(2, count($array));
                $letter = $array['letter'];
                $this->assertInternalType('string', $letter);
                $cipher = $array['cipher'];
                $this->assertInternalType('integer', $cipher);
            });
    }
}
```

Sequences

Sequences are defined as numeric arrays with a variable amount of elements of a single type. Both the length of the array and its values will be randomly generated.

```php
<?php
use Eris\Generator;
use Eris\Listener;

class SequenceTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testArrayReversePreserveLength()
    {
        $this
            ->forAll(
                Generator\seq(Generator\nat())
            )
            ->then(function ($array) {
                $this->assertEquals(count($array), count(array_reverse($array)));
            });
    }

    public function testArrayReverse()
    {
        $this
            ->forAll(
                Generator\seq(Generator\nat())
            )
            ->then(function ($array) {
                $this->assertEquals($array, array_reverse(array_reverse($array)));
            });
    }

    public function testArraySortingIsIdempotent()
    {
        $this
            ->forAll(
                Generator\seq(Generator\nat())
            )
            ->then(function ($array) {
                sort($array);
                $expected = $array;
                sort($array);
                $this->assertEquals($expected, $array);
            });
    }
}
```

Vectors

Vectors are defined as numeric arrays with a fixed amount of elements of a single type. Only the values contained will be randomly generated.

As an example, consider vectors inside a fixed space such as the set of 2D or 3D points.

```
<?php
use Eris\Generator;

class VectorTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testConcatenationMaintainsLength()
    {
        $this->forAll(
            Generator\vector(10, Generator\nat(1000)),
            Generator\vector(10, Generator\nat(1000))
        )
            ->then(function ($first, $second) {
                $concatenated = array_merge($first, $second);
                $this->assertEquals(
                    count($concatenated),
                    count($first) + count($second),
                    var_export($first, true) . " and " . var_export($second, true) .
↪" do not maintain their length when concatenated."
                );
            });
    }
}
```

Tuples

Tuples are defined as a small array of fixed size, consiting of a few heteregeneous types.

```
<?php
use Eris\Generator;

class TupleTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testConcatenationMaintainsLength()
    {
        $this->forAll(
            Generator\tuple(
                Generator\elements("A", "B", "C"),
                Generator\choose(0, 9)
            )
        )
            ->then(function ($tuple) {
                $letter = $tuple[0];
                $cipher = $tuple[1];
                $this->assertEquals(
```

```
                    2,
                    strlen($letter . $cipher),
                    "{$letter}{$cipher} is not a 2-char string"
                );
            });
    }
}
```

Sets

Sets are defined as array with a variable amount of elements of a single type, without any repeated element.

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class SetTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testSetsOfAnotherGeneratorsDomain()
    {
        $this->forAll(
            Generator\set(Generator\nat())
        )
            ->then(function ($set) {
                $this->assertInternalType('array', $set);
                foreach ($set as $element) {
                    $this->assertGreaterThanOrEqual(0, $element);
                }
            });
    }
}
```

Subsets

Subsets are set whose elements are extracted from a fixed universe set, specified as an input.

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class SubsetTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testSubsetsOfASet()
    {
        $this->forAll(
            Generator\subset([
                2, 4, 6, 8, 10
            ])
        )
```

```
            ->then(function ($set) {
                $this->assertInternalType('array', $set);
                foreach ($set as $element) {
                    $this->assertTrue($this->isEven($element), "Element $element is␣
↪not even, where did it come from?");
                }
                var_dump($set);
            });
    }

    private function isEven($number)
    {
        return $number % 2 == 0;
    }
}
```

CHAPTER 6

Composite generators

These Generators implement the Composite pattern to wire together existing Generators and callables.

Frequency

`Generator\frequency` randomly chooses a Generator to use from the specified list, weighting the probability of each Generator with the provided value.

```
<?php
use Eris\Generator;

class FrequencyTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testFalsyValues()
    {
        $this
            ->forAll(
                Generator\frequency(
                    [8, false],
                    [4, 0],
                    [4, '']
                )
            )
            ->then(function ($falsyValue) {
                $this->assertFalse((bool) $falsyValue);
            });
    }

    public function testAlwaysFails()
    {
        $this
            ->forAll(
```

```
                Generator\frequency(
                    [8, Generator\choose(1, 100)],
                    [4, Generator\choose(100, 200)],
                    [4, Generator\choose(200, 300)]
                )
            )
            ->then(function ($element) {
                $this->assertEquals(0, $element);
            });
    }
}
```

`testFalsyValues` chooses the `false` value half of the times, `0` one quarter of the time, and `''` one quarte of the time.

`testAlwaysFails` chooses the Generator from 1 to 100 half of the times. However, in case of failure it will try to shrink the value only with the original Generator that created it. Therefore, each of the possible outputs will be possible:

```
Failed asserting that 1 matches expected 0.
Failed asserting that 100 matches expected 0.
Failed asserting that 200 matches expected 0.
```

One Of

`Generator\oneOf` is a special case of `Generator\frequency` which selects each of the specified Generators with the same probability.

```
<?php
use Eris\Generator;

class OneOfTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testPositiveOrNegativeNumberButNotZero()
    {
        $this
            ->forAll(
                Generator\oneOf(
                    Generator\pos(),
                    Generator\neg()
                )
            )
            ->then(function ($number) {
                $this->assertNotEquals(0, $number);
            });
    }
}
```

See also:

elements() does the same with values instead of Generators.

Map

Map allows a Generator's output to be modified by applying the callable to the generated value.

```
<?php
use Eris\Generator;

class MapTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testApplyingAFunctionToGeneratedValues()
    {
        $this->forAll(
            Generator\vector(
                3,
                Generator\map(
                    function ($n) {
                        return $n * 2;
                    },
                    Generator\nat()
                )
            )
        )
            ->then(function ($tripleOfEvenNumbers) {
                foreach ($tripleOfEvenNumbers as $number) {
                    $this->assertTrue(
                        $number % 2 == 0,
                        "The element of the vector $number is not even"
                    );
                }
            });
    }

    public function testShrinkingJustMappedValues()
    {
        $this->forAll(
            Generator\map(
                function ($n) {
                    return $n * 2;
                },
                Generator\nat()
            )
        )
            ->then(function ($evenNumber) {
                $this->assertLessThanOrEqual(
                    100,
                    $evenNumber,
                    "The number is not less than 100"
                );
            });
    }

    public function testShrinkingMappedValuesInsideOtherGenerators()
    {
        $this->forAll(
            Generator\vector(
                3,
```

```
                Generator\map(
                    function ($n) {
                        return $n * 2;
                    },
                    Generator\nat()
                )
            )
        )
            ->then(function ($tripleOfEvenNumbers) {
                $this->assertLessThanOrEqual(
                    100,
                    array_sum($tripleOfEvenNumbers),
                    "The triple sum " . var_export($tripleOfEvenNumbers, true) . " is not less than 100"
                );
            });
    }

    // TODO: multiple generators means multiple values passed to map
}
```

`testApplyingAFunctionToGeneratedValues` generates a vector of even numbers. Notice that any mapping can still be composed by other Generators: in this case, the even number Generator can be composed by `Generator\vector()`,

`testShrinkingJustMappedValues` shows how shrinking respects the mapping function: running this test produces 102 as the minimal input that still makes the assertion fail. The underlying `Generator\nat()` shrinks number by decrementing them, but the mapping function is still applied so that only even numbers are passed to the `then()`.

```
1) MapTest::testShrinkingJustMappedValues
The number is not less than 100
Failed asserting that 102 is equal to 100 or is less than 100.

/home/giorgio/code/eris/examples/MapTest.php:42
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/MapTest.php:43
/home/giorgio/code/eris/examples/MapTest.php:43

FAILURES!
Tests: 1, Assertions: 254, Failures: 1.
```

`testShrinkingMappedValuesInsideOtherGenerators` puts both examples together and generates a triple of even numbers, failing the test if their sum is greater than 100. The minimal failing example is a triple of number whose sum is 102.

```
1) MapTest::testShrinkingMappedValuesInsideOtherGenerators
The triple sum array (
  0 => 52,
  1 => 36,
  2 => 14,
) is not less than 100
Failed asserting that 102 is equal to 100 or is less than 100.
```

```
/home/giorgio/code/eris/examples/MapTest.php:62
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/MapTest.php:63
/home/giorgio/code/eris/examples/MapTest.php:63

FAILURES!
Tests: 1, Assertions: 216, Failures: 1.
```

Such That

Such That allows a Generator's output to be filtered, excluding values that to do not satisfy a condition.

```php
<?php
use Eris\Generator;
use Eris\Listener;

class SuchThatTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testSuchThatBuildsANewGeneratorFilteringTheInnerOne()
    {
        $this
            ->forAll(
                Generator\vector(
                    5,
                    Generator\suchThat(
                        function ($n) {
                            return $n > 42;
                        },
                        Generator\choose(0, 1000)
                    )
                )
            )
            ->then($this->allNumbersAreBiggerThan(42));
    }

    public function testFilterSyntax()
    {
        $this
            ->forAll(
                Generator\vector(
                    5,
                    Generator\filter(
                        function ($n) {
                            return $n > 42;
                        },
                        Generator\choose(0, 1000)
                    )
                )
            )
            ->then($this->allNumbersAreBiggerThan(42));
```

```
    }

    public function testSuchThatAcceptsPHPUnitConstraints()
    {
        $this
            ->forAll(
                Generator\vector(
                    5,
                    Generator\suchThat(
                        $this->isType('integer'),
                        Generator\oneOf(
                            Generator\choose(0, 1000),
                            Generator\string()
                        )
                    )
                )
            )
            ->hook(Listener\log('/tmp/eris-such-that.log'))
            ->then($this->allNumbersAreBiggerThan(42));
    }


    public function testSuchThatShrinkingRespectsTheCondition()
    {
        $this
            ->forAll(
                Generator\suchThat(
                    function ($n) {
                        return $n > 42;
                    },
                    Generator\choose(0, 1000)
                )
            )
            ->then($this->numberIsBiggerThan(100));
    }

    public function␣
↪testSuchThatShrinkingRespectsTheConditionButTriesToSkipOverTheNotAllowedSet()
    {
        $this
            ->forAll(
                Generator\suchThat(
                    function ($n) {
                        return $n <> 42;
                    },
                    Generator\choose(0, 1000)
                )
            )
            ->then($this->numberIsBiggerThan(100));
    }

    public function␣
↪testSuchThatAvoidingTheEmptyListDoesNotGetStuckOnASmallGeneratorSize()
    {
        $this
            ->forAll(
                Generator\suchThat(
                    function (array $ints) {
```

```
                        return count($ints) > 0;
                    },
                    Generator\seq(Generator\int())
                )
            )
            ->then(function (array $ints) use (&$i) {
                $this->assertGreaterThanOrEqual(1, count($ints));
            })
        ;
    }

    public function allNumbersAreBiggerThan($lowerLimit)
    {
        return function ($vector) use ($lowerLimit) {
            foreach ($vector as $number) {
                $this->assertTrue(
                    $number > $lowerLimit,
                    "\$number was asserted to be more than $lowerLimit, but it's
↪$number"
                );
            }
        };
    }

    public function numberIsBiggerThan($lowerLimit)
    {
        return function ($number) use ($lowerLimit) {
            $this->assertTrue(
                $number > $lowerLimit,
                "\$number was asserted to be more than $lowerLimit, but it's $number"
            );
        };
    }
}
```

`testSuchThatBuildsANewGeneratorFilteringTheInnerOne` generates a vector of numbers greater than 42. Notice that any filterting can still be composed by other Generators: in this case, the greater-than-42 number Generator can be composed by `Generator\vector()`,

`testFilterSyntax` shows the `Generator\filter()` syntax, which is just an alias for `Generator\suchThat()`. The order of the parameters requires to pass the callable first, for consistency with `Generator\map()` and in opposition to `array_filter`.

`testSuchThatAcceptsPHPUnitConstraints` shows that you can pass in PHPUnit constraints in lieu of callables, in the same way as they are passed to `assertThat()`, or to `with()` when defining PHPUnit mock expectations.

`testSuchThatShrinkingRespectsTheCondition` shows that shrinking takes into account the callable and stops when it is not satisfied anymore. Therefore, this test will fail for all numbers lower than or equal to 100, but the minimum example found is 43 as it's the smallest and simplest value that still satisfied the condition.

```
1) SuchThatTest::testSuchThatShrinkingRespectsTheCondition
$number was asserted to be more than 100, but it's 43
Failed asserting that false is true.

/home/giorgio/code/eris/examples/SuchThatTest.php:85
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
```

```
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/SuchThatTest.php:34
/home/giorgio/code/eris/examples/SuchThatTest.php:34
```

`testSuchThatShrinkingRespectsTheConditionButTriesToSkipOverTheNotAllowedSet` shows instead how shrinking does not give up easily, but shrinks the inner generator even more to see is simpler values may still satisfy the condition of being different from 42. Therefore, the test fails with the shrunk input 0, not 43 as before:

```
1) SuchThatTest::testSuchThatShrinkingRespectsTheConditionButTriesToSkipOverTheNotAllowedSet
$number was asserted to be more than 100, but it's 0
Failed asserting that false is true.

/home/giorgio/code/eris/examples/SuchThatTest.php:85
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:130
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:158
/home/giorgio/code/eris/examples/SuchThatTest.php:47
/home/giorgio/code/eris/examples/SuchThatTest.php:47
```

Bind

Bind allows a Generator's output to be used as an input to create another Generator. This composition allows to create several random values which are correlated with each other, by using the same input for their Generators parameters.

For example, here's how to create a vector along with a random element chosen by it.

```php
<?php
use Eris\Generator;

class BindTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testCreatingABrandNewGeneratorFromAGeneratedValueSingle()
    {
        $this->forAll(
            Generator\bind(
                Generator\vector(4, Generator\nat()),
                function ($vector) {
                    return Generator\tuple(
                        Generator\elements($vector),
                        Generator\constant($vector)
                    );
                }
            )
        )
            ->then(function ($tuple) {
```

```
                list($element, $vector) = $tuple;
                $this->assertContains($element, $vector);
            });
    }

    // TODO: multiple generators means multiple values passed to the
    // outer Generator factory
}
```

CHAPTER 7

Domain-based generators

Some default Generators target a particular business domain. They can be useful to test applications with plausible data instead of with universal values coming from a mathematical set like natural numbers or all the possible strings.

Names

Person names can be generated by selecting random elements from a dataset stored inside Eris source code.

```
<?php
use Eris\Generator;

class NamesTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testGeneratingNames()
    {
        $this->forAll(
            Generator\names()
        )->then(function ($name) {
            $this->assertInternalType('string', $name);
            var_dump($name);
        });
    }

    public function testSamplingShrinkingOfNames()
    {
        $generator = Generator\NamesGenerator::defaultDataSet();
        $sample = $this->sampleShrink($generator);
        $this->assertInternalType('array', $sample->collected());
        var_dump($sample->collected());
    }
}
```

`testGeneratingNames` shows a list of sample generated names. Their length increase with the size passed to Generators:

```
string(0) ""
string(0) ""
string(3) "Ita"
string(6) "Teresa"
string(8) "Raimunde"
string(7) "Laelius"
string(5) "Fanny"
string(6) "Aileen"
string(11) "Marie-Elise"
string(7) "Ignacio"
string(8) "Hendrick"
```

`testSamplingShrinkingOfNames` shows how names are shrinked to the slightly shorter name in the data set that is more similar to the current value:

```
array(8) {
  [0]=>
  string(9) "Gwenaelle"
  [1]=>
  string(8) "Ganaelle"
  [2]=>
  string(7) "Anaelle"
  [3]=>
  string(6) "Abelle"
  [4]=>
  string(5) "Abele"
  [5]=>
  string(4) "Abel"
  [6]=>
  string(3) "Abe"
  [7]=>
  string(2) "Di"
}
```

Dates

The `date()` Generator produces uniformly distributed DateTime objects.

```
<?php
use Eris\Generator;

class DateTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testYearOfADate()
    {
        $this->forAll(
            Generator\date("2014-01-01T00:00:00", "2014-12-31T23:59:59")
        )
            ->then(function (DateTime $date) {
                $this->assertEquals(
                    "2014",
```

```
                    $date->format('Y')
                );
            });
    }

    public function testDefaultValuesForTheInterval()
    {
        $this->forAll(
            Generator\date()
        )
            ->then(function (DateTime $date) {
                $this->assertGreaterThanOrEqual(
                    "1970",
                    $date->format('Y')
                );
                $this->assertLessThanOrEqual(
                    "2038",
                    $date->format('Y')
                );
            });
    }

    public function testFromDayOfYearFactoryMethodRespectsDistanceBetweenDays()
    {
        $this->forAll(
            Generator\choose(2000, 2020),
            Generator\choose(0, 364),
            Generator\choose(0, 364)
        )
        ->then(function ($year, $dayOfYear, $anotherDayOfYear) {
            $day = fromZeroBasedDayOfYear($year, $dayOfYear);
            $anotherDay = fromZeroBasedDayOfYear($year, $anotherDayOfYear);
            $this->assertEquals(
                abs($dayOfYear - $anotherDayOfYear) * 86400,
                abs($day->getTimestamp() - $anotherDay->getTimestamp()),
                "Days of the year $year: $dayOfYear, $anotherDayOfYear" . PHP_EOL
                . "{$day->format(DateTime::ISO8601)}, {$anotherDay->
↪format(DateTime::ISO8601)}"
            );
        });
    }
}

function fromZeroBasedDayOfYear($year, $dayOfYear)
{
    return DateTime::createFromFormat(
        'z Y H i s',
        $dayOfYear . ' '. $year . ' 00 00 00',
        new DateTimeZone("UTC")
    );
}
```

`testYearOfADate` shows how to specify the lower and upper bound of an interval to pick dates from. These bounds are included in the interval.

`testDefaultValuesForTheInterval` shows that by default, given the 32-bit random generators used as a source, dates span the 1970-2038 interval of 32-bit UNIX timestamps.

`testFromDayOfYearFactoryMethodRespectsDistanceBetweenDays` uses the

:ref:`choose()<choose>` Generator to pick directly integers and build `DateTime` objects itself. The test demonstrates a bug in the `datetime` PHP extension when an off-by-one error can be introduced when dealing with leap years.

Regex

The `regex()` Generator attempts to build a string matching the specified regular expression. It can be used to produce input strings much more close to a plausible format than totally random values.

```
<?php
use Eris\Generator;

class RegexTest extends \PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    /**
     * Note that * and + modifiers are not supported. @see Generator\regex
     */
    public function testStringsMatchingAParticularRegex()
    {
        $this->forAll(
            Generator\regex("[a-z]{10}")
        )
            ->then(function ($string) {
                $this->assertEquals(10, strlen($string));
            });
    }
}
```

Here is a sample of generated values from `testStringsMatchingAParticularRegex`:

```
string(10) "ylunkcebou"
string(10) "whkjewwhud"
string(10) "pwirjzhbdw"
string(10) "dxsdwnsmyi"
string(10) "ttgczpimxs"
string(10) "jdsmlexlau"
```

CHAPTER 8

Runtime limits

Time and iterations

By default Eris extracts a sample of 100 values for each `forAll()` call, and runs the `then()` callback over each of them.

For tests which take very long to run, it is possible to either limit the number of elements in the sample, or to specify a time limit the test should not exceed. For this purpose, the `limitTo()` method accepts either:

- an integer requesting a fixed number of iterations;
- a DateInterval object from the standard PHP library.

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class LimitToTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testNumberOfIterationsCanBeConfigured()
    {
        $this->limitTo(5)
             ->forAll(
                 Generator\int()
            )
            ->then(function ($value) {
                $this->assertInternalType('integer', $value);
            });
    }

    /*
     * future feature
    public function␣
↪testTimeIntervalToRunForCanBeConfiguredButItNeedsToProduceAtLeastHalfOfTheIterationsByDefault()
```

```
    {
        $this->minimum(10)
             ->limitTo(new DateInterval("PT2S"))
             ->forAll(
                Generator\int()
            )
            ->then(function($value) {
                usleep(100 * 1000);
                $this->assertTrue(true);
            });
    }
     */

    public function␣
↪testTimeIntervalToRunForCanBeConfiguredAndAVeryLowNumberOfIterationsCanBeIgnored()
    {
        $this->minimumEvaluationRatio(0.0)
             ->limitTo(new DateInterval("PT2S"))
             ->forAll(
                Generator\int()
            )
            ->then(function ($value) {
                usleep(100 * 1000);
                $this->assertTrue(true);
            });
    }
}
```

In the first example, the test is stopped after 5 generations.

The second example is about a future feature, not implemented yet, which will make it possible to specify a time limit while requiring a minimum number of operations.

In the third example, a time limit of 2 seconds is specified. Whenever a new element has to be added to the sample, the time limit is checked to see if the elapsed time from the start of the test has exceeded it.

Since it is possible for the generation process to have some overhead, the time specified is not an hard limit but will only be approximately respected. More precisely, the iteration running when the time limit is reached still has to be finished without being interrupted, along with any shrinking process derived from its potential failure.

Size of generated data

Many Generators accept a `size` parameter that should be used as an upper bound when creating new random elements. For example, this bound corresponds to a maximum positive integer, or to the maximum number of elements inside an array.

```
<?php
use Eris\Generator;
use Eris\TestTrait;

class SizeTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    /**
     * With the default sizes this test would pass,
```

```
     * as numbers greater or equal than 100,000 would never be reached.
     */
    public function testMaxSizeCanBeIncreased()
    {
        $this
            ->forAll(
                Generator\int()
            )
            ->withMaxSize(1000 * 1000)
            ->then(function ($number) {
                $this->assertLessThan(100 * 1000, $number);
            });
    }
}
```

By default size is equal to `1000`, which means no number greater than `1000` in absolute value will be generated. This test sets the maximum size to `1,000,0000`, and naturally fails when a number greater than `100,000` is picked and passed to the assertion. The failure message shows the shrunk input, exactly `100,000`:

```
There was 1 failure:

1) SizeTest::testMaxSizeCanBeIncreased
Failed asserting that 100000 is less than 100000.

/home/giorgio/code/eris/examples/SizeTest.php:21
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Shrinker/Random.php:68
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:126
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:53
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:156
/home/giorgio/code/eris/examples/SizeTest.php:22
/home/giorgio/code/eris/examples/SizeTest.php:22
```

The maximum sizes that can be reached are also limited by the *underlying random number generator*.

CHAPTER 9

Reproducibility

Eris allows you to seed the pseudorandom number generator in order to attempt to reproduce the same test run and check if a previously found bug has now been fixed.

Consider this test:

This test will fail, no matter which value is generated. No shrinking will be performed as the selected Generator considers the elements of equal complexity.

When you run this test, you may obtain an output very similar to:

```
F                                                                   1 / 1 (100%)
Reproduce with:
ERIS_SEED=1458646953837419 vendor/bin/phpunit --filter␣
↪AlwaysFailsTest::testFailsNoMatterWhatIsTheInput


Time: 44 ms, Memory: 3.50Mb

There was 1 failure:

1) AlwaysFailsTest::testFailsNoMatterWhatIsTheInput
This test fails by design. 'd' was passed in

/home/giorgio/code/eris/examples/AlwaysFailsTest.php:15
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:156
/home/giorgio/code/eris/examples/AlwaysFailsTest.php:16
/home/giorgio/code/eris/examples/AlwaysFailsTest.php:16

FAILURES!
Tests: 1, Assertions: 0, Failures: 1.
```

If you take the suggested command line and execute it, you will see the same error message, selecting `d` as the random input:

```
$ ERIS_SEED=1458646953837419 vendor/bin/phpunit --filter␣
↪AlwaysFailsTest::testFailsNoMatterWhatIsTheInput
PHPUnit 5.0.9 by Sebastian Bergmann and contributors.

F                                                                   1 / 1 (100%)
Reproduce with:
ERIS_SEED=1458646953837419 vendor/bin/phpunit --filter␣
↪AlwaysFailsTest::testFailsNoMatterWhatIsTheInput


Time: 130 ms, Memory: 10.75Mb

There was 1 failure:

1) AlwaysFailsTest::testFailsNoMatterWhatIsTheInput
This test fails by design. 'd' was passed in

/home/giorgio/code/eris/examples/AlwaysFailsTest.php:15
/home/giorgio/code/eris/src/Eris/Quantifier/Evaluation.php:51
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:128
/home/giorgio/code/eris/src/Eris/Quantifier/ForAll.php:156
/home/giorgio/code/eris/examples/AlwaysFailsTest.php:16
/home/giorgio/code/eris/examples/AlwaysFailsTest.php:16

FAILURES!
Tests: 1, Assertions: 0, Failures: 1.
```

Running the test without a `ERIS_SEED` environment variable will restore the previous behavior, exploring the Generator space in search of brand new values.

CHAPTER 10

Listeners

Eris provide the possibility to pass in Listeners to be notified of events happening during a test run.

Listeners implement the `Eris\Listener` interface and are advised to extend the `Eris\EmptyListener` abstract base class to get an empty implementation for all the methods corresponding to events they don't need to listen to.

Consider that Eris performs (by default) 100 iterations for each `forAll()` instance, each corresponding to a different set of generated values. The following methods can be overridden to receive an event:

- `startPropertyVerification()` is called before the first iteration starts.
- `endPropertyVerification($ordinaryEvaluations, $iterations, Exception $exception = null)` is called when no more iterations will be performed, both in the case of test success and failure. The `$ordinaryEvaluations` parameter provides the actual number of evaluations performed. This number may be less than than the number of target `$iterations` due to failures or `when()` filters not being satisfied. The `$exception`, when not null, indicated that the test has finally failed and corresponds to the error that is actually bubbling up rather than the original, unshrunk error.
- `newGeneration(array $generation, $iteration)` is called after generating a new iteration, and is passed the tuple of values along with the 0-based index of the iteration.
- `failure(array $generation, Exception $e)` is called after the failure of an assertion (and not for generic exceptions). The method can be called only once per `then()` run, and is called before any shrinking takes place.
- `shrinking(array $generation)` is called before each shrinking attempt, with the values that will be used as the simplified input.

`$generation` is always an array of the same form as the arguments passed to `then()`, without any Eris class wrapping them.

Collect Frequencies

The `collectFrequencies()` Listener allows to gather all generated values in order to display their statistical distribution.

```php
<?php
use Eris\Generator;
use Eris\TestTrait;
use Eris\Listener;

class CollectTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testGeneratedDataCollectionOnScalars()
    {
        $this
            ->forAll(Generator\neg())
            ->hook(Listener\collectFrequencies())
            ->then(function ($x) {
                $this->assertTrue($x < $x + 1);
            });
    }

    public function testGeneratedDataCollectionOnMoreComplexDataStructures()
    {
        $this
            ->forAll(
                Generator\vector(2, Generator\int()),
                Generator\char()
            )
            ->hook(Listener\collectFrequencies())
            ->then(function ($vector) {
                $this->assertEquals(2, count($vector));
            });
    }

    public function testGeneratedDataCollectionWithCustomMapper()
    {
        $this
            ->forAll(
                Generator\seq(Generator\nat())
            )
            ->withMaxSize(10)
            ->hook(Listener\collectFrequencies(function ($array) {
                return count($array);
            }))
            ->then(function ($array) {
                $this->assertEquals(count($array), count(array_reverse($array)));
            });
    }
}
```

`testGeneratedDataCollectionOnScalars` collects integers:

```
12%  -1
6%  -2
4%  -5
4%  -18
4%  -11
3%  -4
...
```

`testGeneratedDataCollectionOnMoreComplexDataStructures` shows how by default more complex structures are encoded into a JSON value, to be used as the bin key in the map of values to counters:

```
1%  [[-19,-16],"m"]
1%  [[-3,-30],";"]
1%  [[-9,1],"\f"]
1%  [[-7,-1],"P"]
1%  [[-1,-9],"^"]
1%  [[1,18],"8"]
1%  [[-53,-1],"."]
...
```

`testGeneratedDataCollectionWithCustomMapper` shows how to provide a custom callable to map the generated values into a bin key. Arguments are passed to the callable in the same way as `then()`. In this example, we are discovering that 10% of the generated arrays have length 3.

```
39%  0
26%  1
10%  3
5%  4
5%  2
4%  5
3%  7
3%  6
3%  8
1%  10
1%  9
```

Log

The `log()` Listener allows to write a log file while particularly long tests are executing, showing the partial progress of the test.

```php
<?php
use Eris\Generator;
use Eris\TestTrait;
use Eris\Listener;

class LogFileTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testWritingIterationsOnALogFile()
    {
        $this
            ->forAll(
                Generator\int()
            )
            ->hook(Listener\log('/tmp/eris-log-file-test.log'))
            ->then(function ($number) {
                $this->assertInternalType('integer', $number);
            });
    }

    public function testLogOfFailuresAndShrinking()
    {
```

```
        $this
            ->forAll(
                Generator\int()
            )
            ->hook(Listener\log('/tmp/eris-log-file-shrinking.log'))
            ->then(function ($number) {
                $this->assertLessThanOrEqual(42, $number);
            });
    }
}
```

A file will be written during the test run with the following contents:

```
...
[2016-03-24T09:14:20+00:00][2593] iteration 12: [-9]
[2016-03-24T09:14:20+00:00][2593] iteration 13: [-59]
[2016-03-24T09:14:20+00:00][2593] iteration 14: [-51]
[2016-03-24T09:14:20+00:00][2593] iteration 15: [-52]
[2016-03-24T09:14:20+00:00][2593] iteration 16: [-83]
[2016-03-24T09:14:20+00:00][2593] iteration 17: [78]
[2016-03-24T09:14:20+00:00][2593] failure: [78]. Failed asserting that 78 is equal
to 42 or is less than 42.
[2016-03-24T09:14:20+00:00][2593] shrinking: [77]
[2016-03-24T09:14:20+00:00][2593] shrinking: [76]
[2016-03-24T09:14:20+00:00][2593] shrinking: [75]
[2016-03-24T09:14:20+00:00][2593] shrinking: [74]
[
...
```

It is not advised to rely on this format for parsing, being it only oriented to human readability.

Minimum Evaluations —

The `minimumEvaluations($ratio)` API method instantiates and wires in a Listener that checks that at least `$ratio` of the total number of inputs being generated is actually evaluated. This Listener is only needed in case of an aggressive use of `when()`.

Management of this Listener is provided through this method instead of explicitly adding a Listener object, as there is a default Listener instantiated with a threshold of 0.5 that has to be replaced in case a new minimum is chosen.

```
<?php
use Eris\Generator;

class MinimumEvaluationsTest extends PHPUnit_Framework_TestCase
{
    use Eris\TestTrait;

    public function testFailsBecauseOfTheLowEvaluationRatio()
    {
        $this
            ->forAll(
                Generator\choose(0, 100)
            )
            ->when(function ($n) {
                return $n > 90;
            })
            ->then(function ($number) {
                $this->assertTrue($number * 2 > 90 * 2);
            });
```

```
    }

    public function testPassesBecauseOfTheArtificiallyLowMinimumEvaluationRatio()
    {
        $this
            ->minimumEvaluationRatio(0.01)
            ->forAll(
                Generator\choose(0, 100)
            )
            ->when(function ($n) {
                return $n > 90;
            })
            ->then(function ($number) {
                $this->assertTrue($number * 2 > 90 * 2);
            });
    }
}
```

Both tests generate inputs in the range from 0 to 100, and since the condition of them being greater than 90 is rare, most of them will be discarded. By default Eris will check that 50% of the inputs are actually evaluated; therefore `testFailsBecauseOfTheLowEvaluationRatio` will fail with this message:

```
...
There was 1 error:

1) MinimumEvaluationsTest::testFailsBecauseOfTheLowEvaluationRatio
OutOfBoundsException: Evaluation ratio 0.05 is under the threshold 0.5
...
```

The actual ratio may vary depending on the inputs being generated and may not be `0.05`.

In *testPassesBecauseOfTheArtificiallyLowMinimumEvaluationRatio*, we accept a lower minimum evaluation ratio of 1%; therefore the test does not ordinarily fail. Its coverage will still be very poor, so the user is advised to precisely specify the inputs rather than generating a lot of them and discarding a large percentage with `when()`.

CHAPTER 11

Randomness

Eris allow multiple sources of randomness, with the requirements that they must accept a seed for reproducibility. Therefore, sequence Pseudo Random Number Generators (PRNG) are used instead of Cryptographically Secure PRNG, which would provide no additional value in generating test cases but make impossible to run the same test twice.

The supported random number generators are:

- the `rand` PHP function: this is the default, and simpler, choice.
- the `mt_rand` PHP function: this is a faster PRNG.
- the PHP code implementation `purePhpMtRand()` is equivalent to `mt_rand`.

Being implemented inside a PHP object, `purePhpMtRand()` allows to isolate its state while the first two implementations modify the global state of the PHP process. Use `purePhpMtRand()` when your code calls `rand()` or `mt_rand()` and you don't want it to interact with the testing framework.

Configuration

```php
<?php
use Eris\Generator;
use Eris\Random;
use Eris\TestTrait;

class RandConfigurationTest extends PHPUnit_Framework_TestCase
{
    use TestTrait;

    public function testUsingTheDefaultRandFunction()
    {
        $this
            ->withRand('rand')
            ->forAll(
                Generator\int()
```

```
            )
            ->withMaxSize(1000 * 1000* 1000)
            ->then($this->isInteger());
    }

    public function testUsingTheDefaultMtRandFunction()
    {
        $this
            ->withRand('mt_rand')
            ->forAll(
                Generator\int()
            )
            ->then($this->isInteger());
    }

    public function testUsingThePurePhpMtRandFunction()
    {
        if (defined('HHVM_VERSION')) {
            $this->markTestSkipped('MersenneTwister class does not support HHVM');
        }

        $this
            ->withRand(Random\purePhpMtRand())
            ->forAll(
                Generator\int()
            )
            ->then($this->isInteger());
    }

    private function isInteger()
    {
        return function ($number) {
            $this->assertInternalType('integer', $number);
        };
    }
}
```

`testUsingTheDefaultRandFunction` specifies the `rand` variant, but is equivalent to not calling `withRand()` at all. `srand` is the corresponding seed function.

`testUsingTheDefaultMtRandFunction` configured `mt_rand` and `mt_srand` as its seed function.

`testUsingThePurePhpMtRandFunction` configures `purePhpMtRand()`.

Maximum sizes

The size that can be set and actually reached with `withMaxSize()` is limited by the chosen PRNG.

- For `rand` the maximum values is the result of `getrandmax()`, which is platform dependent but usually 2^31-1.
- For `mt_rand` the maximum value is the result of `mt_getrandmax()`, which is also platform dependent but usually 2^31-1.
- For `purePhpMtRand()`, being implemented in PHP code, the maximum value is 2^32-1.

The limitation on size depends not only on the processor architecture but also on the parameters of the algorithm. Both `mt_rand` and *purePhpMtRand()'* implement MT19937, which generates 32-bit integers that can be scaled on

any smaller interval.

However, according to the PHP source code, `mt_rand` implementations uses a lower limit for backward compatibility with `rand`. `purePhpMtRand()` has no need for backward compatibility and chooses to allow numbers up to 2^32-1, which is the maximum unsigned number representable with 32 bit.

Seeding

The PRNGS are seeded using the `microtime()` of the system, or with the `ERIS_SEED` environment variable for *test reproducibility*.

Comparison

Variant	Portability	Speed	Global state
`rand`	PHP core	Slow	Yes
`mt_rand`	PHP core	Fast	Yes
`purePhpMtRand()`	Eris source code	Medium	No

CHAPTER 12

Using Eris outside of PHPUnit

Eris can be reused as a library for (reproducibly) generating random data, outside of PHPUnit test cases. For example, it may be useful in other testing frameworks or in scripts that run inside your testing infrastructure but not tied to a specific PHPUnit test suite.

Usage

```
<?php
use Eris\Generator;

require __DIR__.'/../vendor/autoload.php';

$eris = new Eris\Facade();
$eris
    ->forAll(Generator\int())
    ->then(function ($integer) {
        var_dump($integer);
    });
```

This script instantiates a `Eris\Facade`, which offers the same interface as `Eris\TestTrait`. `forAll()` is the main entry point and should be called over this object rather than `$this`.

The Facade is automatically initialized, and is used here to dump 100 random integers. At this time, reproducibility can be obtained by explicitly setting the *ERIS_SEED* environment variable.