Towards random and enumerative testing for OCaml and WhyML properties

Clotilde Erard, Jérome Ricciardi, Alain Giorgetti

Towards Random and Enumerative Testing for OCaml and WhyML Properties

Clotilde Erard, Alain Giorgetti, and Jérome Ricciardi

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

Abstract. Deductive program verification greatly improves software quality, but proving formal specifications is difficult, and this activity can only be partially automated. It is therefore relevant to supplement deductive verification tools, such as Why3, with the ability to test the properties to be verified. We present a methodological study and a prototype for the random and enumerative testing of properties written either in the Why3 input language WhyML or in the OCaml programming language used by Why3 to run programs written in WhyML. An originality is that we propose enumerative testing based on data generators themselves written in WhyML and formally verified with Why3. Another specificity is that the development effort is reduced by exploiting Why3's extraction mechanism to OCaml and an existing random testing tool for OCaml. These design choices are applied in a propotypal implementation of a tool, called AutoCheck. The prototype 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, a tutorial illustrates the implemented features with many examples presented in increasing complexity order.

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

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 a syntactic entity of formal assertion available in some programming languages and 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 [4] and Spec# for C# [3]. Deductive verification tools – such as KeY [5] for Java/JML, the WP plugin [13] of Frama-C

for C/ACSL or Boogie [8] 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 [7], 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 specification or incomplete specification [41]. Why3 integrates a prover-based counterexample generator [26], but this feature suffers from the limitations of the external provers exploited to find these counterexamples [30].

Following the principles of property-based testing, we suggest to supplement deductive verification tools, such as Why3, with the ability to test the properties to be verified. We present design principles and illustrate them with a prototype, named AutoCheck, to test properties written in WhyML, the specification and programming language of Why3. AutoCheck aims at the integration of the complementary techniques of random and enumerative test data generation. AutoCheck is not yet complete enough for industrial applications, but it can already be used and extended by OCaml and WhyML developers. It is presented here as a proof-of-concept for the following design choices, which are as many contributions:

1. AutoCheck includes a library of enumeration programs in WhyML, named ENUM, which are certified by formal proofs with Why3.
2. AutoCheck integrates the mature third-party random testing tool QCheck [43] for OCaml.
3. AutoCheck completes QCheck with random testing for WhyML properties and enumerative testing for WhyML and OCaml properties.
4. This implementation of random and enumerative testing for WhyML exploits Why3's extraction mechanism from WhyML to OCaml and an implementation of random and enumerative testing for OCaml. This shallow approach by extraction greatly reduces the amount of code to develop.

The paper is organized as follows. Section 2 introduces some background about property-based testing and the tools involved in AutoCheck design, presented in Sect. 4. Section 3 presents our library of certified enumeration programs and the principles of their certification by formal proofs with Why3. Section 5 is a tutorial on random testing for WhyML properties. Section 6 presents our implementation of enumerative testing for OCaml and WhyML. Section 7 is dedicated to concluding remarks.

2 Background

This section shortly presents OCaml, the Why3 platform and its extraction mechanism (Sect. 2.1), the principles of property-based testing and the notion of property in this context (Sect. 2.2), some background on existing tools for random and enumerative testing (Sect. 2.3) and a discussion about various possible origins of properties for property-based testing (Sect. 2.4).

2.1 OCaml and Why3

OCaml is a programming language developed and distributed by INRIA since 1996 [38]. 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 the language 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.

WhyML offers usual non-mutable types such as `unit`, `bool`, `int`, or the polymorphic type `list 'a`, where `'a` is a type variable that can be replaced by any type. WhyML also offer mutable types such as `array 'a`. It is possible to change the value of a variable if its type is mutable. The value of a variable with a non-mutable type cannot be changed, but it is possible to declare a reference on a non-mutable type, with the keyword `ref`, and to access its value with the operator `!`. Some syntactic sugar is provided to lighten the notation – a documentation on this can be found in Why3's manual [52, section 7.4.3].

The functional behavior of WhyML programs can be specified with formal annotations, globally called *contracts*: preconditions, postconditions, invariants and loop variants, assertions, etc., in a first-order logic with polymorphic types. Why3's 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., Isabelle [10] or Coq [12]) 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 [40] 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 [49] or Rust [22] languages.

2.2 Property-based testing

> "First things first, what is property-based testing? A property of a program is an observation that we expect to hold true regardless of the

> program's inputs. It may involve only the output ("always outputs a positive number") or compare input and output ("preserves list length") or even assess external effects ("matches the output of a trusted external program")." [37].

Property-based testing (PBT, for short) 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, properties addressed here are *relational properties* which can concern several functions and/or several calls of the same function [6]. Some tests of relational properties are presented in Sect. 5. Temporal properties, written in a temporal logic such as LTL, CTL or μ-calculus, are out of the scope of the present work.

Along the line of many property-based testing tools described in Sect. 2.3, we assume that each property to be tested comes under the form of an executable function returning a Boolean value providing the test oracle (true if the test passes, false if it fails). Hereinafter this function is often called an *executable property*.

We shall see in Sect. 2.4 how properties could be derived from functional specifications. 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.

2.3 Random and enumerative testing tools

Our goal is to adapt to OCaml and Why3 the two most basic and oldest techniques of PBT, which are random and enumerative testing. The integration of more recent PBT approaches, such as fuzzing [39], may be studied later. Below we list some random and enumerative testing tools and we note the complementarity between the two approaches.

Random testing. Random testing consists of the automatic generation of random test cases. The ancestor of property-based random testing is the QuickCheck tool [11], originally written for the Haskell language. It has been adapted to more than thirty languages (see, e.g., fast-check for JavaScript [19], jetCheck for Java [28], PropEr for ERlang [42], QuickChick for Coq [45] and theft for C [51]).

There are several random testing tools for OCaml, e.g., the QuickCheck module from JaneStreet's core_kernel framework [44], Kaputt [48], Crowbar [14] or QCheck [43]. Among them, we choose to embed QCheck in our prototype. Used

to test OCaml functions [34], QCheck provides many useful combinators 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. In addition, QCheck is used in several OCaml teaching courses [33,35,36].

Enumerative testing. Enumerative testing, also known as bounded exhaustive testing (BET, for short), is used in a variety of property-based testing tools. It consists of generating and testing all possible inputs at a size limit. It has first been used to check properties of functional languages, as exemplified by SmallCheck in Haskell [50]. Then, it has been adapted to several proof assistants, e.g., to Isabelle in Quickcheck [9] and to Coq, in an extension of QuickChick named CUT (Coq Unit Testing) [16]. In a former work we have initiated a BET tool for WhyML [18].

Complementarity. The 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 [17], 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" [47]. Enumerative testing, in contrast, is more likely to find this witness or prove its absence below some size.

2.4 Where can properties come from?

Most PBT tools, starting with the pioneer QuickCheck for Haskell, generate tests from *user-provided properties*, i.e., properties assumed to be handwritten by the user. Can we further assist the user, with methods and tools that would automatically generate executable properties from formal specifications? We briefly explore this question in the context of the OCaml and WhyML languages.

The answer depends on the language: In a programming language like OCaml, a property can only be specified as a function returning a Boolean value, because the language supports no syntactic entity for logical formulas. In a logical framework such as Coq, an executable property could also be derived from a conjecture – a not-yet-proved lemma or theorem – in order to detect errors in it before attempting to prove it. In a language such as WhyML, for logical formulas, programs and formal program specifications, a property can again be a function returning a Boolean value or be derived from a conjecture, but it might also be derived from programs and their formal specifications.

Whatever the possible origin of a property in a given language, to be tested it has to be turned into an executable function returning a Boolean value. 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. It is therefore restricted to the limits of decidability.

In the deductive program verification method implemented in Why3, the program and its formal specification are transformed into *verification conditions* (VC, for short) by a *verification condition generator* (VCGEN, for short). Then external automated provers are called separately on each VC, to try to prove it. Because these automated proof attempts can fail or take much time, it can be useful to supplement them with PBT on these VC.

For the sake of simplicity we hereafter say that the properties under test are "user-provided", even if they may be produced mechanically by some *executable property generator*, as detailed here. The design of such a tool is left as future work. For now, AutoCheck helps find errors in user-provided properties, thus playing a role similar to that of an automated prover in the deductive method.

3 Library 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 effective generators from data definitions (see [15, Sect. 7] for references). However, these techniques may fail or provide too slow enumerations. For efficiency and generality, we consider enumerative tests with *custom enumeration programs* (sometimes hereafter called *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. [23] developed a first version of a library of enumeration programs in the C language, named ENUM, whose properties were formally specified with ACSL clauses and proved with the Frama-C plugin WP for deductive verification [13]. A large fragment of this library has been adapted in WhyML and certified with Why3 [18]. ENUM is freely distributed at https://github.com/alaingiorgetti/enum. Its programs implement algorithms that enumerate combinatorial structures [2] and have various applications in combinatorics.

This section details the principles and contents of the library ENUM. All enumeration programs implement the same interface and share the same specification, both described in Sect. 3.1. As an illustrative example, the implementation and certification of a generator of permutations are presented in Sect. 3.2. Section 3.3 presents a simple way to define a generator, by filtering the output of another generator. The certified enumeration programs distributed to date are described in Sect. 3.4. The integration of ENUM in AutoCheck is detailed in Sect. 6.2.

3.1 Generic interface and contract of enumeration programs

Since enumeration is a particular form of iteration, the enumeration programs in ENUM are adaptations of the modular iterators defined by Filliâtre and Pereira [20,21]. 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. For simplicity, it is here a mutable array of integers, but the approach could be extended to other datatypes. 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 and formally specified in WhyML in Listing 1.1. A constructor `create_cursor` initiates the cursor with the first element of length `n` of the iteration. 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 { result.new → sound result }
4   ensures { result.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 1.1. Enumeration functions and their contracts.

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 length, 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 *incrementality* property 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 length, progress entails termination of enumeration.

Listing 1.1 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 length n of data should be a natural number. Most of the properties are formalized by postconditions guarded by a condition on the value of the cursor field `new`. Indeed the value of this Boolean flag should be initialized to `true` if and only if the set to be enumerated is not empty, and set to `false` as soon as the set of data remaining to be enumerated becomes empty. This informal specification of the cursor field `new` could also be formalized as postconditions for functions `create_cursor` and `next`. Since proving this additional contract can be hard, we defer these specifications and proofs for future work.

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*, *incrementality* and *max* (entailing the *completeness* property) 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`.

The library ENUM provides formal definitions (in WhyML) of these predicates `min`, `inc` and `max` for any data invariant, when the order $\prec$ is the lexicographic order induced on arrays of integers by the standard order $<$ on integers. Thus, the designer of a program enumerating a new family of integer arrays in lexicographic order can re-use these definitions. She just has to implement the enumeration functions and perform their deductive verification, as detailed on an example in Sect. 3.2.

The contracts of the enumeration functions 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.

3.2 Certified enumeration of permutations

This section presents an implementation and a deductive verification of enumeration functions for permutations on the set $[0..n-1]$ of first n natural numbers. We encode such a permutation p by the integer array `a` of its images. It is the array of length `a.length` $= n$ such that `a`$[i] = p(i)$ for $0 \leq i < n$. We characterize these permutation arrays with the predicate

```
predicate is_permut (a: array int) = range a ∧ injective a
```

where `(range a)` specifies that the values of array a are in $[0...$`a.length` $- 1]$ and `(injective a)` specifies injectivity of the function represented by a, *i.e.*, uniqueness of values in a.

Initialization. The smallest permutation in lexicographical order is the one sorted in ascending order, *i.e.* the identity function. It is characterized on any subarray `a[l..u]` by the predicate

```
predicate is_id_sub (a:array int) (l u:int) =
  ∀ i:int. l ≤ i < u → a[i] = i
```

which specifies that each array value is its index. The function `create_cursor` (Listing 1.2) returns a cursor initialized with the identity table and the `new` field equal to true.

```
1  let create_cursor (n: int) : cursor
2    requires { n ≥ 0 }
3    ensures { result.new && sound result }
4    ensures { min result.current }
5    ensures { result.current.length = n }
6  = let p = make n 0 in
7    for i = 0 to n-1 do
8      invariant { 0 ≤ i ≤ n }
9      invariant { is_id_sub p 0 i }
10     p[i] ← i
11   done;
12   { current = p; new = true }
```

Listing 1.2. Initialization function for a generator of permutations.

The function first creates an array initialized to 0 (Line 6), then sets each array value to its index (Line 10). The second invariant (Line 9) asserts that at each loop iteration the array is the identity up to the current index. The postconditions ensure the soundness property (Line 3) and the minimality property (Line 4).

Successor function. The function `next` computing the next permutation in lexicographic order is presented in the Listing 1.3. Its contract specifies the soundness, progress and completeness properties by one precondition (line 2) and four postconditions (lines 3-6).

For $n = 2$, repetitive calls to the function `next`, starting from the array $\boxed{0}\boxed{1}\boxed{2}$ generated by the function `create_cursor`, generate (in place, in the cursor `c`) the arrays $\boxed{0}\boxed{2}\boxed{1}$, $\boxed{1}\boxed{0}\boxed{2}$, $\boxed{1}\boxed{2}\boxed{0}$, $\boxed{2}\boxed{0}\boxed{1}$ and $\boxed{2}\boxed{1}\boxed{0}$.

```
1  let next (c: cursor) : unit
2    requires { sound c }
3    ensures { sound c }
4    ensures { c.new → lt (old c.current) c.current }
5    ensures { c.new → inc (old c.current) c.current }
6    ensures { not c.new → max c.current }
7  =
8    let p = c.current in
9    let n = p.length in
10   label L in
11   if n ≤ 1 then
12     c.new ← false
13   else
14     let ref r = (n-2) in
15     while r ≥ 0 && p[r] > p[r+1] do
16       invariant { -1 ≤ r ≤ n-2 }
17       invariant { is_dec_sub p (r+1) n }
18       variant  { r+1 }
19       r := r-1
20     done;
21     if r < 0 then
22       c.new ← false
23     else
24       let ref j = (n-1) in
25       while p[r] > p[j] do
26         invariant { r+1 ≤ j ≤ n-1 }
27         invariant { all_lt p r j }    (* p[j+1..n-1] < p[r] *)
28         variant  { j }
29         j := j-1
30       done;
31       swap p r j;
32       reverse p (r+1) n;
33       assert { lt_at (p at L) p r };
34       c.new ← true
```

Listing 1.3. Second enumeration function for a generator of permutations.

This function calls two auxiliary functions `swap` and `reverse` not reproduced here. The function `swap` comes from Why3's standard library. The statement `(swap` a i j`)`

swaps the elements of the array a at the indices i and j. The function `reverse` is such that `(reverse a l u)` reverses the subarray $a[l..u-1]$ of the array a.

In order to lighten the code, the variables p and n respectively represent the current permutation and its size. If this size is 0 or 1, the current permutation is the last permutation (`c.new` $\leftarrow$ `false`). Otherwise, the program proceeds by revising the suffix of the array p, as detailed in the following execution example. Let p be the integer array

i	0	1	2	3	4	5
$p[i]$	4	1	2	5	3	0

storing the values of a permutation on [0..5], also noted p ($p[i] = p(i)$ for $i = 0, \ldots, 5$). The program transforms the array p in place, in order to turn it into the smallest array p' that is strictly greater than p (according to the lexicographic order $\prec$) and represents a permutation p'. The first step of the program (lines 14-20) looks for the *revision index* r such that p and p' have the largest common prefix $p[0..r-1] = p'[0..r-1]$. When p is a permutation, this index is the largest index i such that $p[i]$ is less than $p[i+1]$. In our example of permutation p, the revision index is $r = 2$. The *suffix* is the subarray $p[r..n-1]$, from the revision index to the end of the array. The second step of the program (lines 24-30) determines the new value of $p[r]$, such that the array p' is greater than p, is as small as possible and represents a permutation. This new value of $p[r]$ is the smallest value $p[j]$ greater than $p[r]$ and present in the subarray $p[r+1..n-1]$ after the revision index. In our example, it is the value $p[4] = 3$, for $j = 4$. The third step (line 31) exchanges the values of $p[r]$ and $p[j]$, thanks to the function `swap`. We obtain then the array $p_1 = \boxed{4}\boxed{1}\boxed{3}\boxed{5}\boxed{2}\boxed{0}$. The fourth step (line 32) computes the smallest possible subarray $p'[r+1..n-1]$. For p' to be a permutation, this subarray must be the subarray $p_1[r+1..n-1]$ sorted in ascending order. Since this subarray $p_1[r+1..n-1]$ is sorted in descending order, it is sufficient to invert it with the function `reverse`, which produces the array $p' = \boxed{4}\boxed{1}\boxed{3}\boxed{0}\boxed{2}\boxed{5}$. If a revision index was not found during the first step, then r is -1 and p is the last permutation, which is indicated by assigning the value `false` to the `new` field of the cursor (line 22).

The loop invariant on Line 17 states that the subarray $p[r+1..n-1]$ is decreasing. The loop invariant `(all_lt p r j)` on Line 27 states that all values in the subarray `p[j+1..n-1]` are strictly lower than `p[r]`. Consequently, the value `p[j]` after the loop is the smallest value greater than or equal to $p[r]$ in the subarray $p[r+1..n-1]$, so the swap and the reverse after the loop minimally increase the array, a key argument to prove the *completeness* property. In the assertion on Line 33 `(p at L)` denotes the permutation `p` at the beginning of the function (Line 10). The assertion states that the subarrays `(p at L)[0..r-1]` and `p[0..r-1]` are equal and that `(p at L)[r]` $<$ `p[r]`.

With these annotations, auto-active verification of the *soundness*, *progress*, *min* and *max* properties succeeds. However an interactive proof in Coq was required to prove the harder property of *incrementality*. An intermediate version of this work, without the proof of completeness, has been presented during a French conference [24].

3.3 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 former work [18, Sect. 3.2] that this *filtering* technique provides a specification, an implementation and a certification of the resulting enumeration program almost for free.

3.4 Contents of ENUM 1.3

Table 1 presents the generators in ENUM 1.3 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 to prove their soundness, progress and completeness properties. All of them have been proved automatically with Why3 1.4.0 and the SMT solvers Alt-Ergo 2.4.0, CVC4 1.6, Z3 4.7.1 and Z3 4.8.10, except the completeness property for the generator of permutations, which required an interactive proof of two lemmas with Coq 8.12.2.

Array family	Code	Specification	Transformations	Lemmas
RGF	26	22	1	0
SORTED	22	26	4	0
PERM	42	86	5	16
BARRAY	22	23	3	0
FACT	22	20	1	0
ENDO	22	22	0	0
SORTED $\subset$ BARRAY	24	15	0	0
INJ $\subset$ BARRAY	24	16	0	0
SURJ $\subset$ BARRAY	34	25	0	0
COMB $\subset$ BARRAY	17	10	0	0

Table 1. Generators in ENUM 1.3.

The first block of lines in Table 1 concerns effective enumeration programs. The first four are adaptations of C++ programs proposed in [2]. 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 [25] 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 ⊂ X an enumeration program of data Z implemented by filtering among more general data X. For instance, SORTED ⊂ 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 ⊂ BARRAY), surjections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ for $n \geq k$ (SURJ ⊂ BARRAY), and combinations of n elements selected from k, which are encoded by arrays c of length n such that $0 \leq c[0] < \ldots < c[n-1] \leq k-1$ (COMB ⊂ BARRAY).

4 AutoCheck

This section presents the principles, design choices and architecture of our prototype AutoCheck for random and enumerative test data generation and test execution. It is freely distributed at https://github.com/alaingiorgetti/autocheck. The work presented in this paper corresponds to its pre-release 0.1.2. It contains the most basic functionalities, and is 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 system command (provided). The command builds 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 prototype. Some of these examples are documented in Sections 5 and 6. Moreover, syntaxes for OCaml and WhyML random and enumerative tests have been chosen to be as similar as possible.

The prototype 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. Each AutoCheck's input is represented by a rectangle with square corners. It is either a WhyML or an OCaml file (respectively named Tests.mlw or Tests.ml in the figure) containing the implementation under test and a description of tests. Since the properties to be tested and their tests respectively are ordinary OCaml or WhyML executable functions and function calls, and since OCaml and WhyML applications can be made up of multiple files, the implementations under test, their properties and the tests of these properties can be in a single file or provided in multiple files.

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 prototype for OCaml (whose main file is SCheck.ml). As detailed in Section 6.2 the latter encapsulates several enumeration programs from release 1.3 of ENUM library. This OCaml code, gathered in the file Enum.ml, is automatically extracted by Why3 from WhyML enumeration programs whose properties are proved with Why3, as de-

tailed in Sect. 3. The files QCheck.mlw and SCheck.mlw respectively define random and enumerative testing in WhyML, so that tests can be written in WhyML (in Tests.mlw in the figure) and their automated extraction with Why3 generates tests in OCaml, exploiting the random and enumerative testing functionalities for OCaml defined in *QCheck.ml* and SCheck.ml.

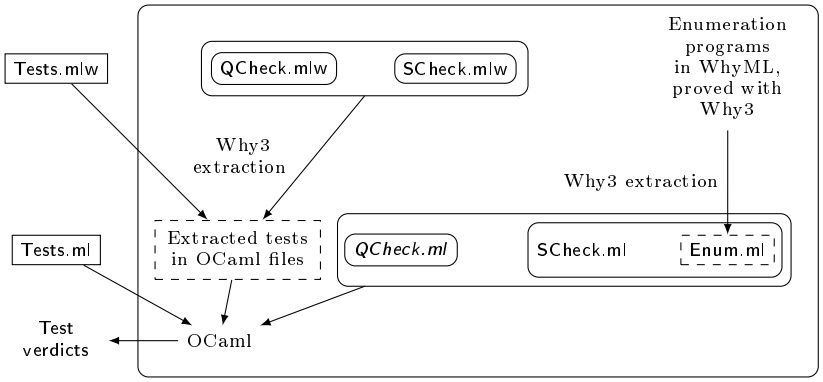

Fig. 1. AutoCheck workflow.

AutoCheck 0.1.2 is developed for release 1.4.0 of Why3. It exploits QCheck 0.17 and the SMT solvers Alt-Ergo 2.4.0, CVC4 1.6, Z3 4.7.1 and Z3 4.8.10.

5 Random testing for Why3

This section is a tutorial on random testing for WhyML properties with AutoCheck. The pre-release presented in this paper provides random generators for WhyML built-in types (`unit`, `bool` and Cartesian products) and some types from Why3's standard library (`(option 'a)`, `(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's standard library or relational properties between functions defined in that library. In order to eliminate any risk of confusion between a function under test returning a Boolean value and an executable property, the name of all the user-defined executable properties presented below is suffixed by `_prop`.

5.1 Basic types and Cartesian products

Example for the `unit` type. The most elementary type in WhyML (and OCaml) is `unit`. Its unique inhabitant is `()`. To illustrate counterexample generation, let us start with the false property "`()` is not an inhabitant of `unit`". The

property is implemented by the function `is_unit_prop` reproduced on Lines 1-2 in the Listing 1.4.

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

Listing 1.4. Test of a false property about the `unit` type.

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 executable property `is_unit_prop`. The function `QCheck_runner.run_tests` implements test execution.

Assume that the code in Listing 1.4 is in the module `RandomTests` of the file TestExamples.mlw. Then, the command

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

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

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

Test is_unit_prop failed (0 shrink steps):

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

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

Random tests for two Boolean functions. 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's standard library[1]. (This is not a language constraint, but, for clarity, the name of each Boolean function used or defined here ends with a 'b', when it is not intended to be an executable property.)

```
1  let function equivb (x y : bool) : bool
2  =
3    match x with
4    | True → y
5    | False → notb y
6    end
```

Listing 1.5. A user-defined function for Boolean equivalence.

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

Let us implement a Boolean function for equivalence and test this new function. Function `equivb` in Listing 1.5 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 the Boolean functions `equivb`, `andb`, `implb` and `notb` is implemented by the function `equivb_prop` on Lines 1-7 in Listing 1.6, taking a pair of Boolean values as input. In order to test this property (on Lines 11-12 in Listing 1.6) we define a random generator `bool_pair_arbitrary` of pairs of Booleans (Listing 1.6, line 9) 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 equivb_prop (x : (bool,bool)) : bool
2  =
3    let (a,b) = x in
4    match andb (implb a b) (implb b a) with
5    | True → equivb a b
6    | False → notb (equivb a b)
7    end
8
9  let bool_pair_arbitrary = QCheck.pair QCheck.bool QCheck.bool
10
11 let equivb_test
12 = QCheck_runner.run_tests (Test.make bool_pair_arbitrary equivb_prop)
```

Listing 1.6. Test of the relational property `equivb_prop` about the function `equivb`.

The execution output

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

indicates a successful test. The property has been tested by generating 100 test data and no counterexamples have been found. The complementary output "`ran 1 tests`" between parentheses, also produced by the third-party tool QCheck, can be confusing. It does not mean that the property has only been tested once, but that only one property has been tested. The default number of 100 test data can be changed by using the function `Test.make_count` instead of `Test.make`. For instance, the code

```
let equivb_test
= QCheck_runner.run_tests (
    Test.make_count bool_pair_arbitrary equivb_prop 10000)
```

defines a test of the `equivb_prop` property by random generation of 10000 data.

As detailed in Listing 1.8, let us now use the new executable property `equivb` to check the commutativity property of the `orb` function for disjunction, whose definition is recalled in Listing 1.7.

```
let function orb (x y : bool) : bool =
```

```
match x with
| False → y
| True → True
end
```

Listing 1.7. Function `orb`.

```
1  let orb_commut_prop (x: (bool,bool)) : bool
2  =
3    let (a,b) = x in equivb (orb a b) (orb b a)
4
5  let orb_commut_test = QCheck_runner.run_tests (
6    Test.make bool_pair_arbitrary orb_commut_prop)
```

Listing 1.8. Test of the property "`orb` is commutative".

Examples of random tests with integers. Now let us consider the WhyML type `int` for integers and its theory in Why3's standard library. Since WhyML integers represent unlimited mathematical integers, they are usually extracted to the arbitrary-precision integers of `Zarith` OCaml library [53]. 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's 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 $>$. The corresponding lemma is on Line 1 in Listing 1.9. This false property is implemented as shown on Line 3 and tested with two different random generators as shown on Lines 5-6 and 8-9 in Listing 1.9.

```
1  lemma Abs_gt0: ∀ x:int. abs x > 0
2
3  let wrong_abs_pos_prop (n: int) : bool = abs n > 0
```

```
4
5  let wrong_abs_pos_test1 = QCheck_runner.run_tests (
6    Test.make QCheck.(int_range (-100000) 100000) wrong_abs_pos_prop)
7
8  let wrong_abs_pos_test2 = QCheck_runner.run_tests (
9    Test.make QCheck.(int_range (-10) 10) wrong_abs_pos_prop)
```

Listing 1.9. Test of a wrong property, mutation of lemma `Abs_pos`.

The first test (lines 5-6) 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 8-9) 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 $[-1000..1000]$. The duration of both tests is about half a second.

Now, let us check

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

from Why3's standard library. We turn it into an executable property `abs_le_prop` (Listing 1.10, 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_prop (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_prop)
```

Listing 1.10. Test of Lemma `Abs_le`.

5.2 Option type

The option type in WhyML is defined in Why3's standard library by a module reproduced in Listing 1.11. In the type definition, `'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.

```
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 1.11. Definition of `(option 'a)` in Why3.

In WhyML a definition starting with `let predicate` simultaneously defines a logical predicate (for specifications) and an executable property (for computations). Thus, the function `is_none` implements (for free) the (false) property that "the only inhabitant of type `(option 'a)` is `None`".

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

Listing 1.12. Example of test for option type.

The listing 1.12 shows how to randomly test this property. For the option 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`. In this example, it uses a random generator of integers named `Qcheck.int`.

```
--- Failure ------------------------------------------------------------------
Test is_none failed (63 shrink steps):
Some (0)
```

QCheck always finds the counterexample `Some (0)`. Any term of the form `Some (_)` would be a counterexample for the wrong property claiming that the type `option` is only inhabited by `None`, but here shrinking is in action and the tool chooses the integer `0` instead of any integer.

5.3 Polymorphic lists

The basic theory of polymorphic lists in Why3's 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`. The false property that "the only inhabitant of type `(list 'a)` is `Nil`" can be directly tested with `is_nil`, as shown in Listing 1.13. Notice that WhyML lists are polymorphic but a generator of list elements (here, `QCheck.bool`, for Booleans) has to be provided to the test generator, fixing the actual type of the generated lists.

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

Listing 1.13. Test with `is_nil` function as property.

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 a property on lists can be constructed with the help of the recursive function `for_all` from Why3's standard library reproduced in Listing 1.14. This function returns `true` if and only if a given 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 1.14. Executable function `for_all` from Why3's standard library.

As an example, let us consider lists of integers and the parity property that "all list items are even". The parity of an integer is defined by the side-effect free function `is_even` on Line 1 of Listing 1.15. The parity property is implemented on Line 2 and tested on Lines 3-4.

```
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 1.15. 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.

5.4 Polymorphic arrays

The theory of polymorphic arrays in Why3's standard library specifies as follows a function `make` creating an array of length `n` whose elements are all initialized with value `v`:

```
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 }
```

Its second postcondition can be tested with random lengths in [0..1000] as follows:

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

let length_make_test = QCheck_runner.run_tests (
  Test.make QCheck.(int_bound 10000) length_make_prop)
```

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

AutoCheck specifies for WhyML the following two 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}
```

They are 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 random generator of integers for the length of the generated arrays, whereas the second uses an implicit random generator of natural numbers to choose this length.

6 Enumerative testing

The first pre-release of AutoCheck presented in this paper offers enumerative testing for the OCaml types `unit`, `bool`, `int`, `('a option)`, `(int array)` and `(int list)`, and for the corresponding WhyML types `unit`, `bool`, `int`, `(option 'a)`, `(array int)` and `(list 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.

Section 6.1 presents a basic example of an enumerative test for WhyML properties. Section 6.2 details the integration in AutoCheck of the certified enumeration programs presented in Sect. 3.

6.1 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 1.16 to test by enumeration the wrong lemma shown on Line 1 in Listing 1.9. The first (resp. second) test finds the counterexample `0` in around 3 seconds (resp. less than 1 second).

```
1  let wrong_abs_pos_test1 = SCheck_runner.run_tests (
2    Test.make SCheck.(int_range (-10000000) 10000000) wrong_abs_pos_prop)
3
4  let wrong_abs_pos_test2 = SCheck_runner.run_tests (
5    Test.make SCheck.(int_bound 10000) wrong_abs_pos_prop)
```

Listing 1.16. Enumerative tests of Lemma `Abs_gt0`.

We can notice that all these tests of this property take less than one second. However, a precise interval and luck are required for the random test to find a counterexample, whereas the enumerative test always finds a counterexample, even with a large interval of data. Thus, this example illustrates an advantage of enumerative testing over random testing.

This example and the one in Listing 1.9 make 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. This integration of random and enumerative testing should be reinforced on two points: instead of being defined in two modules presenting a similar interface, both modules could clone a single module defining a more abstract common signature. This would make it possible to mix random and enumerative generation, *e.g.* have an array generator that uses an enumerator for its length, up to a fixed size, and a random generator for the array elements.

6.2 Integration of certified enumeration programs

Enumerating integer arrays is realistic and useful when their length 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 length, can also be considered as a partial proof (by enumeration) of that property. In Sect. 3 we have presented several effective programs enumerating arrays satisfying given invariants, such as being sorted or duplicate-free, and their certification with Why3. This section presents their 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 1.17. The contract specification syntax (`requires`, `ensures`) is part of

[2] http://toccata.lri.fr/gallery/inverse_in_place.en.html, April 30, 2021

WhyML, its meaning is more detailed in Sect. 3.2. The predicate `is_permutation` used in this contract is similar to the predicate `is_permut` presented in Sect. 3.2. The function 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].

```
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 1.17. Inversion of a permutation in place, function contract.

Here we do not intend 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 1.17:

(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 us first observe that the deductive verification of these properties is highly non-trivial. First, the loop invariant proposed in the version of this example distributed with Why3 1.4.0[3] is made up of seven universal formulas and occupies ten lines of code. Second, Why3's most advanced automated proof strategy, named Auto level 3, does not overcome this proof. It is completed by an interactive proof step, applying the transformation `split_goal_right`. Thus, before looking for an interactive proof of these properties, it is relevant to test them.

Enumerative test session. Let us now detail how to test (P_1) by enumeration with AutoCheck, and how it works internally. The postcondition (`is_permutation a`) is not executable. In order to test it, the logical predicate `is_permutation` has to be implemented by an executable function returning a Boolean value, such as the following one:

```
let b_permutation (a: array int) : bool = b_range a && b_injective a
```

It implements the logical predicate `is_permutation` if the functions `b_range` and `b_injective` respectively implement the predicates `range` and `injective` defined in Sect. 3.2. We only detail the implementation `b_range` of the predicate

```
predicate range (a: array int) =
  ∀ i: int. 0 ≤ i < a.length → in_interval a[j] 0 n
```

[3] In the folder https://gitlab.inria.fr/why3/why3/-/blob/1.4.0/examples.

A naive (i.e., non-optimized) implementation of the predicate `injective` is similar. The definition

```
let predicate in_interval (x l u: int) = l ≤ x < u
```

is a specificity of WhyML. It is indeed both a logical predicate and an executable function, because it is also the case for the comparison operators $\leq$ and $<$ on integers. Thus, it can be used in a specification and in a program.

The function `b_range` in Listing 1.18 implements the predicate `range`. The universal quantification is implemented by a `for` loop that stops at the first array value not in the interval $[0..n-1]$. The postcondition (on Line 2) ensures that the function implements the logical predicate `range`: the function returns `true` if and only if the predicate holds for the input array `a`.

```
1 let function b_range (a: array int) : bool
2   ensures { result ↔ range a }
3 =
4   let n = a.length in
5   for j = 0 to n - 1 do
6     invariant { range_sub a 0 j n }
7     if not (in_interval a[j] 0 n) then return false
8   done;
9   true
```

Listing 1.18. Implementation of the predicate `range`.

A loop invariant (on Line 6) helps to prove the postcondition. It uses the following generalization of `range` which controls that each element of the subarray $a[l..u-1]$ is in the interval $[0...b-1]$:

```
predicate range_sub (a: array int) (l u b: int) =
  ∀ i: int. l ≤ i < u → in_interval a[i] 0 b
```

The function `b_permut` allows us to define an executable property for (P_1), as follows:

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

Then, an enumerative test of (P_1) with all permutations of size 6 is

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

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_prop`, such as

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

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

The functions `SCheck_runner.run_tests`, `Test.make` and `Scheck.permut_of_size` are automatically extracted into OCaml functions with the same names. The OCaml function `Test.make` builds a test case by assembling a serial and an executable property. A *serial* is an OCaml record grouping a printer of integer arrays, borrowed from the third-party tool QCheck, and a data generator. Here, the latter is a generator of permutations from ENUM library, automatically extracted with Why3 from a certified generator of permutations written in WhyML.

Each enumerative test 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 provided test oracle. 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_prop succeeds (ran 720 tests)
```

Property (P_2) is checked similarly.

Enumerative testing is suitable for arrays containing integers in a small interval, as it is the case for permutations here. For larger integer ranges, random generation is preferable.

7 Conclusion

In this work, we laid foundations for random and enumerative data generation to test properties expressed in OCaml or WhyML language. These properties are assumed to be provided as executable functions returning a Boolean value, providing the test oracle. They can be defined by the user or produced by an external tool.

We presented several design choices and illustrated them by an open-source prototype, named AutoCheck. The first originality of our work is to propose formal specification and formal proofs for the data enumeration programs, thus addressing the certifiability issue of automated testing tools. A second methodological proposition is to lighten the development of the tool for the WhyML language, by exploiting an existing extraction mechanism from WhyML to OCaml and a third-party random testing tool for OCaml. Another contribution is a tutorial, with elementary examples, that a beginner can follow to practice property-based testing on the supported types. We also explained code certification (Section 3) and the tool's architecture (Section 4). This allows OCaml and WhyML developers to contribute to its extension.

A major direction of future work is to design and implement mechanisms to automate the generation of properties to be tested, from formal not-yet-proved lemmas or function contracts, when it is reasonably feasible. For function contracts, this generation would work as a verification condition generator in the deductive verification method, with the difference that the generated verification conditions would have to be executable. AutoCheck has to be extended before

pretending to compete with industrial tools such as Quviq [46], the commercial version of QuickCheck. The presented certification of enumeration programs should be extended to the entire code. Data enumeration should be generalized to user-defined datatypes. The specification and certification of more efficient enumeration programs may also be explored. Another direction could be to integrate fuzzing, which has become very popular for property testing [27,1] and even coverage-guided fuzzing [39] which makes random testing more efficient.

Acknowledgements

The authors would like to thank the anonymous referees for their relevant suggestions. This work has been supported by the EIPHI Graduate School (contract "ANR-17-EURE-0002").

References

1. An introduction to fuzzing OCaml with AFL, Crowbar and Bun. https://tarides.com/blog/2019-09-04-an-introduction-to-fuzzing-ocaml-with-afl-crowbar-and-bun (2019)
2. Arndt, J.: Matters Computational - Ideas, Algorithms, Source Code [The Fxtbook] (2010), https://www.jjj.de/fxt/fxtpage.html
3. 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). Lecture Notes in Computer Science, vol. 3362, pp. 49–69. Springer-Verlag, Marseille, France (Mar 2004)
4. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, Lecture Notes in Computer Science, vol. 4334. Springer, Heidelberg (2007)
6. 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. TAP 2018. pp. 44–62. Springer International Publishing, Cham (2018), https://doi.org/10.1007/978-3-319-21215-9_7
7. Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 Platform (2018), http://why3.lri.fr/manual.pdf
8. The Boogie intermediate verification language., https://github.com/boogie-org/boogie/
9. Bulwahn, L.: The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. Lecture Notes in Computer Science, vol. 7679, pp. 92–108. Springer, Heidelberg, Kyoto, Japan (2012), https://doi.org/10.1007/978-3-642-35308-6_10
10. Cambridge, U., München, T.U., contributors: Isabelle (2021), http://isabelle.in.tum.de/
11. 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), http://dx.doi.org/10.1145/351240.351266

12. The Coq proof assistant (2021), http://coq.inria.fr/
13. Correnson, L.: Qed. Computing what remains to be proved. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. Lecture Notes in Computer Science, vol. 8430, pp. 215–229. Springer, Heidelberg (2014), http://dx.doi.org/10.1007/978-3-319-06200-6_17
14. Dolan, S.: Crowbar - Git. https://github.com/stedolan/crowbar (Feb 2021)
15. Dubois, C., Giorgetti, A.: Tests and proofs for custom data generators. Formal Aspects Comput. **30**(6), 659–684 (Jul 2018), https://doi.org/10.1007/s00165-018-0459-1
16. Dubois, C., Giorgetti, A., Genestier, R.: Tests and proofs for enumerative combinatorics. In: Aichernig, K.B., Furia, A.C. (eds.) Tests and Proofs. TAP 2016. Lecture Notes in Computer Science, vol. 6792, pp. 57–75. Springer, Cham (2016), https://doi.org/10.1007/978-3-319-41135-4_4
17. Duregård, J., Jansson, P., Wang, M.: Feat: functional enumeration of algebraic types. In: Proceedings of the 2012 Haskell Symposium. ACM SIGPLAN Notices, vol. 47, pp. 61–72. Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2364506.2364515
18. Erard, C., Giorgetti, A.: Bounded exhaustive testing with certified and optimized data enumeration programs. In: Testing Software and Systems. ICTSS 2019. Lecture Notes in Computer Science, vol. 11812, pp. 159–175. Springer, Cham (2019), https://doi.org/10.1007/978-3-030-31280-0_10
19. Property-based testing framework for JavaScript., https://github.com/dubzzz/fast-check
20. Filliâtre, J.-C., Pereira, M.: Itérer avec confiance. In: Journées Francophones Des Langages Applicatifs. JFLA 2016. (2016), https://hal.inria.fr/hal-01240891
21. Filliâtre, J.-C., Pereira, M.: A modular way to reason about iteration. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. Lecture Notes in Computer Science, vol. 9690, pp. 322–336. Springer, Cham (2016), https://doi.org/10.1007/978-3-319-40648-0_24
22. Fitinghoff, N.: Extraction of Rust Code from the Why3 Verification Platform. Ph.D. thesis, Luleå University of Technology (2019), http://www.diva-portal.org/smash/get/diva2:1303268/FULLTEXT02#page20
23. Genestier, R., Giorgetti, A., Petiot, G.: Sequential generation of structured arrays and its deductive verification. In: Blanchette, J.C., Kosmatov, N. (eds.) Tests and Proofs. TAP 2015. Lecture Notes in Computer Science, vol. 9154, pp. 109–128. Springer, Cham (2015), https://doi.org/10.1007/978-3-319-21215-9_7
24. Giorgetti, A., Lazarini, R.: Preuve de programmes d'énumération avec Why3. In: AFADL 2018. pp. 14–19 (2018), http://afadl2018.ls2n.fr/wp-content/uploads/sites/38/2018/06/AFADL_Procs_2018.pdf
25. Giorgetti, A., Dubois, C., Lazarini, R.: Combinatoire formelle avec why3 et coq. In: Magaud, N., Dargaye, Z. (eds.) Journées Francophones Des Langages Applicatifs. JFLA 2019. pp. 139–154 (2019), https://hal.inria.fr/hal-01985195
26. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from proof failures in SPARK. In: De Nicola, R., Kühn, E. (eds.) Software Engineering and Formal Methods. SEFM 2016. Lecture Notes in Computer Science, vol. 9763, pp. 215–233. Springer, Cham (2016), https://hal.inria.fr/hal-01314885
27. Herdt, V., Große, D., Le, H.M., Drechsler, R.: Verifying instruction set simulators using coverage-guided fuzzing(*). In: Teich, J., Fummi, F. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019. pp. 360–365. IEEE (2019).

https://doi.org/10.23919/DATE.2019.8714912, https://doi.org/10.23919/DATE.2019.8714912
28. Property-based testing library for Java 8+., https://github.com/JetBrains/jetCheck
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: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. Lecture Notes in Computer Science, vol. 9952, pp. 461–478. Springer and Springer, Cham (2016), https://doi.org/10.1007/978-3-319-47166-2_32
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 **27**(6), e1640 (2017), https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1640
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.: Programmation fonctionnelle. (2016), http://assert-false.science/callcc/Guyslain/Teaching/ProgFonc/Cours/cours-2-4-quickcheck
37. Nelson, J.: The design and use of QuickCheck. Blog post https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html (Jan 2017)
38. What is OCaml?, https://ocaml.org/learn/description.html
39. Padhye, R., Lemieux, C., Sen, K.: JQF: Coverage-guided property-based testing in Java. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 398–401. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (Jul 2019). https://doi.org/10.1145/3293882.3339002, https://doi.org/10.1145/3293882.3339002
40. Pereira, M.J.P.: Tools and Techniques for the Verification of Modular Stateful Code. Ph.D. thesis, Université Paris-Sud (2018), https://tel.archives-ouvertes.fr/tel-01980343/document
41. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: How testing helps to diagnose proof failures. Formal Aspects of Computing **30**(6), 629–657 (Jun 2018), https://doi.org/10.1007/s00165-018-0456-4
42. PROPerty-based testing tool for ERlang., https://github.com/proper-testing/proper
43. QuickCheck inspired property-based testing for OCaml., https://github.com/c-cube/qcheck
44. Module core_kernel.quickcheck (2020), https://ocaml.janestreet.com/ocaml-core/latest/doc/core_kernel/Core_kernel/Quickcheck/index.html
45. Randomized property-based testing plugin for Coq., https://github.com/QuickChick/QuickChick
46. QuviQ testing tools, http://www.quviq.com

47. Reich, J.S., Naylor, M., Runciman, C.: Advances in Lazy SmallCheck. In: Hinze, R. (ed.) Implementation and Application of Functional Languages. IFL 2012. Lecture Notes in Computer Science, vol. 8241, pp. 53–70. Springer, Berlin, Heidelberg (2013), https://doi.org/10.1007/978-3-642-41582-1_4
48. Reis, J.S.: Kaputt (2017), https://github.com/joaosreis/kaputt
49. Rieu-Helft, R.: Un mécanisme d'extraction vers C pour Why3. In: Journées Francophones Des Langages Applicatifs. JFLA 2018. pp. 203–209 (2018), https://hal.inria.fr/hal-01707376v1
50. 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
51. Theft: Property-based testing for C., https://github.com/silentbicycle/theft
52. WhyML Language Reference — Why3 1.3.3 documentation (2020), http://why3.lri.fr/doc/syntaxref.html#function-declarations
53. The zarith library., https://github.com/ocaml/Zarith