# Random and Enumerative Testing for OCaml and WhyML Properties

Clotilde Erard and Alain Giorgetti

FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France

**Abstract.** We present AutoCheck, a new tool to test properties expressible in OCaml or WhyML language. The tool integrates random and enumerative testing. It relies on Why3 extraction mechanism to OCaml and an existing random testing tool for OCaml. An originality is that enumerative testing here uses data generators themselves written in the WhyML language of the Why3 deductive verification platform, and formally verified with this tool. Our tool and the paper are designed with simplicity and usability in mind, in order to make them accessible to the widest audience. Starting from the most elementary cases, many examples present all the tool features in increasing complexity order.

**Keywords:** property based testing · random testing · enumerative testing · deductive verification

## Note to reviewers

Dear Reviewers,

We have presented at IFIP-ICTSS 2019 a work entitled “Bounded Exhaustive Testing with Certified and Optimized Data Enumeration Programs” [22]. For the Special Issue of Software Quality Journal we first considered presenting an extension of that work, but it appeared to us that it was far less interesting and relevant than integrating our certified data generators into a testing tool for properties expressible in OCaml and WhyML languages.

Therefore the present paper presents this new testing tool, moves the subject of data generator certification to Section 5.2 and eludes the question of their optimization. Given that shift in the paper subject, it is up to you to consider the present paper either as an extension of [22], or as an external submission, also welcome for that special issue. As an extension, the main changes are:

1. an implementation of a new random and enumerative testing tool for WhyML language, based on an existing random testing tool for OCaml,
2. an enumerative testing tool for OCaml,
3. a generic interface between both enumerative testing tools and certified enumeration programs presented in [22].

# 1 Introduction

By proving that a given program respects a given functional specification, once for all its possible inputs, *deductive verification*, aka. program proof, provides a high level of confidence in software correctness. However, many obstacles limit the spread of deductive verification and its practice by development and validation engineers, despite the existence of numerous deductive verification tools.

The first obstacle is the formalization of specifications. Their writing has become easier thanks to *behavioral interface specification languages*, aka. *contract languages*, that are close to programming languages and are integrated in code as formal comments, named *annotations*. Examples of contract languages are JML for Java [31], ACSL for C [12] and Spec# for C# [11]. Deductive verification tools – such as KeY [13] for Java/JML, the WP plugin [18] of Frama-C for C/ACSL or Boogie [6] for Spec# /C# – then reduce annotated code to logical formulas, named *verification conditions*, whose validity entails conformance between the code and its specification.

Unfortunately, the complexity of main-stream programming languages often leads to verification conditions that are too difficult to be automatically proved by deductive verification tools. A good strategy is to write specification and programs in the language of a tool dedicated to deductive verification, such as Why3 [15], which optimizes the interface with automated provers.

A remaining issue is that deductive verification tools often do not provide enough feedback to understand why a proof fails. A recent work has shown how automated test generation can provide a better understanding of the origin of proof failures, by classifying them as prover weakness, wrong or incomplete specification [38]. Why3 integrates a prover-based counterexample generator [28], but this feature suffers from the limitations of the external provers exploited to find these counterexamples [30].

In this paper we present the first release of a tool, named AutoCheck, to test properties written in WhyML, the specification and programming language of Why3 deductive verification platform. AutoCheck integrates the complementary techniques of random and enumerative test data generation. It exploits Why3 extraction mechanism from WhyML to OCaml, a library of enumeration programs in WhyML, named ENUM, and a third-party random testing tool for OCaml named QCheck [4]. AutoCheck completes the latter with random testing for WhyML properties and enumerative testing for WhyML and OCaml properties. By interfacing with the enumeration programs in ENUM, which are certified by formal proofs with Why3 [22], AutoCheck offers certified enumerative testing.

The paper is organized as follows. Section 2 introduces some background about property-based testing and the tools involved in AutoCheck design, presented in Section 3. Section 4 is a tutorial on random testing for WhyML properties. Section 5 presents our implementation of enumerative testing for OCaml and WhyML.

## 2 Background

This section shortly presents OCaml, the Why3 platform and its extraction mechanism (Section 2.1), the concept of property-based testing (Section 2.2), and a discussion about the notion of (program) property (Section 2.3). It provides some background on existing tools for random and enumerative testing (Section 2.4) and on the library ENUM of enumeration programs certified with Why3 (Section 2.5).

### 2.1 OCaml and Why3

OCaml is a programming language developed and distributed by INRIA since 1996 [9]. The powerful type system, as well as the automated memory management (incremental garbage collector) make OCaml a very safe language. It comes with a compiler producing native code for many architectures, and a compiler producing bytecode, for increased portability.

Why3 is a platform for deductive program verification. Programs for Why3 are written in WhyML, a verification-oriented dialect of ML with some functional features, such as polymorphic algebraic types, but also imperative features, such as loops or records with mutable fields. The functional behavior of WhyML programs can be specified with formal annotations: preconditions, postconditions, invariants and loop variants, assertions, etc., in a first-order logic with polymorphic types. Why3 standard library defines theories or data structures for common types such as integers, lists or arrays. Why3 reduces programs and specifications to logical verification conditions whose satisfiability entails that the programs meet their specifications. Then, automated provers (e.g., SMT solvers) or proof assistants (e.g., Coq) can be used to prove these logical statements. Why3 also provides a driver-based automated extraction mechanism. The driver maps WhyML symbols to the syntax of the target language. A user can write WhyML programs directly and get correct-by-construction OCaml [37] programs using the OCaml driver provided by Why3. Why3 also accepts custom extraction drivers. Thus, the extraction can be adapted to different languages, as is the case for the C [40] or Rust [25] languages.

### 2.2 Property-based Testing

Property-based testing consists in identifying and testing a set of properties that some functions should satisfy. Beyond the basic case of function contracts, that are properties about one call of one function, more generally properties of high interest are *relational properties* which can concern several functions and/or several calls of the same function [14]. Some tests of relational properties are presented in Section 4.

We shall see in Section 2.3 that the concept of property is closely related to the concept of logical specification. The task of identifying the properties to be tested can be difficult, especially for programmers who have no background in formal program verification. Property-based testing can allow these programmers to become familiar with formal methods, while increasing their level of code understanding, since reasoning about code properties forces us to reason at higher levels of abstraction than we do with traditional unit tests. For more advanced users in formal verification, property-based testing can be an excellent complement during the formal proof process, allowing the discovery of incomplete or erroneous understanding of logical conjectures or specifications. Before investing time in interactively proving a non-trivial lemma or theorem, it is wise to test it. Indeed, programming does not flow in a single direction from specifications to implementation, but must evolve by cross-checking and updating both specifications and implementation.

### 2.3 What is a property, syntactically?

What a property is, syntactically, depends on the language: In a programming language like OCaml, a property has to be specified as a Boolean function, since the language supports no syntactic entity for logical formulas. In a logical framework like/such as Coq or Why3, a property can also be a conjecture, i.e., a not-yet-proved lemma or theorem. In a contract-based verification platform, such as Why3, a property can be a Boolean function or a conjecture, but it can also be a verification condition generated by the tool from function contracts.

Whatever is a property in a given language, it has to be turned into a Boolean function to be tested. This implementation can be arbitrarily hard or impossible, since it is nothing less than providing a decision procedure for the problem expressed by the property.

### 2.4 Random and Enumerative Testing Tools

The two most popular approaches of property-based testing are random and enumerative testing. Below, we explain the difference between these two approaches, which is how test cases are generated, we list some random and enumerative testing tools, and then note the complementarity between the two approaches.

**Random Testing** consists of the automatic generation of random test cases. The ancestor of property-based random testing is the QuickCheck tool [17], originally written for the Haskell language. It has been adapted to more than thirty languages (see, e.g., [8,5,3,2,1]). Among these tools, QCheck [4] implements random testing for the OCaml language. Used to test OCaml functions [34], QCheck provides many useful combiners to generate different types of data, and also allows users to write their own generators, especially for recursive types, algebraic types or tuples. QCheck also provides the shrinking function, which reduces the size of the counterexample provided in case of test failure. For example, if the tested property is the existence of a given number in a list, it should return a list of length 1 containing only this number. To our knowledge, QCheck is the only property-based testing tool for OCaml that is regularly maintained. In addition, QCheck is used in several OCaml teaching courses [36,35,33].

**Enumerative Testing** generates and tests all possible inputs up to a size limit. It is also known as bounded exhaustive testing (BET, for short). Enumeration is used in a variety of property-based testing tools. It has first been used to check properties of functional languages, as exemplified by SmallCheck in Haskell [41]. Then, it has been adapted to several proof assistants, e.g., to Isabelle in Quickcheck [16] and to Coq, in an extension of QuickChick named CUT (Coq Unit Testing) [20]. In a former work we have initiated a BET tool for WhyML, but this prototype was limited to integer arrays [22].

**Complementarity** of random and enumerative testing becomes clear after listing some advantages and drawbacks of both approaches. Indeed, while an enumerative test is useful for small data sizes [21], the number of test cases often increases exponentially with the size limit, meaning that the test becomes too slow, perhaps impossible, beyond a relatively small input size. Random testing can be an alternative to check data with large size. Unfortunately, random testing does not support existential properties: "the random testing would rarely give useful information about an existential property: often there is a unique witness and it is most unlikely to be selected at random" [39]. Enumerative testing, in contrast, is more likely to find this witness or prove its absence below some size.

### 2.5 Certified Library of Enumeration Programs

ENUM is a library of certified enumeration programs for BET. It is freely distributed at https://github.com/alaingiorgetti/enum. Its first releases were composed of C programs specified in ACSL language and verified with Frama-C plugin WP for deductive verification [26]. Since release 1.2 used in this work, ENUM also contains enumeration programs specified and implemented in WhyML. It is an almost complete adaptation in WhyML of the C/ACSL enumeration programs, completed by new generators. Its programs implement algorithms that enumerate combinatorial structures [10] and have various applications in combinatorics. More details about ENUM 1.2 and its integration in AutoCheck are provided in Section 5.2.

## 3 AutoCheck

This section presents the principles, design choices and architecture of our testing tool AutoCheck. It is freely distributed at https://github.com/alaingiorgetti/autocheck. The work presented in this paper corresponds to its release 0.1.0. It is a prototype with the most basic functionalities, intended to be completed collaboratively in the coming years.

AutoCheck has been designed with simplicity (for users, but also for tool authors) and usability as highest priority. Firstly, a Dockerfile is provided, making installation as simple as running a shell script or a Makefile entry, building a virtual machine (a *container* in docker terminology) in which the tool can be executed safely for the host system. Secondly, many examples of tests in OCaml (resp. WhyML) syntax are provided, in a single file named TestExamples.ml (resp. TestExamples.mlw). They are ordered by increasing complexity and they cover all the functionalities of the tool. Some of these examples are documented in Sections 4 and 5. Moreover, syntaxes for OCaml and WhyML, as well as for random and enumerative tests, have been chosen to be as similar as possible.

The tool workflow is depicted in Fig. 1. AutoCheck itself is represented by the largest rectangle with rounded corners. Automatically generated files are represented by dashed rectangles. AutoCheck's input is represented by a rectangle with square corners. It is either a WhyML or an OCaml file containing tests. It is named Tests.mlw or Tests.ml in the figure. Each test in OCaml exploits one or more random or enumerative data generators, respectively defined in the third-party random testing tool QCheck (whose main file is *QCheck.ml*) and in our enumerative testing tool for OCaml (whose main file is SCheck.ml). As detailed in Section 5.2 the latter encapsulates several enumeration programs from release 1.2 of ENUM library. This OCaml code in the file Enum.ml is automatically extracted by Why3 from WhyML enumeration programs whose properties are proved with Why3. The files QCheck.mlw and SCheck.mlw respectively specify random and enumerative testing in WhyML, so that tests can be written in WhyML (in Tests.mlw in the figure), and their extraction with Why3 generates tests in OCaml, exploiting the random and enumerative testing tools for OCaml defined in *QCheck.ml* and SCheck.ml.

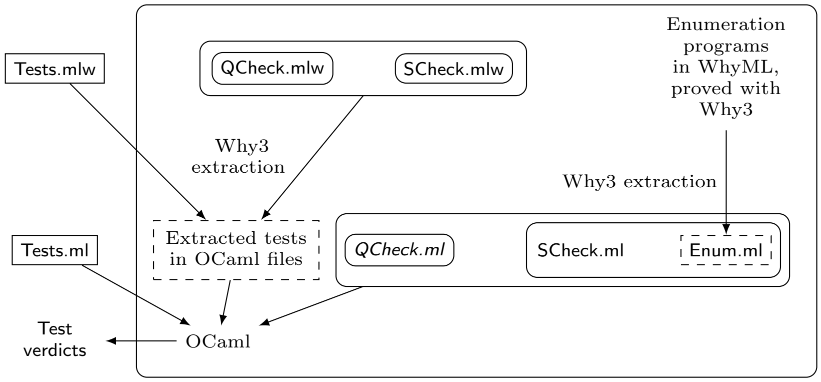

**Fig. 1.** AutoCheck workflow.

For technical reasons, AutoCheck 0.1.0 is developed for release 1.2.0 of Why3, exploiting the SMT solvers Alt-Ergo 2.2.0, CVC4 1.6 and Z3 4.7.1.

## 4 Random Testing for Why3

This section is a tutorial on random testing for WhyML properties with AutoCheck. Its first release provides random generators for WhyML built-in types (`unit`, `bool`, `option` and Cartesian products) and some types from Why3 standard library (`(list 'a)` and `(array 'a)`, for any type variable `'a`). The tutorial presents examples of random tests for each type, in increasing order of complexity. The tested properties are either lemmas in Why3 standard library or relational properties between functions defined in that library.

### 4.1 Basic types and Cartesian products

The most elementary type in WhyML (and OCaml) is `unit`. Its unique inhabitant is `()`. Let us start with the false property "`()` is not an inhabitant of `unit`", to illustrate counterexample generation. The property is implemented by the Boolean function `is_unit` reproduced on Lines 1-2 in Listing 1.1. It intentionally contains an error. The test (on Lines 4-5) is built by the `Test.make` function, applied to a random generator `QCheck.unit` of data with type `unit`, and to the Boolean function `is_unit`. The function `QCheck_runner.run_tests` implements test execution.

```
1  let function is_unit (x: unit) : bool =
2    match x with () → False | _ → True end
3
4  let is_unit_test = QCheck_runner.run_tests (
5    Test.make QCheck.unit is_unit)
```

**Listing 1.1.** Test of a false property about `unit`.

Assume that the code in Listing 1.1 is in the module `RandomTests` of the file TestExamples.mlw. Then, the command

```
bash ./why3_check.sh TestExamples RandomTests
```

execute all the tests defined in that module. Here, it generates the following result:

```
--- Failure --------------------------------------------

Test is_unit failed (0 shrink steps):

()
--------------------------------------------------------
```

The test fails, as expected, and returns as counterexample the inhabitant `()` of type `unit`.

Let us now consider the type `bool` for Booleans. The Boolean functions `andb`, `orb`, `notb`, `xorb` and `implb`, respectively for conjunction, disjunction, negation, exclusive disjunction and implication on Booleans are defined in Why3 standard library[1]. Let us complete this list with a Boolean function for equivalence, and then test this new function. All of these functions can be used to implement properties for further testing.

```
let function equivb (x y : bool) : bool
=
  match x with
  | True → y
  | False → notb y
  end
```

**Listing 1.2.** A user-defined function for Boolean equivalence.

Function `equivb` in Listing 1.2 is implemented by using the Boolean function `notb` for negation. In order to check this implementation, let us now test that this equivalence corresponds to the conjunction of two implications. This relational property about `equivb`, `andb`, `implb` and `notb` is implemented by the Boolean function `equivb_prop` on Lines 3-9 in Listing 1.3, taking a pair of Boolean values as input. In order to test this Boolean function (on Lines 11-12 in Listing 1.3) we define a random generator `bool_pair_arbitrary` of pairs of Booleans (Listing 1.3, line 1) by specialization of a generic generator `QCheck.pair` for the Cartesian product of two types, provided for WhyML by AutoCheck, by extraction to a similar generator provided for OCaml by QCheck.

```
 1  let bool_pair_arbitrary = QCheck.pair QCheck.bool QCheck.bool
 2
 3  let function equivb_prop (x : (bool,bool)) : bool
 4  =
 5    let (a,b) = x in
 6    match andb (implb a b) (implb b a) with
 7    | True → equivb a b
 8    | False → notb (equivb a b)
 9    end
10
11  let equivb_test =
12    QCheck_runner.run_tests (Test.make bool_pair_arbitrary equivb_prop)
```

**Listing 1.3.** Test of `equivb_prop` function.

The execution output

```
success (ran 1 tests)
```

indicates a successful test.

For instance, as detailed in Listing 1.4, we can use the new function `equivb` to check the commutativity property of the `orb` function for disjunction, whose definition is recalled in Listing 1.5.

[1] http://why3.lri.fr/stdlib/bool.html.

```
let orb_commut (x: (bool,bool)) : bool
=
  let (a,b) = x in equivb (orb a b) (orb b a)

let orb_commut_test = QCheck_runner.run_tests (
  Test.make bool_pair_arbitrary orb_commut)
```

**Listing 1.4.** Test of the property “`orb` is commutative”.

```
let function orb (x y : bool) : bool =
  match x with
  | False → y
  | True → True
  end
```

**Listing 1.5.** Function `orb`.

Now let us consider the WhyML type `int` for integers and its theory in Why3 standard library. Since WhyML integers represent unlimited mathematical integers, they are usually extracted to the arbitrary-precision integers of `Zarith` OCaml library [7]. However QCheck for OCaml does not provide any support for arbitrary-precision integers, and it is tricky to extend it to `Zarith`, because a choice must be made between the types `Zarith.t` of arbitrary-precision integers and `int` of limited-precision integers for each use of integers in this third-party code. Therefore, we have chosen to extract WhyML integers to OCaml regular integers. Of course, this semantical change may lead to contradictions between test and proof results. Properties with integers can only be safely tested under the hypothesis that there will be no arithmetic overflow.

AutoCheck promotes to WhyML the three random generators of integers defined in QCheck: a random generator `int` of OCaml integers, a generator `int_range` of random values in some interval $[a..b]$, and a generator `int_bound` of random values in some interval $[0..n]$. The following example shows how using a generator of limited integers increases the chances of finding a counterexample. Consider

```
lemma Abs_pos: ∀ x:int. abs x ≥ 0
```

about the `abs` function from Why3 standard library. The lemma claims that the absolute value of a number is non-negative. Let us test a mutation of this property, where the large order $\geq$ is replaced by the strict order $>$. This false property is implemented in Listing 1.6 (lines 1), and tested with two different random generators (lines 3-4 and lines 5-6).

The first test (lines 3-4) uses the random integer generator `QCheck.int_range` with a large interval, and thus passes almost always without finding a counterexample. The second test (lines 6-7) uses the same generator with a smaller interval, and thus almost always fails. For example, when running several times, the test failed 6 times out of 10 for the interval $[-100..100]$, and only once out of 10 for the interval $[-1000..1000]$.

```
let wrong_abs_pos (n: int) : bool = abs n > 0

let wrong_abs_pos_test1 = QCheck_runner.run_tests (
  Test.make QCheck.(int_range (-100000) 100000) wrong_abs_pos)

let wrong_abs_pos_test2 = QCheck_runner.run_tests (
  Test.make QCheck.(int_range (-10) 10) wrong_abs_pos)
```

**Listing 1.6.** Test of a wrong property, mutation of lemma `Abs_pos`.

Now, let us check

```
lemma Abs_le: ∀ x y:int. abs x ≤ y ↔ -y ≤ x ≤ y
```

from Why3 standard library. We turn it into the Boolean function `abs_le` (Listing 1.7, lines 1-4) which uses the previously defined Boolean equivalence `equivb`. A generator of pairs of bounded integers is defined on Lines 6-9. This makes the test on Lines 11-12 more readable.

```
 1  let abs_le (n: (int, int)) : bool
 2  =
 3    let (x,y) = n in
 4    equivb (abs x ≤ y) (-y ≤ x ≤ y)
 5
 6  let pair_int_arbitrary =
 7    QCheck.(pair
 8      QCheck.(int_range (-100) 100)
 9      QCheck.(int_range (-100) 100))
10
11  let abs_le_test = QCheck_runner.run_tests (
12    Test.make pair_int_arbitrary abs_le)
```

**Listing 1.7.** Test of `abs_le` function.

### 4.2 Option type

The option type in WhyML is defined in Why3 standard library by a module reproduced in Listing 1.8.

```
module Option

  type option 'a = None | Some 'a

  let predicate is_none (o: option 'a)
    ensures { result ↔ o = None }
  =
    match o with None → true | Some _ → false end

end
```

**Listing 1.8.** Definition of `(option 'a)` in Why3.

Here, `'a` is a type variable, which can be replaced by any type expression. Thus, we consider here the first example of random testing with a polymorphic type. For this type, AutoCheck promotes to WhyML the random generator `(option _)` defined in QCheck. Inspection of its code reveals that it chooses the constructor `None` in 15% of the cases. When it chooses the constructor `Some`, it uses the generator provided as parameter to derive data of type `'a`. For instance, the listing 1.9 shows how to randomly test the property `is_none`.

```
let is_none_test = QCheck_runner.run_tests (
  Test.make
    QCheck.(option QCheck.int)
    is_none
)
```

**Listing 1.9.** Example of test for option type.

### 4.3 Polymorphic Lists

The basic theory of polymorphic lists in Why3 standard library contains the definition

```
let predicate is_nil (l: list 'a)
  ensures { result ↔ l = Nil }
=
  match l with Nil → true | Cons _ _ → false end
```

to characterize the empty list `Nil`. Such a construction starting with `let predicate` is a specificity of WhyML. It simultaneously defines a logical predicate, for specifications, and a Boolean function, for computations. Consequently, the property "`Nil` is a list" can be directly tested with `is_nil`, as shown in Listing 1.10, knowing that during extraction, the logical clause `ensures` will be omitted. Notice that WhyML lists are polymorphic but the actual type of the generated lists (Booleans, here) has to be provided to the test generator.

```
let is_nil_test = QCheck_runner.run_tests (
  Test.make QCheck.(list QCheck.bool) is_nil)
```

**Listing 1.10.** Test of `is_nil` function.

The test fails after reducing the counterexample to a list of length 1:

```
--- Failure -----------------------------------------------

Test is_nil failed (64 shrink steps):

[true]
-----------------------------------------------------------
```

Let us now see how to test a recursive Boolean function such as `for_all` (reproduced in Listing 1.11) from Why3 standard library. This function returns `true` if and only if a given Boolean function `p` returns `true` for all items in a given list `l`. So, it provides a Boolean implementation for a family of universal properties over list items.

```
let rec function for_all (p: 'a → bool) (l: list 'a) : bool =
  match l with
  | Nil → true
  | Cons x r → p x && for_all p r
  end
```

**Listing 1.11.** Boolean function `for_all` from Why3 standard library.

It can be executed, and thus tested, only by applying it to an executable predicate or a side-effect free Boolean function. As an example, let us consider lists of integers and the parity property, specified by the side-effect free Boolean function `is_even` defined on Line 1 of Listing 1.12.

```
1 let is_even (n: int) : bool = mod n 2 = 0
2 let for_all_prop (l: list int) : bool = for_all is_even l
3 let for_all_test = QCheck_runner.run_tests (
4                      Test.make QCheck.(list QCheck.int) for_all_prop)
```

**Listing 1.12.** Parity of all items in a list of integers.

After execution, the test fails by returning a list of length 1 containing an odd integer.

### 4.4 Polymorphic arrays

The theory of polymorphic arrays in Why3 standard library specifies by

```
val function make (n: int) (v: 'a) : array 'a
  requires { [@expl:array creation size] n ≥ 0 }
  ensures { ∀ i:int. 0 ≤ i < n → result[i] = v }
  ensures { result.length = n }
```

a function `make` creating an array of length `n` whose elements are all initialized with value `v`. Its second postcondition can be tested with random lengths in [0..1000] as follows:

```
let length_make (n: int) : bool
=
  length (Array.make n 0) = n

let length_make_test = SCheck_runner.run_tests (
  Test.make SCheck.(int_bound 10000) length_make)
```

This is an example of relational property about arrays whose test does not require any array generator.

AutoCheck specifies for WhyML the array generators

```
val function array_of_size
  (n: Gen.int) (a: arbitrary 'a) : arbitrary {array 'a}
val function array (a: arbitrary 'a) : arbitrary {array 'a}
```

extracted to the OCaml array generators

```
array_of_size : (RS.t → int) → 'a arbitrary → 'a array arbitrary
array : 'a arbitrary → 'a array arbitrary
```

The first one accepts as first parameter any generator of integers for the length of the generated arrays, whereas the second uses a random generator of natural numbers to do it.

## 5 Enumerative Testing

The first release of AutoCheck presented in this paper offers enumerative testing for the OCaml types `unit`, `bool`, `int`, `('a option)` and `(int array)`, and for the corresponding WhyML types `unit`, `bool`, `int`, `(option 'a)` and `(array int)`, where `'a` is a type variable. Subsequent releases will moreover cover Cartesian products, polymorphic lists and arrays and user-defined types, which require a more substantial implementation effort.

Enumerating integer arrays is realistic and useful when their size and range of values are not too large. It is typically the case when arrays represent combinatorial objects such as permutations. An exhaustive testing of some array property, up to a given upper bound for array size, can also be considered as a partial proof (by enumeration) of that property. In a former work we have specified, implemented and certified with Why3 several effective programs enumerating arrays satisfying given invariants, such as being sorted or duplicate-free [22]. Section 5.2 details the integration of these generators in AutoCheck. Before that, Section 5.1 presents a basic example of enumerative test for WhyML properties.

### 5.1 Elementary Example for WhyML

AutoCheck provides generators `(SCheck.int_range a b)` and `(SCheck.int_bound n)` to enumerate integers in an interval $[a..b]$ or $[0..n]$. They are used in Listing 1.13 to test the wrong lemma

```
lemma Abs_gt0: ∀ x:int. abs x > 0
```

by enumeration. The first (resp. second) test finds the counterexample `0` in around 3 seconds (resp. less than 1 second).

```
let wrong_abs_pos_test1 = SCheck_runner.run_tests (
  Test.make SCheck.(int_range (-10000000) 10000000) wrong_abs_pos)

let wrong_abs_pos_test2 = SCheck_runner.run_tests (
  Test.make SCheck.(int_bound 10000) wrong_abs_pos)
```

**Listing 1.13.** Enumerative test of `Abs_gt0`.

This example makes it clear that the syntaxes of random and enumerative tests have been made so similar that it is elementary to turn a random test into an enumerative one, when a generator is available for it.

### 5.2 Integration of Certified Enumeration Programs

Some enumerative testing tools implement techniques such as constraint solving or local choice with backtracking, either to enumerate data or to derive s from data definitions (see [19, Section 7] for references). However, these techniques may fail or provide too slow enumerations. For efficiency and generality, we consider enumerative tests with *custom generators*, which are different enumeration programs handwritten for each family of data of interest.

Confidence in enumerative testing is increased if its enumeration programs are certified, ideally with formal proofs of their properties. Genestier et al. [26] developed a first version of a library of enumeration programs in C language, named ENUM, whose properties were formally specified with ACSL clauses and proved with Frama-C plugin WP for deductive verification [18]. A large fragment of this library has been adapted in WhyML and certified with Why3 [22].

This section summarizes the principles of this library of formally verified enumeration programs (sometimes hereafter called *generators*), and its integration in AutoCheck.

**Illustrative example.** Our illustrative example is the function `inverse_in_place` from the gallery of verified WhyML programs[2]. Its specified header is reproduced in Listing 1.14. It computes the inverse of its input array, assumed to be a permutation, in place, i.e. in the array itself. It is a specification and implementation in WhyML, by M. Clochard, J.-C. Filliâtre and A. Paskevich, of an adaptation to an array on $[0..n-1]$ of Algorithm I described by D. Knuth for an array on $[1..n]$ in Section 1.3.3, page 176 of *The Art of Computer Programming, volume 1* [29]. We do not intend here to explain the code – it is well done in the provided references – but to test by enumeration its following two properties, corresponding to the two postconditions in Listing 1.14:

($P_1$) The function `inverse_in_place` preserves permutations.
($P_2$) The function `inverse_in_place` computes in place the inverse permutation of its input.

```
let inverse_in_place (a: array int)
  requires { is_permutation a }
  ensures { is_permutation a }
  ensures { ∀ i. 0 ≤ i < length a → (old a)[a[i]] = i }
```

**Listing 1.14.** Inversion of a permutation in place, function contract.

[2] http://toccata.lri.fr/gallery/inverse_in_place.en.html

Let us first observe that the deductive verification of these properties is highly non-trivial and fragile. First, the proposed loop invariant is made up of seven universal formulas and occupies ten lines of code. Second, Why3's most advanced automatic strategy, named Auto level 2, does not overcome this proof. Third, the interactive proof distributed with this example is sensitive to changes in SMT solvers' releases. It was complete with releases 2.0.0, 1.4 and 4.4.1 of Alt-Ergo, CVC4 and Z3, but is no longer complete with our more recent releases 2.2.0, 1.6 and 4.7.1 of these tools, for the same release 1.2.0 of Why3!

**Enumerative test session.** Let us now detail how to test ($P_1$) by enumeration with AutoCheck, and how it works internally. A test of ($P_1$) with all permutations of size 6 is

```
let function inverse_in_place_permut_test
= SCheck_runner.run_tests (
    Test.make SCheck.(permut_of_size 6) inverse_in_place_permut)
```

with the Boolean implementation

```
let function inverse_in_place_permut (a: array int) : bool
= let newa = copy a in inverse_in_place newa; b_permut newa
```

of ($P_1$). An important limitation of Why3 at work here is that the second parameter of `Test.make`, as a function, should be without side effect. So, a simpler version of `inverse_in_place_permut`, such as

```
let inverse_in_place_permut (a: array int) : bool
= inverse_in_place a; b_permut a
```

would not be accepted, since it modifies the input array `a`.

Internally, the functions `Scheck.permut_of_size`, `SCheck_runner.run_tests` and `Test.make` are automatically extracted into OCaml functions with the same names. The OCaml function `Test.make` builds a test case by assembling a *serial* and a test oracle, implemented as a Boolean function. Each enumerative test case is executed by the OCaml function `SCheck_runner.run_tests`, which enumerates all data and checks the same property for each data, thanks to the test oracle included in the test case. Moreover, the execution counts the number of passing data before failure. So, the output is either a counterexample or the number of passed tests. For the present example the output is:

```
Test inverse_in_place_permut succeeds (ran 720 tests)
```

Property ($P_2$) is checked similarly. The remainder of this section present serials, such as `Scheck.permut_of_size`, generators and their certification.

**Serials.** The OCaml function `Scheck.permut_of_size` constructs a *serial*. It is an OCaml record grouping a printer of integer arrays, borrowed from the third-party tool QCheck, and a generator of permutations from ENUM library. The latter is automatically extracted with Why3 from a generator of permutations in WhyML.

**Generic interface of all WhyML generators.** Since enumeration is a particular form of iteration, the generators in ENUM are adaptations of the modular iterators defined by Filliâtre and Pereira [23,24]. They modify a state, called a *cursor*, whose type is

```
type cursor = {
  current: array int;
  mutable new: bool;
}
```

in WhyML. The field `current` only stores the last data generated so far. The Boolean flag `new` is set to `false` if and only if the data stored in the `current` field has already been exploited, for instance to test a property.

Each generator is composed of two *enumeration functions*, declared on Lines 1 and 6 in Listing 1.15: a constructor `create_cursor` initiates the cursor with the first element of the iteration, and a function `next` replaces the data in the cursor with the next one, if it exists. Otherwise, it sets the field `c.new` to false.

```
 1 val create_cursor (n: int) : cursor
 2   requires { n ≥ 0 }
 3   ensures { c.new → sound result }
 4   ensures { c.new → min result.current }
 5
 6 val next (c: cursor) : unit
 7   requires { sound c }
 8   ensures { c.new → sound c }
 9   ensures { c.new → lt (old c.current) c.current }
10   ensures { c.new → inc (old c.current) c.current }
11   ensures { not c.new → max (old c.current) }
```

**Listing 1.15.** Enumeration functions and their contracts.

**Generator properties.** Each generator is expected to satisfy the following behavioral properties. *Soundness* is the property that each generated data satisfies the characteristics (or *data invariant*) of its family, such as being a duplicate-free or a sorted array. *Completeness* is the property that the program produces all existing data with a given size, without omitting any of them. Generally, proving completeness is more challenging than proving soundness. Therefore, we limit ourselves to algorithms enumerating data in a predefined strict total order, hereafter denoted by $\prec$, and we adopt two strategies. The first strategy is to specify completeness as the conjunction of the following three properties: the property *min* that the first generated data is the smallest one, the property *max* that the last generated data is the largest one, and the property *inc* (for "incrementality") that each data $a_2$ generated from data $a_1$ is the smallest data strictly greater than $a_1$. In other words, no sound data $a_3$ is such that $a_1 \prec a_3 \prec a_2$. When proving completeness seems too difficult, the second strategy is to address the less challenging property – named *progress* – that each generated data is strictly greater than the former generated data. Since we assume that there are finitely many data with each size, progress entails termination of enumeration.

Listing 1.15 shows a formalization of these properties in WhyML, as contracts (pre- and postconditions) for the enumeration functions. The precondition on Line 2 specifies that the size $n$ of data should be a natural number. The function `create_cursor` (resp. `next`) should set the cursor field `c.new` to false if and only if there is no data for a given size $n$ (resp. the input cursor contains the last data). Therefore, most of the properties are formalized by postconditions guarded by the condition that the Boolean flag `c.new` is true. We assume that a predicate

```
predicate sound (c: cursor)
```

encapsulates the data invariant. Then, a generator is sound if the first generated data satisfies this predicate (postcondition on Line 3) and if the output of the `next` function satisfies this predicate (postcondition on Line 8) whenever its input does (precondition on Line 7). The progress property is formalized on Line 9, with a predicate `lt` formalizing the strict total order $\prec$. (The expressions `(old e)` and `e` in a function postcondition respectively denote the values of the expression `e` before and after the function call.) The properties *min*, *inc* and *max* (entailing completeness) are respectively formalized on Lines 4, 10 and 11, with predicates `min`, `inc` and `max` respectively formalizing minimality, incrementality and maximality of the restriction of the order $\prec$ to data satisfying the data invariant `sound`.

These contracts are proved by a combination of the following two deductive verification techniques: *Auto-active verification* [32] consists in providing additional specifications, such as variants (for termination), invariants, assertions and lemmas (for partial correctness), before running an automated prover. *Interactive verification* consists in reducing the proof goal step by step, by applying rules – named *tactics* in Coq and *transformations* in Why3.

**Enumeration by filtering.** Assume you already have implemented, specified and certified an enumeration program for some family of data. Then an enumeration program for those data that satisfy an additional constraint can easily be implemented by running your program and selecting among its outputs those satisfying that constraint. Of course, the more data are rejected, the less effective is the resulting program. However, we have shown in a fomer work [22, Section 3.2] that this *filtering* technique provides a specification, an implementation and a certification of the resulting enumeration program almost for free.

**Contents of ENUM 1.2.** Table 1 presents the generators in ENUM 1.2 and some metrics about them. The first column assigns a name to each generator. The number of lines of code (resp. WhyML annotations) is recorded in the second (resp. third) column. The fourth (resp. fifth) column gives the number of transformations (resp. lemmas) needed for the proof of soundness, progress and completeness properties. All of them have been proved automatically with Why3 1.2.0 and the SMT solvers Alt-Ergo 2.2.0, CVC4 1.6 and Z3 4.7.1, except the completeness property for the generator of permutations, which required an interactive proof of two lemmas with Coq 8.9.0.

| Array family | Code | Specification | Transformations | Lemmas | Time (s) |
|---|---|---|---|---|---|
| RGF | 26 | 22 | 1 | 0 | 1.98 |
| SORTED | 22 | 26 | 4 | 0 | 3.21 |
| PERM | 42 | 86 | 5 | 16 | 16.35 |
| BARRAY | 22 | 23 | 3 | 0 | 3.14 |
| FACT | 22 | 20 | 1 | 0 | 1.53 |
| ENDO | 22 | 22 | 0 | 0 | 1.13 |
| SORTED $\subset$ BARRAY | 24 | 15 | 0 | 0 | 1.05 |
| INJ $\subset$ BARRAY | 24 | 16 | 0 | 0 | 0.92 |
| SURJ $\subset$ BARRAY | 34 | 25 | 0 | 0 | 1.1 |
| COMB $\subset$ BARRAY | 17 | 10 | 0 | 0 | 0.84 |

**Table 1.** Generators in ENUM 1.2.

The first block of lines in Table 1 concerns effective enumeration programs. The first four are adaptations of C++ programs proposed in [10]. The program RGF (for "Restricted Growth Function") enumerates the arrays $a$ of length $n$ such that $a[0] = 0$ and $a[i] \leq a[i-1]+1$ for $1 \leq i \leq n-1$. SORTED generates all arrays from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ sorted in increasing order. PERM enumerates the permutations on $\{0, ..., n-1\}$. BARRAY (for "bounded array") (resp. ENDO) (for "endo-array") enumerates the arrays of length $n$ whose values are in $\{0, ..., k-1\}$ (resp. $\{0, ..., n-1\}$). FACT enumerates the $n!$ *factorial* arrays [27] $f$ of length $n$ such that $0 \leq f[i] \leq i$ for $1 \leq i \leq n-1$.

The second block concerns enumeration programs obtained by filtering. We denote by Z $\subset$ X an enumeration program of data Z by filtering among more general data X. For instance, SORTED $\subset$ BARRAY enumerates increasing arrays filtered among bounded arrays. By filtering from BARRAY we get generators for the following data families: arrays sorted in increasing order, injections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$, for $n \leq k$ (INJ $\subset$ BARRAY), surjections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$, for $n \geq k$ (SURJ $\subset$ BARRAY), and combinations of $n$ elements selected from $k$, (COMB $\subset$ BARRAY), which are encoded by arrays $c$ of length $n$ such that $0 \leq c[0] < \ldots < c[n-1] \leq k-1$.

## References

1. Property based testing framework for JavaScript., https://github.com/dubzzz/fast-check
2. Property-based testing library for Java 8+., https://github.com/JetBrains/jetCheck
3. PROPerty-based testing tool for ERlang., https://github.com/proper-testing/proper
4. QuickCheck inspired property-based testing for OCaml., https://github.com/c-cube/qcheck

5. Randomized property-based testing plugin for Coq., https://github.com/QuickChick/QuickChick
6. The Boogie intermediate verification language., https://github.com/boogie-org/boogie/
7. The Zarith library., https://github.com/ocaml/Zarith
8. Theft: property-based testing for C., https://github.com/silentbicycle/theft
9. What is OCaml?, https://ocaml.org/learn/description.html
10. Arndt, J.: Matters Computational - Ideas, Algorithms, Source Code [The fxtbook] (2010), https://www.jjj.de/fxt/fxtpage.html
11. Barnett, M., Leino, K., Schulte, W.: 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). LNCS, vol. 3362, pp. 49–69. Springer-Verlag, Marseille, France (March 2004)
12. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, published electronically at http://frama-c.com/acsl.html
13. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS 4334, Springer, Heidelberg (2007)
14. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V., Petiot, G.: Static and dynamic verification of relational properties on self-composed c code. In: Dubois, C., Wolff, B. (eds.) Tests and Proofs. pp. 44–62. Springer International Publishing, Cham (2018)
15. Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 Platform (2018), http://why3.lri.fr/manual.pdf
16. Bulwahn, L.: The new quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer, Heidelberg (2012), https://doi.org/10.1007/978-3-642-35308-6_10
17. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. SIGPLAN Not., vol. 35, pp. 268–279. ACM, New York (2000). https://doi.org/10.1145/351240.351266
18. Correnson, L.: Qed. Computing what remains to be proved. In: NFM 2014. LNCS, vol. 8430, pp. 215–229. Springer, Heidelberg (2014)
19. Dubois, C., Giorgetti, A.: Tests and proofs for custom data generators. Formal Aspects Comput. **30**, 659–684 (Jul 2018), https://doi.org/10.1007/s00165-018-0459-1
20. Dubois, C., Giorgetti, A., Genestier, R.: Tests and proofs for enumerative combinatorics. In: Aichernig, K.B., Furia, A.C. (eds.) TAP 2016. LNCS, vol. 6792, pp. 57–75. Springer, Cham (2016), https://doi.org/10.1007/978-3-319-41135-4_4
21. Duregård, J., Jansson, P., Wang, M.: Feat: Functional enumeration of algebraic types. ACM SIGPLAN Notices (12 2012)
22. Erard, C., Giorgetti, A.: Bounded Exhaustive Testing with Certified and Optimized Data Enumeration Programs, pp. 159–175 (10 2019)
23. Filliâtre, J.-C., Pereira, M.: Itérer avec confiance. In: Journées Francophones des Langages Applicatifs (JFLA 2016) (2016), https://hal.inria.fr/hal-01240891
24. Filliâtre, J.-C., Pereira, M.: A modular way to reason about iteration. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 322–336. Springer, Cham (2016), https://doi.org/10.1007/978-3-319-40648-0_24
25. Fitinghoff, N.: Extraction of Rust code from the Why3 verification platform. Master's thesis, Luleå University of Technology (2019), http://www.diva-portal.org/smash/get/diva2:1303268/FULLTEXT02#page20

26. Genestier, R., Giorgetti, A., Petiot, G.: Sequential generation of structured arrays and its deductive verification. In: Blanchette, J.C., Kosmatov, N. (eds.) TAP 2015. LNCS, vol. 9154, pp. 109–128. Springer, Cham (2015), https://doi.org/10.1007/978-3-319-21215-9_7
27. Giorgetti, A., Dubois, C., Lazarini, R.: Combinatoire formelle avec Why3 et Coq. In: Journées Francophones des Langages Applicatifs (JFLA 2019). pp. 139–154 (2019), https://hal.inria.fr/hal-01985195
28. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from Proof Failures in SPARK. In: Software Engineering and Formal Methods. Software Engineering and Formal Methods, Springer, Vienna, Austria (Jul 2016), https://hal.inria.fr/hal-01314885
29. Knuth, D.E.: The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., USA (1997)
30. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014. In: 7th International Symposium on Leveraging Applications. pp. 461–478. Springer, Corfu, Greece (2016), https://hal.inria.fr/hal-01344110
31. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer Academic Publishers, Boston (1999)
32. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010), http://fm.csl.sri.com/UV10/
33. Midtgaard, J.: Functional Programming and Property-Based Testing., http://janmidtgaard.dk/quickcheck/
34. Midtgaard, J., Møller, A.: Quickchecking static analysis properties. Software Testing, Verification and Reliability (09 2017), https://cs.au.dk/~amoeller/papers/lcheck/paper.pdf
35. Miné, A., Ouadjaout, A., Journault, M., Monat, R.: QuickCheck de bibliothèques d'analyse statique en OCaml., http://www-master.ufr-info-p6.jussieu.fr/2019/QuickCheck-de-bibliotheques-d
36. Naves, G., Spiwack, A.: Programmation fonctionnelle., http://assert-false.net/callcc/Guyslain/Teaching/ProgFonc/Cours/cours-2-4-quickcheck
37. Pereira, M.J.P.: Tools and Techniques for the Verification of ModularStateful Code. Ph.D. thesis, Université Paris-Sud (2018), https://tel.archives-ouvertes.fr/tel-01980343/document
38. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: How testing helps to diagnose proof failures. Formal Aspects of Computing **30**, 629–657 (Jun 2018). https://doi.org/10.1007/s00165-018-0456-4, https://doi.org/10.1007/s00165-018-0456-4, http://nikolai.kosmatov.free.fr/publications/petiot_kbgj_faoc_2018.pdf
39. Reich, J., Naylor, M., Runciman, C.: Advances in lazy smallcheck. vol. 8241, pp. 53–70 (11 2013)
40. Rieu-Helft, R.: Un mécanisme d'extraction vers C pour Why3. In: Journées Francophones des Langages Applicatifs (JFLA'18). pp. 203–209 (2018), https://hal.inria.fr/hal-01707376v1
41. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell. pp. 37–48. ACM (2008), http://doi.org/10.1145/1411286.1411292