Sequential generation of structured arrays and its deductive verification

Richard Genestier[1], Alain Giorgetti[1,2], and Guillaume Petiot[1,3]

[1] FEMTO-ST Institute, University of Franche-Comté, 25030 Besançon CEDEX, France
firstname.lastname@femto-st.fr
[2] INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-les-Nancy, France
[3] CEA, LIST, Software Reliability Laboratory PC 174, 91191 Gif-sur-Yvette, France
firstname.lastname@cea.fr

Abstract. A structured array is an array satisfying given constraints, such as being sorted or having no duplicate values. Generation of all arrays with a given structure up to some given length has many applications, including bounded exhaustive testing. A sequential generator of structured arrays can be defined by two C functions: the first one computes an initial array, and the second one steps from one array to the next one according to some total order on the set of arrays. We formally specify with ACSL annotations that the generated arrays satisfy the prescribed structural constraints (soundness property) and that the generation is in increasing lexicographic order (progress property). We refine this specification into two programming and specification patterns: one for generation in lexicographic order and one for generation by filtering the output of another generator. We distribute a library of generators instantiating these patterns. After adding suitable loop invariants we automatically prove the soundness and progress properties with the Frama-C platform.

Keywords: formal specification, deductive verification, combinatorial enumeration, sequential generation, imperative program.

1 Introduction

Automated techniques for software testing are attractive because they produce many test cases in a more rational, reliable and affordable way than manual ones. We consider here unit testing for functions inputting a structured array. An array is said to be *structured* if it satisfies given constraints, such as being sorted or having no duplicate values. A challenge in input data generation for unit testing is to design and implement correct generators of complex data structures.

A recent trend in research in software verification aims at building verification environments that are themselves certified, in order to avoid erroneously validating safety properties of critical software. Randomized property-based testing has been formalized in Coq [15] to certify random generators. M. Carlier, C. Dubois and A. Gotlieb formally certify a constraint solver in Coq [6] as a piece of a certified testing environment. A certified constraint solver on a finite domain of arrays needs certified sequential generators of these structures to explore their domain. As a complement to random testing we address *Bounded Exhaustive Testing* (BET for short) with algorithms generating all

the arrays with given length and structure. We formally specify the behavior of these exhaustive array generators. As an alternative to interactive proving, we annotate them with loop invariants and variants, so that their formal contracts can be proved automatically. The BET approach is relevant [19] because it offers the advantage of providing counterexamples of minimal size, and errors in the function to be tested can often be revealed using input arrays of small size.

For a predefined total order on all the arrays of the same length, a *sequential generator* of all arrays with a given structure is composed of two functions: the first function constructs the smallest array of a given length satisfying the structural constraint, and from any array, the second function constructs the next array in that order satisfying the constraint. We present a uniform approach to the rational implementation of sequential generators of structured arrays. They are implemented as C functions and formally specified in the ANSI C Specification Language (ACSL) [2]. We consider three behavioral properties for the generation functions. The *soundness* property asserts that both functions generate arrays satisfying the prescribed constraints. The *progress* property asserts that the second function generates arrays in increasing lexicographic order. It entails the termination of repeated calls to the second function. The *exhaustivity* property asserts that the generator does not omit any solution. According to the deductive approach promoted by Floyd [11], Hoare [13] and Dijkstra [9], we statically verify the soundness and progress properties. In addition, we execute the generator up to some array length to check dynamically the exhaustivity property, either by counting or by comparison with the output of another generator. For deductive verification, we use the WP plugin [8] of Frama-C, which implements the Weakest Precondition calculus for C programs annotated in ACSL, assisted by SMT solvers [21] to prove the verification conditions generated by WP. Frama-C is a framework for the analysis of C programs developed by CEA LIST and INRIA Saclay.

We also propose programming and specification patterns to facilitate the design of the generation functions and the verification of their properties. A first pattern formalizes the principle of generation in lexicographic order by modifying the end of the array. A second pattern describes generation by filtering the output of an existing generator. It is completed by a pattern outlining how to uniformly transform a first-order constraint into a Boolean function.

The contributions of this paper are (i) general programming and specification patterns to speed up the construction and verification of sequential generators, (ii) a verified library of C programs and ACSL specifications implementing sequential generation algorithms, and (iii) automated formal proofs of their soundness and progress properties.

After giving some definitions, Section 2 presents generation in lexicographic order through a running example and a general pattern. Section 3 illustrates generation by filtering with the same example and proposes patterns that make this method easy to apply to obtain and verify many sequential generators. Verification results for several generators constructed from these patterns are presented in Section 4. Section 5 presents some related work, and Section 6 concludes.

2 Generation in lexicographic order

In all that follows, array values are mathematical integers. Bounded exhaustive generation of arrays only makes sense when there are finitely many arrays of each length. To this end array values are assumed to be lower- and upper-bounded by two C integers, whose absolute value is usually small, so that the number of arrays to generate does not become too large. Moreover it can often be assumed that all the computations for new array values are performed within these bounds. Under these assumptions, array values can be safely represented by C integers with the type **`int`**, without any risk of arithmetic overflows.

Let $<$ denote the strict total order on integers, such that $i < i + 1$ for any integer i.

Definition 1. *The* lexicographic order *on integer arrays, denoted by $\prec$, is such that $b \prec c$ if and only if there is an index i ($0 \leq i \leq n - 1$) such that $b[i] < c[i]$ and $b[j] = c[j]$ for $0 \leq j \leq i - 1$, for all integer arrays b and c of length $n \geq 0$.*

The binary relation $\prec$ is a strict total order. All the programs presented in the paper generate structured arrays in increasing lexicographic order.

Section 2.1 defines sequential generation functions. Sections 2.2 and 2.4 respectively present the principle of generation in lexicographic order through the example of a family of structured arrays and through a formal pattern, while Section 2.3 presents a formalization of the progress property.

2.1 Sequential generation functions

Consider a family `z` of structured C arrays of length `n` whose values are of type **`int`**. A sequential generator of arrays in this family consists of two C functions, called the *(sequential) generation functions*. The first function

```
int first_z(int a[], int n, ...)
```

generates the first array `a` of length `n` in the family `z`. It returns `1` if there is at least one array of this length in this family. Otherwise, it returns `0`. The second function

```
int next_z(int a[], int n, ...)
```

returns `1` and generates in the array `a` of length `n` the next element of the family `z` immediately following the one stored in the array `a` when the function is called, if this array is not the last one in the family. Otherwise, it returns `0`. The function `next_z` is thereafter called the *successor function*. In the header of these two C functions, the dots represent other parameters which may be required for the generation of the structured array. We only consider the cases where none of these parameters is an additional structure.

A typical program successively generating in the unique variable `a` all the arrays of length `n` in the family `z` consists of a call `first_z(a,n,...);` to the first function, a treatment of the first array, and then a treatment of all the subsequent arrays in the body of a loop **`while`** `(next_z(a,n,...) == 1)`.

2.2 Running example

Catalogs such as the *fxtbook* [1] propose effective sequential generators of combinatorial structures stored in a structured array. We consider the combinatorial structure of restricted growth function as a running example.

Definition 2. *A* Restricted Growth Function *(RGF, for short) of size n is an endofunction a of $\{0,\ldots,n-1\}$ such that $a(0)=0$ and $a(k)\leq a(k-1)+1$ for $1\leq k\leq n-1$.*

An endofunction a of $\{0,\ldots,n-1\}$, and thus an RGF, can be represented by the C array $\boxed{a(0)}\boxed{a(1)}\boxed{\ldots}\boxed{a(n-1)}$ of its n integer values. The *fxtbook* proposes an algorithm [1, page 235] to compute the RGF immediately following a given RGF a in ascending lexicographic order:

1. Find the maximum integer j such that $a(j)\leq a(j-1)$.
2. If this integer exists, increment the value $a(j)$ and fix $a(i)=0$ for all $i>j$. The other values of a remain unchanged.
3. Otherwise, the generation is complete, a is the largest RGF and remains unchanged.

For example, the five RGFs of size 3 generated by this algorithm are 000, 001, 010, 011 and 012. The first RGF is the constant function equal to 0.

```
1  #include "global.h"
2  /*@ predicate is_non_neg(int *a, ℤ n) = ∀ ℤ i; 0 ≤ i < n ⇒ a[i] ≥ 0;
3    @ predicate is_le_pred(int *a, ℤ n) = ∀ ℤ i; 1 ≤ i < n ⇒ a[i] ≤ a[i-1]+1;
4    @ predicate is_rgf(int *a, ℤ n) = a[0] == 0 ∧ is_non_neg(a,n)
5    @  ∧ is_le_pred(a,n); */
6
7  /*@ requires n > 0 ∧ \valid(a+(0..n-1));
8    @ assigns a[0..n-1];
9    @ ensures is_rgf(a,n); */
10 void first_rgf(int a[], int n);
11
12 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
13   @ requires is_rgf(a,n);
14   @ assigns a[1..n-1];
15   @ ensures is_rgf(a,n);
16   @ ensures \result == 1 ⇒ lt_lex{Pre,Post}(a,n); */
17 int next_rgf(int a[], int n);
```

Fig. 1. ACSL predicates and contracts of RGF generation functions (file `rgf.h`).

Figures 1 and 2 respectively show an ACSL specification and a C code for the sequential generation functions `first_rgf` and `next_rgf`. We explain through these examples the features of ACSL we use. To facilitate the reading of the specifications, some ACSL notations are replaced by mathematical symbols (e.g. keywords `\forall` and `integer` are respectively denoted by $\forall$ and $\mathbb{Z}$).

The file `rgf.h` given in Figure 1 and included in Figure 2 is composed of three predicates and two function contracts. The characteristic property of RGFs from Definition 2 is expressed between line 2 and line 5 of Figure 1 by the three ACSL predicates `is_rgf`, `is_non_neg` and `is_le_pred`. The constraint that the array values are in $\{0,\ldots,n-1\}$ is not specified because it is a consequence of the other constraints. In both function contracts an annotation `requires R;` specifies that the *precondition* `R` must be satisfied by the parameters of the function when it is called. On lines 7 and 12, we require that array `a` is of positive length `n` and is allocated in memory. It is also

required (line 13) that the input array `a` of the successor function represents an RGF. An annotation of the form **`assigns`** `A;` before the header of a function declares in `A` the function parameters and global variables that it can modify. Thus, line 14 declares that all the values of array `a` can be changed except the first one `a[0]`.

An annotation **`ensures`** `E;` asserts that the *postcondition* `E` holds at the end of the function execution. The **soundness property** asserts that all the generated arrays satisfy the prescribed constraint, for the corresponding function to be an RGF. It is formally specified on lines 9 and 15 of Figure 1. The postcondition on line 16 is explained in Section 2.3.

```
1  #include "rgf.h"
2
3  /*@ predicate is_zero(int *a, ℤ b) = ∀ ℤ i; 0 ≤ i < b ⇒ a[i] == 0; */
4
5  void first_rgf(int a[], int n) {
6   int k;
7   /*@ loop invariant 0 ≤ k ≤ n;
8     @ loop invariant is_zero(a,k);
9     @ loop assigns k, a[0..n-1];
10    @ loop variant n-k; */
11  for (k = 0; k < n; k++) a[k] = 0;
12 }
13
14 int next_rgf(int a[], int n) {
15  int rev,k;
16  /*@ loop invariant 0 ≤ rev ≤ n-1;
17    @ loop invariant (∀ ℤ j; rev < j < n ⇒ a[j] > a[j-1]);
18    @ loop assigns rev;
19    @ loop variant rev; */
20  for (rev = n-1; rev ≥ 1; rev--) if (a[rev] ≤ a[rev-1]) break;
21  if (rev == 0) return 0;
22  a[rev]++;
23  /*@ loop invariant rev+1 ≤ k ≤ n;
24    @ loop invariant is_non_neg(a,k);
25    @ loop invariant is_le_pred(a,k);
26    @ loop assigns k, a[rev+1..n-1];
27    @ loop variant n-k; */
28  for (k = rev+1; k < n; k++) a[k] = 0;
29  return 1;
30 }
```

Fig. 2. Effective generation of RGFs in C/ACSL.

The file `rgf.c` shown in Figure 2 is composed of one predicate and two function definitions specified in ACSL. The predicate `is_zero` defined on line 3 is introduced to express the loop invariant of the function `first_rgf` (line 8 of Figure 2). We now explain the C statements in the body of the function `next_rgf` in Figure 2. On line 20, a loop traverses the array from right to left to find a position from which the end of the array will be modified. This position is called the *revision index* of the array `a`. In this example, the revision index `rev` is reached when meeting the rightmost element (i.e. maximum index) less than or equal to its predecessor. If the search fails, then the final structure is reached (line 21). Otherwise, the contents of the array are changed from the revision index to the end, so that the new array also satisfies the constraint and is greater than the current array in lexicographic order. The way to effect this revision depends on the prescribed constraints of the array. For RGFs, the property `a[rev]` $\leq$ `a[rev-1]` of the revision index `rev` makes it possible to increment `a[rev]` (line 22) and fill the rest of the array with `0` (line 28) to obtain the next array satisfying the restricted growth

constraint. Figure 2 also shows annotations concerning the loops of the functions. An annotation **`loop invariant I;`** immediately before a loop states that the formula `I` is an *(inductive) invariant* of this loop, i.e., a property that holds the first time the loop is entered and is preserved by each iteration of the loop body. For instance, the loop invariant on line 17 asserts that the revision index is the rightmost position from which the end of the array can be modified to obtain a greater array satisfying the constraint. Before the second loop of the function `next_rgf`, three loop invariants successively assert that the loop variable `k` stays between `rev+1` and `n` (line 23), that the `k` first values of the array are non-negative (line 24), and that the property `is_le_pred` is satisfied up to `k` (line 25). The annotation **`loop assigns`** line 26 asserts that the only values that the loop body can change are the elements of `a` between the indexes `rev+1` and `n-1`. An annotation **`loop variant V;`** defines a *loop variant* `V` to ensure the termination of the loop. The entire expression must be non-negative at the beginning of each loop iteration and strictly decrease between two successive loop iterations. For example, as declared on line 27, the term `n-k` is a variant of the loop on line 28. The ACSL annotations in the body of the function `first_rgf` are similar and therefore not detailed.

Suppose that Figure 2 is the content of a file `rgf.c`. The static deductive verification of the function `next_rgf` with Frama-C and its plugin WP is realized by running the command `frama-c -wp-fct next_rgf rgf.c`. Frama-C indicates whether each proof obligation generated by WP is proved by the SMT solver Alt-Ergo and indicates the duration of each proof. Verification results are detailed in Section 4.

2.3 Progress property

The progress property asserts that the successor function generates arrays in increasing lexicographic order. It is specified in ACSL by the postcondition

`ensures \result` $==$ `1` $\Rightarrow$ `lt_lex{Pre,Post}(a,n);`

on line 16 in Figure 1. The ACSL formula `lt_lex{L1,L2}(a,n)` formalizes that the array `a` at label `L1` is lexicographically less than at label `L2`. A label represents a position in the program. Every expression `e` in ACSL can be written **`\at`**`(e,L)`, meaning that `e` is evaluated at the label `L`. The predefined label **`Pre`** (resp. **`Post`**) in the postcondition refers to the state before (resp. after) execution of the function `next_rgf`.

The predicate `lt_lex` and two auxiliary predicates formalize Definition 1 in a header file `global.h` included in all the generators. These definitions are shown in Figure 3.

```
1 /*@ predicate is_eq{L1,L2}(int *a, ℤ i) =
2   @  ∀ ℤ j; 0 ≤ j < i ⇒ \at(a[j],L1) == \at(a[j],L2);
3   @ predicate lt_lex_at{L1,L2}(int *a, ℤ i) =
4   @  is_eq{L1,L2}(a,i) ∧ \at(a[i],L1) < \at(a[i],L2);
5   @ predicate lt_lex{L1,L2}(int *a, int n) =
6   @  ∃ int i; 0 ≤ i < n ∧ lt_lex_at{L1,L2}(a,i); */
```

Fig. 3. Progress predicates (file `global.h`).

2.4 Pattern of generation in lexicographic order

The function `next_rgf` and the successor function of many other effective sequential generators of structured arrays follow a design principle here called "*suffix revision*". Figure 4 presents this principle as a design pattern composed of C code and ACSL annotations, for the successor function `next_z` of a sequential generator in lexicographic order.

The family `z` of structured arrays is defined by a constraint formalized by the predicate `is_z` declared on line 4 of Figure 4. The successor function `next_z` revises the suffix of its input array `a` in two steps. First, it finds the rightmost array index satisfying some predicate called the *revision condition*. This index is called the *revision index* of the array `a`. Second, it modifies the contents of the array `a` from the revision index to the array end. The revision condition is formalized by the predicate `is_rev` (line 5) and the Boolean function `b_rev` (declared and specified on lines 8-13). The loop on line 32 explores the input array `a` from right to left to find the revision index `rev`. The loop invariant on line 29 states that the revision index is the rightmost index satisfying the revision condition. If the search fails (line 33), the input array is the last one and the function returns `0`. Otherwise, the function `suffix` revises the array `a` from its revision index to its end, as specified by the **`assigns`** clause on line 17. Its postcondition on line 18 asserts that it increases the array value `a[rev]` at the revision index.

```
1  #include "global.h"
2
3  /*@ axiomatic preds {
4    @  predicate is_z(int *a, ℤ n) reads a[0..n-1];
5    @  predicate is_rev(int *a, ℤ n, ℤ i) reads a[0..n-1];
6    @ } */
7
8  /*@ requires n > 0 ∧ \valid(a+(0..n-1));
9    @ requires 0 ≤ rev ≤ n-1;
10   @ assigns \nothing;
11   @ ensures \result == 0 ∨ \result == 1;
12   @ ensures \result ⇔ is_rev(a,n,rev); */
13 int b_rev(int a[], int n, int rev);
14
15 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
16   @ requires 0 ≤ rev ≤ n-1;
17   @ assigns a[rev..n-1];
18   @ ensures a[rev] > \at(a[rev],Pre); */
19 void suffix(int a[], int n, int rev);
20
21 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
22   @ requires is_z(a,n);
23   @ assigns a[0..n-1];
24   @ ensures soundness: is_z(a,n);
25   @ ensures \result == 1 ⇒ lt_lex{Pre,Post}(a,n); */
26 int next_z(int a[], int n) {
27  int rev;
28  /*@ loop invariant -1 ≤ rev ≤ n-1;
29    @ loop invariant (∀ ℤ j; rev < j < n ⇒ ! is_rev(a,n,j));
30    @ loop assigns rev;
31    @ loop variant rev; */
32  for (rev = n-1; rev ≥ 0; rev--) if (b_rev(a,n,rev)) break;
33  if (rev == -1) return 0;
34  suffix(a,n,rev);
35  return 1;
36 }
```

Fig. 4. Successor function pattern for suffix revision.

The successor function follows the suffix revision pattern if it satisfies the soundness and progress properties (respectively specified on line 24 and 25), but also the property that the successor function always computes the next array, i.e. that there exists no array in the family `z` between its input and output arrays, for the strict and total lexicographic order. An ingredient for its specification is the loop invariant on line 29. Its verification is further discussed in Section 4.1.

Suppose that Figure 4 is the content of a file `suffix.c`. The postcondition expressing the progress property and the loop invariant on line 29 are automatically proved by WP, with the command

```
frama-c -wp suffix.c -wp-prop=-soundness.
```

Note that the loop invariant on line 29 is not required to prove progress property. Indeed, the algorithm implemented by the function `next_z` corresponds to the definition of the lexicographic order: it leaves a prefix of the array `a` unchanged and increases its value at the revision index. The progress property is thus generically verified, because it is a consequence of the pattern. On the other hand, the soundness property cannot be verified at this level of generality, because it depends on the constraint on the array `a`.

By instantiating the predicates `is_z` and `is_rev` and the subfunctions `b_rev` and `suffix` in this pattern with appropriate code, we obtain a generator of a family `z` of structured arrays in lexicographic order, whose progress property can be verified automatically, assuming that the subfunctions satisfy their contracts. In an instantiation of the pattern the subfunction contracts have to be completed so that the soundness property can also be verified automatically. For simple generators it is easier to replace the subfunction calls by a sequence of statements. For example, we can obtain the successor function `next_rgf` of Figure 2 by replacing the calls `b_rev(a,n,rev)` and `suffix(a,n,rev);` respectively by the statement

```
a[rev] ≤ a[rev-1];
```

and the sequence of statements

```
a[rev]++;
for (k = rev+1; k < n; k++) a[k] = 0;
```

3 Generation by filtering

Generation *by filtering* consists of selecting in a family of structures those that satisfy a given constraint. Of course, the more structures are rejected, the less effective is the generator. However this simple generation approach quickly provides a first generator, whose implementation, specification and deductive verification come almost for free, as we will see throughout this section.

Section 3.1 illustrates the principle of generation by filtering with the example of RGFs. Section 3.2 formalizes this principle in a general pattern for all generators by filtering. The soundness property of the generation functions in this pattern is automatically proved. To instantiate this pattern, it is necessary to implement the constraint of substructures as a Boolean function. Section 3.3 provides a general pattern for this Boolean function and its specification. The soundness of the Boolean function with respect to the constraint is also automatically proved.

3.1 Example

The RGF family is a subfamily of the family of endofunctions of $\{0, ..., n-1\}$. Suppose we already have implemented, specified and automatically verified a generator of endofunctions of $\{0, ..., n-1\}$ consisting of two generation functions `first_endofct(a,n)` and `next_endofct(a,n)`. Figure 5 shows a sequential generator of RGFs filtering those endofunctions of $\{0, ..., n-1\}$ that are RGFs. The generation functions `first_rgf(a,n)` and `next_rgf(a,n)` call the C Boolean function `b_rgf`, which characterizes an RGF among the endofunctions of $\{0, ..., n-1\}$. The ACSL predicate `is_rgf` here is the conjunction of the predicates `is_le_pred` and `is_endofct`. The contracts of functions `first_rgf` and `next_rgf` are not shown, because they are very similar to those of Figure 1, except the postcondition expressing the soundness property for `next_rgf` which is now

`ensures \result` $==$ `1` $\Rightarrow$ `is_rgf(a,n);`

```
1  /*@ requires n > 0;
2    @ requires \valid(a+(0..n-1));
3    @ requires is_endofct(a,n);
4    @ assigns \nothing;
5    @ ensures \result == 0
6    @  ∨ \result == 1;
7    @ ensures \result == 1 ⇔ is_rgf(a,n);
8    @*/
9  int b_rgf(int a[], int n) {
10  int i;
11  if (a[0] ≠ 0) return 0;
12  /*@ loop invariant 1 ≤ i ≤ n;
13    @ loop invariant is_le_pred(a,i);
14    @ loop assigns i;
15    @ loop variant n-i; */
16  for (i = 1; i < n; i++)
17   if (a[i] > a[i-1]+1) return 0;
18  return 1;
19 }
20
21 int first_rgf(int a[], int n) {
22  int tmp;
23  tmp = first_endofct(a,n);
24  /*@ loop invariant
25    @  tmp ≠ 0 ⇒ is_endofct(a,n);
26    @ loop assigns a[0..n-1],tmp; */
27  while (tmp ≠ 0 ∧ b_rgf(a,n) == 0) {
28   tmp = next_endofct(a,n);
29  }
30  if (tmp == 0) { return 0; }
31  return 1;
32 }
33
34 int next_rgf(int a[], int n) {
35  int tmp = 0;
36  /*@ loop assigns a[0..n-1], tmp;
37    @ loop invariant is_endofct(a,n);
38    @ loop invariant
39    @  le_lex{Pre,Here}(a,n); */
40  do {
41   tmp = next_endofct(a,n);
42  } while (tmp ≠ 0 ∧ b_rgf(a,n) == 0);
43  if (tmp == 0) { return 0; }
44  return 1;
45 }
```

Fig. 5. Generation of RGFs by filtering.

```
1 /*@ predicate le_lex{L1,L2}(int *a, int n) = lt_lex{L1,L2}(a,n)
2   @  ∨ is_eq{L1,L2}(a,n); */
3
4 /*@ lemma trans_le_lt_lex{L1,L2,L3}: ∀ int *a; ∀ int n;
5   @  (le_lex{L1,L2}(a,n) ∧ lt_lex{L2,L3}(a,n)) ⇒ lt_lex{L1,L3}(a,n); */
```

Fig. 6. Predicate and lemma to specify and prove progress of a filtering successor function.

The predicate and the lemma defined in Figure 6 are respectively introduced to specify on lines 38-39 a loop invariant for the filtering loop of the successor function and to automatically prove that invariant and thus the progress property for that function, assuming that the successor function of endofunctions ensures the progress property. The current array is indeed equal to the previous one at the beginning of the **`do .. while`** loop (lines 40-42). The pseudo-transitivity lemma `trans_le_lt_lex` helps the prover to derive the progress property – expressed with the strict order predicate `lt_lex` – from that loop invariant and the contract of the called function `next_endofct`.

3.2 General pattern of generation by filtering

The generation of RGFs by filtering can be generalized to any family of arrays defined from more general arrays by additional constraints. Figure 7 provides a general pattern for the generation of arrays in a family `z` by filtering arrays in a family `x` that satisfy the additional constraint `is_y` implemented by the Boolean function `b_y`. The arrays in the family `x` are assumed to satisfy the constraint `is_x`. The contracts of the generation functions `first_x`, `next_x`, `first_z` and `next_z` are similar to the one of the functions `first_rgf` and `next_rgf` in Section 3.1 and are therefore not reproduced in Figure 7. In this pattern, the ACSL predicates `is_x`, `is_y` and `is_z` are not defined. That's why they are declared in an ACSL axiomatic block, on lines 1-8 of Figure 7.

```
1  /*@ axiomatic preds {
2    @  predicate is_x(int *a, ℤ n)
3    @   reads a[0..n-1];
4    @  predicate is_y(int *a, ℤ n)
5    @   reads a[0..n-1];
6    @  predicate is_z(int *a, ℤ n) =
7    @   is_x(a,n) ∧ is_y(a,n);
8    @ } */
9
10 int first_x(int a[], int n);
11 int next_x(int a[], int n);
12
13 /*@ requires n > 0;
14   @ requires \valid(a+(0..n-1));
15   @ assigns \nothing;
16   @ ensures \result == 0
17   @  ∨ \result == 1;
18   @ ensures \result == 1
19   @  ⇔ is_y(a,n); */
20 int b_y(int a[], int n);
21
22 int first_z(int a[], int n) {
23  int tmp;
24  tmp = first_x(a,n);
25  /*@ loop invariant tmp ≠ 0
26    @  ⇒ is_x(a,n);
27    @ loop assigns a[0..n-1], tmp; */
28  while (tmp ≠ 0 ∧ b_y(a,n) == 0) {
29   tmp = next_x(a,n);
30  }
31  if (tmp == 0) { return 0; }
32  return 1;
33 }
34
35 int next_z(int a[], int n) {
36  int tmp;
37  /*@ loop invariant is_x(a,n);
38    @ loop assigns a[0..n-1], tmp;
39    @ loop invariant
40    @  le_lex{Pre,Here}(a,n); */
41  do {
42   tmp = next_x(a,n);
43  } while (tmp ≠ 0 ∧ b_y(a,n) == 0);
44  if (tmp == 0) { return 0; }
45  return 1;
46 }
```

Fig. 7. Pattern of generation by filtering.

Assuming that the functions `b_y`, `first_x` and `next_x` satisfy their specifications, the soundness and progress properties of the functions `first_z` and `next_z` are automatically proved by Frama-C and WP assisted by Alt-Ergo.

The generator of RGFs by filtering (Figure 5) is obtained by instantiating the general pattern as follows: Replace each `x`, `y` and `z` respectively by `endofct`, `rgf` and `rgf`, and implement the property `is_rgf` as a Boolean function. Other examples of sequential generators using this pattern are given in Section 4. Thus, from a specified generator for a family of structured arrays, one can rapidly implement, specify and verify generators of their subfamilies.

3.3 General pattern of Boolean functions

We also propose patterns for the ACSL contract and the C code of a Boolean function corresponding to an array structural constraint expressed in first-order logic. If the constraint is a Boolean combination of atomic predicates, the correspondence is obvious: Boolean operators (such as conjunction or negation) either exist in C or can be readily

expressed by a combination of C operators. Thus, the interesting cases are formulas with quantifiers. The case of a unique quantifier is too restrictive. The general case, where quantifiers are arbitrarily nested and combined with Boolean operators, would be too heavy to formalize, and the result would be painful to read. We have chosen to present the case of two nested quantifiers. This is enough to give an idea of what the general case would be, and this case is useful in itself.

```
1 /*@ axiomatic preds {
2   @  predicate is_x3(int *a, ℤ n, ℤ v1, ℤ v2) reads a[0..n-1];
3   @ }
4   @ predicate is_x2_gen(int *a, ℤ n, ℤ v1, ℤ v2) =
5   @  ∃ ℤ i2; 0 ≤ i2 < v2 ∧ is_x3(a,n,v1,i2);
6   @ predicate is_x2(int *a, ℤ n, ℤ v1) = is_x2_gen(a,n,v1,n);
7   @ predicate is_x1_gen(int *a, ℤ n, ℤ v1) = ∀ ℤ i1; 0 ≤ i1 < v1 ⇒ is_x2(a,n,i1);
8   @ predicate is_x1(int *a, ℤ n) = is_x1_gen(a,n,n); */
```

Fig. 8. Predicates for a constraint ∀∃.

```
1 /*@ requires n ≥ 0;
2   @ requires \valid(a+(0..n-1));
3   @ assigns \nothing;
4   @ ensures \result == 0 ∨
5   @  \result == 1;
6   @ ensures \result == 1 ⇔
7   @  is_x3(a,n,v1,v2); */
8 int b_x3(int a[],int n,int v1,int v2);
9
10 /*@ requires n ≥ 0;
11   @ requires \valid(a+(0..n-1));
12   @ assigns \nothing;
13   @ ensures \result == 0 ∨
14   @  \result == 1;
15   @ ensures \result == 1 ⇔
16   @  is_x2(a,n,v1); */
17 int b_x2(int a[],int n,int v1) {
18  int i;
19  /*@ loop invariant 0 ≤ i ≤ n;
20    @ loop invariant
21    @  ! is_x2_gen(a,n,v1,i);
22    @ loop assigns i;
23    @ loop variant n-i; */
24  for (i = 0; i < n; i++)
25   if (b_x3(a,n,v1,i) == 1) return 1;
26  return 0;
27 }
28
29 /*@ requires n ≥ 0 ∧ \valid(a+(0..n-1));
30   @ assigns \nothing;
31   @ ensures \result == 0 ∨
32   @  \result == 1;
33   @ ensures \result == 1 ⇔
34   @  is_x1(a,n); */
35 int b_x1(int a[],int n) {
36  int i;
37  /*@ loop invariant 0 ≤ i ≤ n;
38    @ loop invariant is_x1_gen(a,n,i);
39    @ loop assigns i;
40    @ loop variant n-i; */
41  for (i = 0; i < n; i++)
42   if (b_x2(a,n,i) == 0) return 0;
43  return 1;
44 }
```

Fig. 9. Pattern of Boolean functions for a constraint ∀∃.

Consider a constraint of the form $\forall i.\, 0 \leq i < n \Rightarrow (\exists j.\, 0 \leq j < n \wedge \varphi)$, where φ is a quantifier-free formula dependent on i and j expressing a constraint on an array of length n. In Figure 8 the constraint is decomposed into three ACSL predicates `is_x1(a,n)`, `is_x2(a,n,v1)` and `is_x3(a,n,v1,v2)`, respectively corresponding to the complete universal property, to the existential sub-formula and to the property φ it quantifies, for an array `a` of length `n`. The additional parameter `v1` of the predicates `is_x2` and `is_x3` corresponds to the free variable i in the subformulas $(\exists j.\, 0 \leq j < n \wedge \varphi)$ and φ. Similarly, the additional parameter `v2` of the predicate `is_x3` corresponds to the free variable j in φ. This quantifier-free formula φ being arbitrary in this pattern, the corresponding predicate `is_x3` is not defined, but only declared in an axiomatic block in lines 1-3 of Figure 8.

In Figure 9 the Boolean function `b_x1` implements the ACSL predicate `is_x1` in the sense that it returns `1` when its parameters satisfy the predicate, and `0` otherwise. The Boolean functions `b_x2` and `b_x3` respectively implement the predicates `is_x2` and

is_x3. For k =1,2, the function b_xk implements a loop that sequentially evaluates the predicate is_x(k+1) for all array elements. The loop invariants are specified using a generalization of the predicate is_xk, named is_xk_gen, defined in lines 4 and 7 of Figure 8.

Suppose that Figure 9 is the content of a file allex.c. Suppose that the Boolean function b_x3 satisfies its specification. By the command frama-c -wp allex.c -wp-skip-fct b_x3 we then automatically prove that the other functions satisfy their specification.

An immediate application of the previous pattern is the generation of surjections by filtering endofunctions. Indeed, a surjection f is an endofunction of $\{0, ..., n-1\}$ which satisfies the property $\forall i.\, 0 \leq i < n \Rightarrow (\exists j.\, 0 \leq j < n \wedge f(j) = i)$. A generator of surjections is easily obtained by merging the pattern of Boolean functions (Figures 9 and 8) with the one of generation by filtering (Figure 7), then by renaming x1, x2, x, y and z respectively as im, eq_im, endofct, im and surj, defining the predicate is_x3 as

```
predicate is_x3(int *a, ℤ n, ℤ v1, ℤ v2) = a[v2] == v1;
```

and implementing the function b_x3 with the unique statement

```
return(a[v2] == v1);
```

From the generator of endofunctions already used for RGFs in Section 3.1, the development and deductive verification of this surjection generator are effected in a few minutes. After this minimal work, we can make various simplifications to the surjection generator while preserving its deductive verification. For example, we can remove the parameter n of the predicate is_x3, which is not used in that example.

4 Verified library

The patterns presented in the previous sections have been implemented and instantiated to produce a library of verified sequential generators of structured arrays.[1] In order to ensure that there are finitely many arrays of each length, all the generators of the library refine a generator of arrays whose codomain is finite. This generator named fct generates functions from $\{0, \ldots, n-1\}$ to $\{0, \ldots, k-1\}$ in increasing lexicographic order.

The literature in enumerative combinatorics [1,17] provides many effective algorithms to generate classical combinatorial structures, such as n tuples, permutations or combinations of k elements from n. Using the patterns in Section 3, we quickly obtain generators of these structures by filtering among functions. Then we implement, specify and verify more effective generators from the literature by instantiating the pattern of suffix revision (Section 2.4). Finally we use the generators obtained by filtering to validate them, as detailed in Section 4.1.

[1] Archives enum.*.tar.gz of the library are available at http://members.femto-st.fr/richard-genestier/en and http://members.femto-st.fr/alain-giorgetti/en.

Array family	C	ACSL	goals	time (s)
suffix	9	12	25	1.929
filtering	14	33	51	5.524
allex	11	28	40	1.936
exall	12	27	40	1.921
all2	40	28	40	1.759
fct	13	25	42	5.622
subset	13	22	40	4.919
endofct	4	12	17	2.003
rgf $\subset$ endofct	25	27	69	4.958
sorted1 $\subset$ fct	19	27	67	4.743
sorted2 $\subset$ fct	28	48	103	6.464
comb $\subset$ fct	21	28	67	4.551
inj $\subset$ fct	29	42	91	6.031
surj $\subset$ fct	29	40	103	7.729
perm $\subset$ fct	30	42	91	7.713
endoinj $\subset$ inj	4	11	15	2.562
endosurj $\subset$ surj	4	11	15	2.638
perm = endofct $\wedge$ inj	17	21	60	7.211
perm = endofct $\wedge$ surj	28	40	102	9.647
invol $\subset$ perm	20	27	66	8.729
derang $\subset$ perm	20	27	66	8.611
rgf	13	28	41	8.598
sorted	13	30	44	28.445
comb	18	33	46	Timeout
perm	23	29	50	8.903

Fig. 10. Verification results.

Metrics on the library are collected in Figure 10. The first column gives the name of the families of structures generated. These names are explained in the remainder of this section. The number of lines of code (resp. ACSL annotations) is recorded in the second (resp. third) column. The soundness and progress properties of these programs have been proved automatically with Frama-C Neon 20140301 and its WP plugin assisted by Why3 0.82 and the SMT solvers Alt-Ergo 0.95.2, CVC3 2.4.1 and CVC4 1.3. The fourth column shows the number of proof obligations (goals) generated by WP. The fifth column gives the time needed for the proof of these goals by the provers Alt-Ergo, CVC3 and CVC4 in seconds on a PC Intel Core i7-3520M 3.00GHz $\times$ 4 under Linux Ubuntu 14.04.

The first block of lines in Figure 10 concerns the patterns presented in Sections 2.4 and 3.2. The patterns allex, exall and all2 respectively correspond to a first-order constraint of the form $\forall\exists$, $\exists\forall$ and $\forall\forall$. Note that only the progress property is proved for the pattern suffix. The second block concerns the above-mentioned generator fct.

The third block in Figure 10 concerns specializations, defined as follows. When a family of arrays has other parameters than their length, one may fix the value of some of these parameters and thus obtain other generators. We say that we have *specialized* the family. For example, the specialization of the family of functions from $\{0, ..., n-1\}$

to $\{0, \ldots, k-1\}$ for $k = 2$ gives Boolean functions encoding the family subset of subsets of a set of n elements [1, page 203]. Its specialization to the case where $k = n$ yields the family endofct of endofunctions of $\{0, \ldots, n-1\}$.

The fourth block in Figure 10 concerns generation by filtering (Section 3). We denote by z $\subset$ x a generator of structures z by filtering among more general structures x. We denote by z = x $\wedge$ y a generator of structures z by filtering among more general structures x the ones having the additional property of structures y. For instance, rgf $\subset$ endofct denotes a generator of restricted growth functions filtered among endofunctions (presented in Section 3.1). From the generator fct we generate by filtering the following families:

- Sorted arrays of length n whose elements are in $\{0, \ldots, k-1\}$, by comparing each array value to the following one if it exists (sorted1),
- sorted arrays of length n whose elements are in $\{0, ..., k-1\}$, by comparing each array value to each other, i.e. using the pattern all2 (sorted2),
- combinations of p elements selected from n (comb $\subset$ fct),
- injections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ for $n \leq k$ (inj $\subset$ fct) and
- surjections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ for $n \geq k$ (generator surj $\subset$ fct).

The combination $\{e_0, \ldots, e_{p-1}\}$ with $0 \leq e_0 < \ldots < e_{p-1} \leq n-1$ is represented by the function c from $\{0, \ldots, p-1\}$ to $\{0, ..., n-1\}$ such that $c(i) = e_i$.

Combining specialization and filtering, we produce four generators of permutations on $\{0, ..., n-1\}$ (structure perm):

- perm $\subset$ surj (resp. perm $\subset$ inj) by specialization of surjections (resp. injections) from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ (for $k = n$) and
- perm = endofct $\wedge$ inj (resp. perm = endofct $\wedge$ surj) by filtering of injections (resp. surjections) among endofunctions of $\{0, ..., n-1\}$.

The generator perm = endofct $\wedge$ surj was detailed in Section 3.3. By filtering from permutations we also obtain involutions on $\{0, ..., n-1\}$ (invol $\subset$ perm) and derangements (fixed-point free permutations) on $\{0, ..., n-1\}$ (derang $\subset$ perm).

Family	nb. goals	time (s)
fct	43	6.858
subset	41	7.277
rgf	42	8.760
sorted	45	8.007
comb	48	29.094
perm	51	9.595

Fig. 11. Verification results for effective algorithms with an additional assertion.

The fifth block in Figure 10 concerns effective generators in lexicographic order implemented by instantiating the pattern presented in Section 2. The generator rgf generates restricted growth functions with the algorithm from [1, page 235], as detailed

in Section 2.2. The generator sorted produces sorted arrays from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ in a more efficient manner than the generators sorted1 and sorted2. The generator comb produces combinations of p elements among n by the algorithm from [1, page 178]. The generator perm produces permutations on $\{0, ..., n-1\}$ by an adaptation of the algorithm from [1, page 243]. Column 4 shows that the proofs of these optimized generators are more complex, and thus take a longer duration, than for generators by filtering. Except for fct and subset, an extension of the timeout of the WP plugin to two minutes is required. In particular, the progress property of the generator comb is difficult to prove. However, an additional assertion

```
/*@ assert lt_lex_at{Pre,Here}(a,rev); */
```

at the end of the successor function substantially speeds up the longest proofs, as shown in Figure 11. Indeed, it specifies that the leftmost difference between the current array content (at label **Here**) and the former one (at label **Pre**) is at the revision index `rev`. This assertion helps the prover choosing the index `rev` to instantiate the existential quantifier in the predicate `lt_lex`.

4.1 Other properties

We have also proved the postcondition

```
ensures \result == 0 ⇒ is_eq{Pre,Post}(a,n);
```

for the successor functions `next_z` of the generators by suffix revision. It expresses the property that the array `a` is not modified when the function returns 0, i.e. when no revision index is found, indicating that the lexicographically maximal array is reached. This property does not hold for a generation by filtering that instantiates the pattern presented in Section 3.2, when the maximal array in the subfamily `z`, say `m_z`, is not the maximal array in the family `x`, say `m_x`. In that case the function `next_z` considers all the arrays greater than `m_z` in the family `x` until reaching `m_x` and returns `0` while the array content has changed.

Exhaustivity. They are several ways to check the exhaustivity property asserting that all the arrays with a given structure are generated. (i) One can store all the generated arrays in a global array and then specify and prove that it contains all the arrays satisfying the constraint. Exhaustivity was formalized in this way for a generator of all the solutions to the n-queens problem [10]. The formal proof of exhaustivity with the Why3 verification tool [3] needs interactive steps. We discard this solution because we want to offer an approach where the verification is completely automated. (ii) Another solution is to specify that the successor function indeed always computes the next array, i.e. that there exists no array with the given structure between its input and output arrays, for the strict and total lexicographic order. This quantification over arrays makes the property more difficult to prove automatically than the soundness and progress properties. (iii) When the soundness and progress properties are already proved, the exhaustivity property can be validated up to some array length simply by counting the number of generated arrays and comparing it to the expected number either from a sequence of

the On-Line Encyclopedia of Integer Sequences (OEIS) [20] or from known counting formulas implemented as C functions. This work follows this third way. We have performed the validation for all the structures of the library, by increasing length up to the limit of the largest positive representable integer, beyond the number of structures one may expect to generate in a reasonable time.

We have also performed validations of a generator with another one. In the context of this work this validation is easy to implement, firstly because we can quickly obtain a reference generator by filtering and secondly because this generator and the effective one it is compared to produce arrays in the same lexicographic order whenever the latter follows the principle of suffix revision. In that case, the storage of generated arrays is not necessary: it is enough to generate arrays in parallel with each generator and then test their equality. For this validation, the generators by filtering $\mathsf{rgf} \subset \mathsf{fct}$, $\mathsf{comb} \subset \mathsf{fct}$, $\mathsf{sorted} \subset \mathsf{fct}$ and $\mathsf{perm} \subset \mathsf{fct}$ were used as reference implementations to validate the optimized generators rgf, comb, sorted and perm.

5 Related work

Several techniques and tools help strengthening the trust in programs manipulating structured data. Randomized property-based testing (RPBT) consists in random generation of test data to validate given assertions about programs. RPBT has gained a lot of popularity since the appearance of QuickCheck for Haskell [7], followed by Quickcheck for Isabelle [4] and re-implementations for many programming languages, among which the C language with the tool quickcheck4c [23]. In RPBT a random data generator can be defined by filtering the output of another one, in a similar way as an exhaustive generator can be defined by filtering another exhaustive generator in BET.

A more generic approach is type targeted testing [18], wherein types are converted into queries to SMT solvers whose answers provide counterexamples. A more specific approach is contract-based testing, using contract languages. For Java programs the tools TestEra [14] and UDITA [12] automatically generate all non-isomorphic test cases within a given input size and evaluate soundness criteria, UDITA ensuring that some complex data structures are supported by the program. In case of violated soundness criteria, they produce concrete Java inputs as counterexamples, but the user has to write data generation methods and predicates. For C programs specified with ACSL, the tool StaDy [16] integrates the structural test generator PathCrawler [22] within the static analysis platform Frama-C. PathCrawler uses concrete execution and symbolic execution based on constraint solving and allows StaDy to guide the user in her proof work, showing inconsistencies between the code and the specification by the test coverage of all feasible paths of code and specification and producing counterexamples. StaDy helped us find errors in preliminary versions of some of our generators. Our work is even more specific: we provide a dedicated generator for each structure of interest. Although it requires more work, it is guaranteed to find the smallest counterexamples and is thus complementary to the other approachs.

Moreover we provide formally verified generators as building blocks for verified verification tools. To our knowledge, the deductive verification of exhaustive generators of constrained data structures has never been addressed yet.

6 Conclusion

The generation of arrays with a given structure in increasing length up to a given length can be very useful for automatically testing programs taking these arrays as inputs. Effective generation algorithms also provide interesting deductive verification problems. Therefore, we have undertaken to develop a library of structured array generators, formally specified and automatically verified. In order to reduce the cost of their specification and deductive verification, we propose general patterns for various families of generators, whose instantiation more easily yields correct programs. In particular, a pattern for the basic principle of filtering makes it possible to implement many generators from a small number of classical ones. The soundness and progress properties of these generators are automatically verified. We also provide a pattern for more effective generators, whose progress property is automatically verified. The soundness of the generators obtained by the instantiation of this pattern is more difficult to verify, but the Frama-C platform and its plugins provide significant help.

The use of deductive verification in combinatorics is not common. In this area, the most notable works are [5] and [10]. The first one specifies in ACSL and checks with Frama-C, a C function computing the conjugate of a partition of integers. The second work proves formally an enumeration of all the solutions to the n-queens problem. The formal proof is performed using the Why3 tool and the proof of exhaustivity is interactive.

Acknowledgments. The authors warmly thank J.-C. Filliâtre, J. Julliand, N. Kosmatov, C. Marché, T. Walsh and the anonymous referees for their suggestions and advice.

References

1. Arndt, J.: Matters Computational - Ideas, Algorithms, Source Code [The fxtbook] (2010), `http://www.jjj.de`
2. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, `http://frama-c.com/acsl.html`
3. Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 platform 0.81 (March 2013), `https://hal.inria.fr/hal-00822856`
4. Bulwahn, L.: The new Quickcheck for Isabelle. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer, Heidelberg (2012)
5. Butelle, F., Hivert, F., Mayero, M., Toumazet, F.: Formal proof of SCHUR conjugate function. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 158–171. Springer, Heidelberg (2010)
6. Carlier, M., Dubois, C., Gotlieb, A.: A certified constraint solver over finite domains. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 116–131. Springer, Heidelberg (2012)
7. 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)
8. Correnson, L.: Qed. Computing what remains to be proved. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 215–229. Springer, Heidelberg (2014)
9. Dijkstra, E.W.: A Discipline of Programming. In: Series in Automatic Computation, Prentice Hall, Englewood Cliffs (1976)

10. Filliâtre, J.C.: Verifying two lines of C with Why3: An exercise in program verification. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 83–97. Springer, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-27705-4
11. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society, Providence (1967)
12. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010. vol. 1, pp. 225–234. ACM, New York (2010)
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
14. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering. pp. 22–31. IEEE Computer Society, Washington, DC (2001)
15. Paraskevopoulou, Z., Hriţcu, C.: A Coq framework for verified property based testing (2014), http://prosecco.gforge.inria.fr/personal/hritcu/publications/verified-testing-report.pdf
16. Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: How test generation helps software specification and deductive verification in Frama-C. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 204–211. Springer, Heidelberg (2014)
17. Ruskey, F.: Combinatorial Generation Working Version (1j-CSC 425/520) (2003), http://www.1stworks.com/ref/RuskeyCombGen.pdf
18. Seidel, E.L., Vazou, N., Jhala, R.: Type targeted testing. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 812–836. Springer, Heidelberg (2015)
19. 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 (July 2004)
20. The OEIS Foundation Inc.: The On-Line Encyclopedia of Integer Sequences (2010), http://oeis.org
21. Weber, T.: SMT solvers: New oracles for the HOL theorem prover. International Journal on Software Tools for Technology Transfer 13(5), 419–429 (Oct 2011)
22. Williams, N.: Abstract path testing with PathCrawler. In: Proceedings of the 5th Workshop on Automation of Software Test, AST 2010. pp. 35–42. ACM, New York (2010)
23. Zito, A.: quickcheck4c: A QuickCheck for C (2014), http://oeis.org