Bounded Exhaustive Testing with Certified and Optimized Data Enumeration Programs

Clotilde Erard and Alain Giorgetti

FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France
alain.giorgetti@femto-st.fr

Abstract. Bounded exhaustive testing (BET) is an elementary technique in automated unit testing. It consists in testing a function with all input data up to a given size bound. We implement BET to check logical and program properties, before attempting to prove them formally with the deductive verification tool Why3. We also present a library of enumeration programs for BET, certified by formal proofs of their properties with Why3. In order to make BET more efficient, we study and compare several strategies to optimize these programs.

Keywords: bounded exhaustive testing · formal verification · algorithmic efficiency

1 Introduction

Bounded Exhaustive Testing (BET, for short) automates unit testing of a function by checking one of its properties for all admissible inputs up to some size. Although this method is limited to small input data, its relevance is recognized [15,21] since it facilitates debugging by providing the smallest counterexamples, and provides confidence by guaranteeing the absence of errors below some size bound. This makes BET complementary to methods adapted to data of larger size, such as random testing. Whatever, the subject of this paper is not to compare BET with other test methods, but to improve the quality and availability of BET tools.

BET has first been used to check properties of functional languages, as exemplified by SmallCheck in Haskell [20]. Then, BET has been adapted to several proof assistants, e.g., to Isabelle in Quickcheck [4] and more recently to Coq, in an extension of QuickChick [14] named CUT (Coq Unit Testing) [7].

BET is also relevant to check properties produced by deductive verification, aka. *verification conditions* that a given program satisfies a given specification. We present a prototypical implementation of BET in the deductive verification tool Why3 [3]. 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 extraction to get correct-by-construction OCaml programs.

Some BET tools implement techniques such as constraint solving or local choice with backtracking, either to enumerate data or to derive enumeration programs from data definitions (see [6, Section 7] for references). However, these techniques may fail or enumerate data too slowly. For effectiveness, we consider BET using a distinct handwritten enumeration program for each family of data of interest. Dubois and Giorgetti proposed BET for Coq with such *custom* enumeration programs, defined either in Coq or in Why3 language [6].

Confidence in BET is increased if its enumeration programs are certified, ideally with formal proofs of their properties. Genestier et al. [10] developed a first version of the ENUM library, gathering enumeration programs in C language, formally specified with ACSL clauses and proved with Frama-C plugin WP for deductive verification. An adaptation to Why3 of a small fragment of this library has been presented to the French community [11,12]. Here we present a larger version of this library and its certification with Why3.

Another challenge for BET is to design and implement efficient enumeration algorithms. We examine here several ways to reduce their algorithmic cost: by implementing algorithms in a more efficient language (C versus WhyML), or by using optimized compilation. We also study the negative impact that these optimizations might have on certification.

The first contribution of this work is an implementation of BET to check Why3 properties (Sect. 2). The second contribution is a library of enumeration programs certified with Why3 (Sect. 3). The third contribution is an experimental study to optimize enumeration programs without sacrificing too much their certification (Sects. 4 and 5).

2 Bounded Exhaustive Testing for Why3

This section presents our implementation of bounded exhaustive testing for Why3 properties. It consists of a generic BET function (described in Sect. 2.2) and a library of enumeration programs (detailed in Sect. 3). All enumeration programs implement the same interface, described in Sect. 2.1. Two examples of BET are given in Sects. 2.3 and 2.4, respectively with success and exhibiting a counterexample.

2.1 Common Interface of Enumeration Programs

Since enumeration is a particular form of iteration, we specify and implement enumeration programs (sometimes hereafter called *generators*) by adapting the modular iterators defined by Filliâtre and Pereira [8,9]. Our generators modify a state, called a *cursor*, whose type is

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

in WhyML. The field `current` stores the last data generated so far. For simplicity, it is here a mutable array of integers, but other types can be used similarly. 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.

The generators presented in this paper are composed of two functions (declared on Lines 3 and 4 in Listing 1.1): 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.

2.2 BET Function

BET is implemented by the generic function `small_check` in Listing 1.1, whose execution tests the property defined by the `oracle` function (first parameter) for all data of size n (second parameter). The first parameter of the module `SmallCheck` (on Line 2) is a characteristic predicate of the enumerated data.

Note that the input type for the `oracle` function is a list rather than an array, because Why3 has limited support for function parameters that are functions working with mutable data. For the same reason, the generator functions cannot be input parameters for `small_check` function. Therefore we define them as module parameters (on Lines 3-4). They can be instantiated thanks to Why3's module cloning mechanism, as detailed in Sect. 2.3.

The return type `verdict` is composed of the field `witness` storing either a counterexample, if it exists, or the empty list (`Nil`) otherwise, and the field `rank` storing either the number of data tested when the witness is found, or the total number of tested data if there is no counterexample. The function `small_check` first creates the cursor (line 12), then converts the cursor array into a list (line 18), by using the `to_list` function from Why3 standard library. Finally, `small_check` tests each generated data with the `oracle` (line 19). If a counterexample is found, it is stored in the local variable `ce` (line 22), the enumeration is stopped and the function returns the counterexample and the number of data tested so far. Otherwise, the function stops when all data have been tested.

The `diverges` clause (on Line 10) declares that the function is not guaranteed to terminate. To prove its termination it is necessary to annotate its `while` loop with a *variant*, an integer expression whose value is non-negative before the loop and strictly decreases between two successive loop iterations. Defining a unique variant for all kinds of enumerated data is a challenging task out of the scope of the present study.

```
module SmallCheck
  predicate is_XXX (a: array int)
  val create_cursor (n: int) : cursor
  val next (c: cursor) : unit

  type verdict = { witness: list int; rank: int; }

  let small_check (oracle: list int → bool) (n: int) : verdict
    requires { n ≥ 0 }
    diverges
  =
    let c = create_cursor n in
    let ref r = 0 in
    let ref ce = Nil in
    while c.new do
      r := r+1;
      let a = c.current in
      let l = to_list a 0 a.length in
      if oracle l then
        next c
      else begin
        ce := l;
        c.new ← false
      end
    done;
    { witness = ce; rank = r }
end
```

Listing 1.1. BET function in WhyML.

2.3 Example of BET

We illustrate our BET for Why3 with functions and properties on permutations of a given size. Permutations on a finite set is an important topic in combinatorics and group theory. They have recently been formalized as injective endofunctions in Coq [6, Section 3]. The present example is the first step of an adaptation of that case study to Why3.

The permutation p on the set $[0..n-1]$ of first n natural numbers is encoded by the Why3 integer array `a` of its images, i.e., $\mathtt{a}[i] = p(i)$ 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 $[0...\mathtt{a.length} - 1]$ and `(injective a)` specifies injectivity of the function represented by a, i.e., uniqueness of values in a.

Let us consider the `reverse` function in Listing 1.2. The function reverses the order of the elements of its input array. For instance, it turns the array $\boxed{4}\boxed{1}\boxed{0}\boxed{7}$ into the array $\boxed{7}\boxed{0}\boxed{1}\boxed{4}$. It proceeds by exchanging symmetrical elements with

respect to the middle of the array. We want to prove that the function `reverse` preserves permutations. This property is specified by the precondition and the postcondition on Lines 2-3.

```
1 let reverse (a: array int) : unit
2   requires { is_permut a }
3   ensures { is_permut a }
4 =
5   let n = a.length in
6   let ref x = 0 in
7   let ref y = n-1 in
8   while x < y do
9     let v = a[x] in
10    a[x] ← a[y];
11    a[y] ← v;
12    y := y - 1;
13    x := x + 1
14  done
```

Listing 1.2. Reverse function under test.

Since WhyML predicates are not necessarly decidable, all specifications are ignored when a program is run. In particular, the postcondition (`is_permut a`) is not executable. In order to test it, a Boolean function implementing the logical predicate `is_permut` has to be provided. A Boolean function implementing a logical predicate, when it exists, is a *decision procedure* for this predicate. The Boolean function and a proof that it corresponds to the predicate are together called a *Boolean reflection*. This mechanism has several applications, e.g., proof automation [13].

The Boolean function

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

decides the predicate `is_permut` if `b_range` and `b_injective` respectively are decision procedures for the predicates `range` and `injective`. We only detail the Boolean reflection `b_range` of the predicate

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

a naive (i.e., non-optimized) implementation of the predicate `injective` being similar. The predicate

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

is a specificity of WhyML. It is indeed both a logical predicate and a Boolean function, because it is also the case for comparison operators on integers. Thus, we have its Boolean reflection for free.

The Boolean function `b_range` in Listing 1.3 is a decision procedure for 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 Boolean function decides 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 1.3. Boolean function `b_range`.

A loop invariant (on Line 6) helps to prove the postcondition. It uses the generalization

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

of `range` which controls that each element of the subarray $a[l..u-1]$ is in the interval $[0...b-1]$.

Whereas implementing a decision procedure is in general a difficult problem, it becomes simple for the family of first-order properties on integer arrays where all quantifications on array indices and values are bounded. All such universal quantifications ($\forall$) can be implemented by a `for` loop as in the former example, and implementing an existential quantification ($\exists$) is similar. Genestier et al. [10] showed that these array properties are common in combinatorics. They proposed a general pattern of Boolean reflection, when the properties are specified by ACSL predicates and implemented by Boolean functions in C language. The decidability property is proved generically, once for all, for all kinds of predicates. So, it holds for free (without requiring specific annotations) for each pattern instantiation. The adaptation of this feature to WhyML is left as future work.

```
1 use permutation.Permutation
2 use permutation.Enum
3
4 clone SmallCheck with
5   predicate is_XXX = is_permut,
6   val create_cursor = create_cursor,
7   val next = next
8
9 let test () : verdict
10   diverges
11 =
12   let n = 6 in
13   small_check (fun l → let a = to_array l in reverse a; b_permut a) n
```

Listing 1.4. Test program.

A simple program to test that the `reverse` function preserves permutations is presented in Listing 1.4. The declarations on Lines 1 and 2 import other modules. The module `Permutation` provides the predicate `is_permut` and its Boolean reflection `b_permut`. The module `Enum` provides a cursor and its functions to enumerate permutations. The declaration on Lines 4-7 imports a clone of the generic module `SmallCheck`, instantiated with the characteristic predicate `is_permut` and the enumeration functions for permutations. This cloning provides the type `verdict` and the right instance of the generic function `small_check` to test properties for all permutations with a given size. For the size $n = 6$ the test program (on Lines 9-13) uses this instance and an anonymous oracle function working as follows: as required by `small_check`, its input `l` is a list of integers. The function `to_array` from Why3 standard library transforms it into an array `a`, then reversed in-place by application of the `reverse` function. Finally the Boolean function `b_permut` is applied to the resulting array.

For efficiency and to get an explicit test result, the test code is executed in OCaml, after extraction of the test program and related modules. Thanks to some additional lines of OCaml code, the test result is displayed as follows:

```
Test passed. 720 data tested.
```

meaning that the test was successful for the $6! = 720$ permutations of size 6. This BET is executed in less than one second, in the environment used for the experimentation described in Sect. 4, where more efficiency results are provided.

The current prototype does not allow to set a time limit for BET, but it can be completed with this feature. The approach 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.

2.4 Counterexample

What happens if there is an error in a tested function? To illustrate the behavior of `small_check` in that case we inject an error on Line 9 of the `reverse` function (in Listing 1.2) that becomes the following one:

```
  let v = a[y] in
```

When running the same test (in Listing 1.4) for this erroneous version, the following output

```
Test failed after 1 test(s). Counterexample:
[0 1 2 3 4 5 ]
```

provides as counterexample a permutation that the false version of the `reverse` function transforms into the array

```
[4 4 3 3 4 4 ]
```

which is not a permutation. This BET discovers this error only after generating one test case. In general, more test cases may be required.

3 Certified Library of Enumeration Programs

ENUM is a library of certified enumeration programs for BET, freely distributed at https://github.com/alaingiorgetti/enum.[1] Its first releases were composed of C programs specified in ACSL language and verified with Frama-C plugin WP for deductive verification [10]. This section presents a new part of ENUM, composed of 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 [2] and have various applications in combinatorics.

Section 3.1 introduces some expected properties of these generators and their formalization in WhyML. Section 3.2 presents a simple way to define a generator, by filtering the output of another generator. Section 3.3 describes the techniques we use to assist formal proofs that the generators satisfy their expected properties. Finally, the content of the library is detailed in Section 3.4.

3.1 Properties

Each data enumeration program is expected to satisfy the following three 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 bounded-exhaustive enumeration.

Listing 1.5 shows a declaration of the enumeration functions with their contracts (pre- and postconditions) formalizing these properties in WhyML. 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.

[1] The work presented in this paper is in release 1.2 of ENUM.

```
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 1.5. Contracts of enumeration functions.

We assume that a predicate

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

encapsulates the data invariant. Then, the 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`.

3.2 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 show in this section that this *filtering* technique provides a specification, an implementation and a certification of the resulting enumeration program almost for free.

The generic module in Listing 1.6 formalizes filtering in WhyML. It provides an enumeration program for a family `Z` of integer arrays by filtering those arrays in a family `X` (characterized by the predicate `is_X`) that satisfy the additional constraint `is_Y`, implemented by the Boolean function `b_Y`. The module is parameterized by the predicates `is_X` and `is_Y`, the Boolean function `b_Y` and the enumeration functions `create_cursor_X` and `next_X` of `X` data. The module provides enumeration functions `create_cursor` and `next` of data in family `Z`.

The function `create_cursor` searches the first `Z` data by enumeration of `X` data started from the first one (given by `create_cursor_X`) and selection of the

first enumerated data satisfying `is_Y`, if it exists. (Otherwise, the field `c.new` is set to `false` by the function `next_X`.)

The function `next` proceeds similarly, but from the current cursor `c`. If the current data in the cursor is the last one satisfying `is_Z` but subsequent `X` data exist, then they are enumerated (by `next_X`) in the cursor. If furthermore none of them are in the `Z` family, then the cursor no longer contains a sound data. This is acceptable because, in that case, the `new` field is set to false. As specified on Line 11 of Listing 1.5, the cursor is expected to contain the maximal data only as input of the `next` function when it sets the `c.new` field to false, not necessarily as its output. It is possible to restore the maximal `Z` data in the output cursor, but this makes the generator less effective.

When the Boolean function `b_Y` decides the predicate `is_Y` and the enumeration functions `create_cursor_X` and `next_X` satisfy their contract given in Listing 1.5, the resulting enumeration functions `create_cursor` and `next` satisfy the same contract. This is automatically proved by Why3. So, it also holds for all instantiations of the module `Filter`, for free.

3.3 Auto-active and Interactive Verification

We combine the following two techniques to assist deductive verification of the enumeration programs. *Auto-active verification* [16] 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.

3.4 Contents of ENUM Library

Metrics on the library and its contents are collected in Table 1. 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 the 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.

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 [12] f of length n such that $0 \leq f[i] \leq i$ for $1 \leq i \leq n-1$.

```
module Filter
  predicate is_X (a: array int)
  predicate is_Y (a: array int)
  predicate is_Z (a: array int) = is_X a ∧ is_Y a

  val b_Y (a: array int) : bool
    ensures { result ↔ is_Y a }

  val create_cursor_X (n: int) : cursor
    requires { n ≥ 0 }
  val next_X (c: cursor) : unit

  let create_cursor (n: int) : cursor
    requires { n ≥ 0 }
    diverges
  =
    let c = create_cursor_X n in
    while c.new && not (b_Y c.current) do
      next_X c
    done;
    c

  let next (c: cursor) : unit
    diverges
  =
    if c.new then next_X c;
    while c.new && not (b_Y c.current) do
      next_X c
    done;
end
```

Listing 1.6. Filtering in WhyML.

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 ⊂ BARRAY	24	15	0	0	1.05
INJ ⊂ BARRAY	24	16	0	0	0.92
SURJ ⊂ BARRAY	34	25	0	0	1.1
COMB ⊂ BARRAY	17	10	0	0	0.84

Table 1. Verification results.

The second block concerns enumeration programs obtained by filtering (see Sect. 3.2). 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 $0 \leq c[0] < \ldots < c[n-1] \leq k-1$.

4 Experimentation Protocol

This section presents the experimental protocol we have designed in order to compare various ways of implementing, certifying and optimizing data enumeration programs. We consider two programming and specification languages, C/ACSL and WhyML, the properties detailed in Sect. 3.1, and the execution techniques (interpretation, extraction and compilation) detailed in Sect. 4.1. The goal of the experimentation is to answer the research questions detailed in Sect. 4.2.

All proofs and time measures were performed on a Ubuntu 18.04 virtual machine, with a Core i5-8259U processor.

4.1 Execution

There are several ways to run an enumeration program: With Why3 as interpreter (command `why3 execute`), by executing code compiled from OCaml source code extracted from WhyML code, or by compiling and executing C code, either extracted from WhyML code or written by hand. Indeed, Rieu-Helft [19] has developed a method to extract in C language a subset of programs written in WhyML. The C code can be compiled with `gcc` or with the certified C compiler CompCert [17]. Indeed, when you compile a program with an ordinary compiler like `gcc`, you have no assurance that the executed code has the same semantics as the source code. In contrast, the CompCert compiler is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues.

4.2 Research Questions

We gather experimental data in order to answer the following research questions. In a nutshell, RQ1 is about certification only, RQ2 about efficiency only and RQ3 about how to find a good compromise between both quality criteria.

RQ1: What is the most convenient approach to certify the enumeration programs? Since we have two versions, one in C/ACSL and another one in WhyML,

we want to compare the effort required to prove their properties with Frama-C/WP and Why3. We quantify this proof effort with the number of lines of specification. These numbers for WhyML version are in Table 1.
RQ2: What is the most efficient way to run our programs? The efficiency of our generators is estimated by computing their speed, i.e., the number of data generated per second, for all the ways to run our programs presented in Sect. 4.1. Indeed, we implement algorithms already optimal in memory, producing each data on the fly, starting from the data previously produced. Thus, only one data is stored in memory at a time.
RQ3: Since certification and optimization are two desirable but potentially antagonistic quality criteria, which language and tool combination provides the best compromise between both? From the answers to the former two questions we try to derive a good compromise between data generation speed and proof effort.

5 Experimentation Results

This section exploits experimental results to answer our research questions.

To answer RQ1 we first analyze some metrics collected in Table 1 for the version in WhyML and the metrics in Table 2 for the version in C/ACSL, for the most effective programs (the first 5 in Table 1, without filtering). These metrics are the numbers of lines of code and specification and the time required for proofs. The number of transformations is not comparable, as Frama-C/WP does not offer a transformation mechanism. We also do not compare the number of lemmas, because all lemmas in WhyML are used to prove completeness, but completeness is neither specified nor proved in the C/ACSL version. Nevertheless, the average proof time with C/ACSL is 1.69 times longer than with WhyML. The total numbers of lines of code and specifications are 76 and 154 in C/ACSL and 134 and 174 in WhyML program, i.e. not much more for one more specified property.

Array family	Code	Specification	Transformations	Lemmas	Time (s)
RGF	13	29	0	0	5.71
SORTED	13	32	0	0	5.50
PERM	24	35	0	0	22.98
BARRAY	13	29	0	0	5.19
FACT	13	29	0	0	5.14

Table 2. Verification results with the C/ACSL version.

Since the completeness property was not proved formerly with Frama-C/WP, we have tried to adapt to that environment its specification and successful proof with Why3. Although the adaptation of the specification to ACSL language did not require much effort, we have not yet managed to demonstrate any fragment

of the completeness property with Frama-C/WP. We assume that this is due to the different memory models used by Why3 and WP. A memory model defines links between the program variables and the mathematical terms used in the proof obligations. It represents a mapping of the memory, management processes (reading, writing, allocating, releasing) and their properties. While Why3 has a simple memory model for arrays, producing concise proof obligations, the WP memory model produces more complex proof obligations. This convinces us that Why3 is more convenient than Frama-C/WP for the certification of ENUM.

To answer RQ2 we compare the speed of data generation of various interpretations or compilations of implementations and extractions in WhyML, OCaml and C of the same enumeration algorithm. We consider an algorithm to enumerate permutations [2, page 243], and assume that speeds would be classified in the same order for other generators.

The first column of Table 3 gives the size of the generated permutations. The other columns display the number of millions of data generated per second, for four implementations and execution scenarios. A dash (-) indicates that generation exceeds the 6 hour time limit.

Size	WhyML	OCaml (extraction)	C (extraction)	C/ACSL (handwritten)
7	0.011	0.3	0.8	1
8	0.019	1.75	5.7	6.7
9	0.02	4.59	21.34	27.91
10	0.021	5.41	43.72	60.48
11	0.021	5.57	50.52	71.28
12	0.021	5.58	51.33	73.57
13	-	5.6	51.53	74.4
14	-	-	51.76	75.62

Table 3. Speed of data generation (number of millions of data per second).

The interpretation of WhyML code is the least efficient enumeration method. It is not surprising since the other methods include a compilation, usually more efficient than an interpretation. Next comes the execution of its extraction in OCaml. For instance, the OCaml program enumerates 5.58×10^6 permutations of size 12 in 1 second. This may be appropriate in some applications, but is well below the speeds of the C programs. Indeed, C is a low-level imperative programming language. It has been designed to provide low-level memory access, which allows it to reduce the memory allocation required and optimize performance, particularly through the use of pointers.

Although the extracted C code is behind the handwritten one, its speed is much higher than that of the OCaml code. Its performance allows us to continue our efficiency study only for the C code extracted from the WhyML code. Figure 1 shows data generation speeds for this C code compiled with `gcc` (without and with `-O3` optimization option) and CompCert compilers. This experiment

confirms the claim that code compiled with CompCert is about twice as fast as that compiled by `gcc` without optimization, and quantifies the claim that it is a bit slower than that compiled by `gcc` with higher levels of optimization[2]: the code compiled by `gcc` with its third level of optimization is about 40% faster than the one compiled by CompCert.

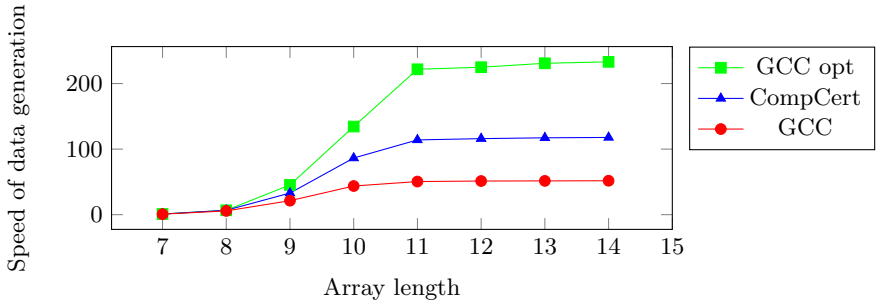

Fig. 1. Speed of data generation for different compilations.

To answer RQ3 we first draw some conclusions from the former answers to RQ1 and RQ2. Firstly, thanks to its elementary theory of arrays, Why3 makes it possible to prove more challenging properties – such as completeness – than Frama-C and its WP plugin. Moreover, C code automatically extracted from WhyML code is almost as fast as handwritten C code, for a much lower implementation effort. If a higher speed (resp. more confidence) is expected, the C code can be compiled with `gcc -O3` (resp. CompCert).

It remains to evaluate the additional effort required to specify and implement in WhyML enumeration programs suitable for C extraction. For the pointer-adapted permutation generator, we had to write 49 lines of code (only 7 lines more than for the original program), and 107 lines of specifications, so 21 lines more than the original code. The number of specification lines is mainly related to the fact that we control the memory manually. To download the proofs, we need 56.31 seconds, 3.44 times more than the original code. In addition, in the case of this program, completeness is not proved. Other generators (BARRAY and FACT) were also adapted for extraction. All properties were proven for these programs, but the specification effort was also greater than for their original codes. However, we noted that many specifications were common to all programs.

6 Conclusion

We have presented a prototypical implementation of a bounded exhaustive testing tool to check properties in the deductive verification tool Why3. It relies on

[2] http://compcert.inria.fr/compcert-C.html

enumeration programs which are specified, implemented and certified by formal proofs with Why3. The impact of several execution scenarios on their efficiency has been evaluated experimentally.

Obviously, we do not claim that BET and our prototype are competing with advanced property testing tools, such as QuickCheck and its commercial version QuviQ [1]. Such a comparison would be of little interest, because we pursue different goals. Our first goal is to certify the test tool, which as far as we know has already been done only for and with the Coq proof assistant, in the Quickchick tool [18]. Our second goal is to offer a free test tool to Why3 users, complementing prover-based counterexample generation [5].

This is ongoing work and directions for future work are numerous. First, the presented certification of enumeration programs should be extended to the entire testing tool. Data enumeration should be generalized to address functions with several parameters, complex datatypes (e.g. tree-like) and constraints between parameters. The specification and certification of more efficient enumeration programs may also be explored.

An important possible improvement concerns Boolean reflection, i.e., implementation and certification of a decision procedure for the characteristic predicate of test data. We have shown two applications of this procedure: as a test oracle, and as a filter to select the test data among a wider family. In the presented prototype the user has to write each procedure manually. A small-term objective is to provide her with an automated mechanism of derivation of these procedures, covering at least a first-order theory including integers and integer arrays.

Acknowledgements

The authors warmly thank Raphaël Rieu-Helft for his help in using extraction of WhyML programs in C, and Jean-Christophe Filliâtre for many suggestions.

References

1. QuviQ testing tools (2019), http://www.quviq.com
2. Arndt, J.: Matters Computational - Ideas, Algorithms, Source Code [The fxtbook] (2010), https://www.jjj.de/fxt/fxtpage.html
3. Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 Platform (2018), http://why3.lri.fr/manual.pdf
4. 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
5. Dailler, S., Hauzar, D., Marché, C., Moy, Y.: Instrumenting a weakest precondition calculus for counterexample generation. J. Logic Algebraic Methods Program. **99**, 97–113 (2018), https://doi.org/10.1016/j.jlamp.2018.05.003
6. 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

7. 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
8. Filliâtre, J.-C., Pereira, M.: Itérer avec confiance. In: Journées Francophones des Langages Applicatifs (JFLA 2016) (2016), https://hal.inria.fr/hal-01240891
9. 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
10. 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
11. 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
12. 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
13. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Formaliz. Reason. **3**, 95–152 (2010)
14. Hriţcu, C., Lampropoulos, L., Dénès, M., Paraskevopoulou, Z.: QuickChick: randomized property-based testing plugin for Coq (2018), https://github.com/QuickChick/QuickChick
15. Jackson, D., Damon, C.: Elements of style: analyzing a software design feature with a counterexample detector. IEEE Trans. Softw. Eng. **22**(7), 484–495 (1996)
16. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Usable Verification Workshop (2010), http://fm.csl.sri.com/UV10/
17. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
18. Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational property-based testing. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 325–343. Springer, Cham (2015), https://doi.org/10.1007/978-3-319-22102-122
19. Rieu-Helft, R., Marché, C., Melquiond, G.: How to get an efficient yet verified arbitrary-precision integer library. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 84–101. Springer, Cham (2017), https://doi.org/10.1007/978-3-319-72308-2_6
20. 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
21. Sullivan, K.J., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004. pp. 133–142. ACM (2004)