# Random Number Generation in Ada 9X

K. W. Dritz*
Argonne National Laboratory
Argonne, IL 60439

The generation of random numbers[1] is central to many kinds of scientific study, especially those involving simulation or modeling. Most libraries of mathematical software have one or more *random number generators* (RNGs), encapsulating the best techniques for random number generation that have been reported in the literature, and at least rudimentary capabilities for generating random numbers are intrinsically provided in particular programming languages (among them, C and Fortran 90). The lack of a predefined RNG in Ada has inhibited the portability of application programs that need random numbers. With Ada 9X, that problem will cease to exist, at least for the vast majority of such applications.

The facilities for random number generation can be found in the Predefined Language Environment (Annex A) of Ada 9X [2], where they take the form of a pair of children of the predefined package `Ada.Numerics`. The package `Ada.Numerics.Float_Random` defines types and operations for the generation of floating-point random numbers, while the generic package `Ada.Numerics.Discrete_Random` plays a similar rôle for the generation of random integers (actually, random values of an aribtrary user-specified discrete subtype).

The content of these two packages underwent many changes after the initial proposal for an RNG facility, which did not even include the latter package. Perhaps more than for any other mathematical library routine, preconceived opinions about the form that the RNG should take ran strong, and many compromises were made before the final RNG was formulated. Nevertheless, certain goals for the design of the facility remained constant throughout the development process:

- The facility should be easy to use; it should appeal to the application programmer migrating from Fortran and should not require a major investment in learning new concepts unique to Ada.

*This chapter was written with financial support from Intermetrics, Inc., to Argonne National Laboratory under proposal No. P-91122.

[1] Technically, of course, we mean to say *pseudo-random numbers*, numbers in an algorithmically generated sequence that do not *appear* to be correlated and that satisfy some of the same statistical properties that truly random numbers satisfy.

- It should be possible to obtain a repeatable sequence of random numbers for use during program testing. When one is trying to understand aberrant program behavior, one needs to be able to rule out differences in the random numbers generated from one run to the next.

- It should be possible to obtain a unique sequence of random numbers in each production run of an application program. Changing from the repeatable mode to the unique mode should be straightforward.

- It should be possible to have multiple random number sequences, for those applications that require it. It should be possible to have separate RNGs in each task, as well as multiple RNGs in a single task. It should be possible to make multiple generators generate the same sequence or different sequences.

- It should be possible to save the state of an RNG and to restore an RNG to a previously saved state. This supports certain debugging requirements as well as the checkpointing and restarting of long-running applications.

- For testing purposes, it should be possible to examine the state of an RNG in an interactive debugger without advanced planning.

We discuss below how these goals have been realized in the Ada 9X RNG facility.

Implementations are free to exploit advances in the theory of random number generation, because Ada 9X does not prescribe the algorithm to be used. In order to provide some assurance that the algorithm used is minimally acceptable, the language prescribes tests of uniformity and randomness that must be satisfied by the implementation, and it also prescribes a lower bound on the periodicity of the RNG algorithm. Several popular RNG algorithms are known to pass the tests, including the venerable multiplicative linear congruential generator with multiplier $7^5$ and modulus $2^{31} - 1$ of Lewis, Goodman, and Miller [4] and both the add-with-carry and subtract-with-borrow Fibonacci generators of Marsaglia and Zaman [5]. Other algorithms that would be expected to pass, but which have not been explicitly tested, include the combination generators of Wichmann and Hill [6] and L'Ecuyer [3] and the $x^2 \bmod N$ generators of Blum, Blum, and Shub [1]. In order to allow users to assess the suitability of the algorithm for their particular application, the implementation must describe the algorithm it uses and must document some of its properties.

The predefined floating-point RNG package has the specification shown in Figure 1. The first point to note about this package is that it defines both *basic facilities*, which are expected to be needed by all or most applications of random numbers, and *advanced facilities*, which are expected to be needed only by those few applications having advanced or specialized needs. Most programmers will need to learn and use only the basic facilities. In fact, to get started, one need only "with" and "use" the package, declare a generator, and invoke the `Random` function on the generator, as illustrated in Figure 2. Another point to note is

```
package Ada.Numerics.Float_Random is

   -- Basic facilities

   type Generator is limited private;

   subtype Uniformly_Distributed is Float range 0.0 .. 1.0;
   function Random (Gen : Generator)
      return Uniformly_Distributed;

   procedure Reset (Gen       : in Generator;
                    Initiator : in Integer);
   procedure Reset (Gen       : in Generator);

   -- Advanced facilities

   type State is private;

   procedure Save  (Gen        : in  Generator;
                    To_State   : out State);
   procedure Reset (Gen        : in  Generator;
                    From_State : in  State);

   Max_Image_Width : constant := implementation-defined integer value;

   function Image (Of_State    : State)  return String;
   function Value (Coded_State : String) return State;

private

   ...  -- not specified by the language

end Ada.Numerics.Float_Random;
```

Figure 1: Specification of `Ada.Numerics.Float_Random`

```
with Ada.Numerics.Float_Random; use Ada.Numerics.Float_Random;
procedure Simple_Application is

   RNG : Generator;
   X : Float;
   ...

begin

   loop

      ...
      X := Random(RNG);
      ...

   end loop;

end Application;
```

Figure 2: Example of a simple use of `Ada.Numerics.Float_Random`

that only uniformly distributed random numbers are provided, and they lie in the customary range of 0.0 to 1.0.[2] The uniform distribution is the one most commonly encountered in practice; other distributions can be obtained from the uniform distribution by techniques covered in standard textbooks.

The example in Figure 2 shows that random number generators in Ada 9X are associated with objects of the type `Generator`. Each such object should be regarded as the source of a sequence of random numbers, successive elements of which can be obtained by applying the `Random` operation (as a function) to the object. The current "position" in the sequence is internal state information that is hidden from the user by virtue of the fact that `Generator` is a private type.

As the example illustrates, the ease with which one can begin to use the floating-point random number generator is likely to appeal to beginning application programmers, who may have some experience with Fortran but little or none with Ada. In particular, it is not necessary for the programmer to learn how to use generics in order to use the floating-point random number generator, and this feature goes a long way toward meeting the first design

[2]Some implementations may be incapable of generating either or both endpoints of the range, but application programmers are forewarned by a note in the reference manual not to depend on that. The subtleties of mapping this range into a particular range of integers was one of the motivations for providing a separate discrete random number generator package.

goal. Floating-point random numbers are provided only in the predefined type `Float`, which programmers migrating from Fortran are likely to use instinctively; random numbers of some other floating-point type can be obtained by explicit conversion.[3]

The second design goal, provision for obtaining repeatable sequences of random numbers (for program testing), is met by making that the default behavior of generator objects. That is, each generator is implicitly initialized to the same fixed state. One has to go a little out of the way to obtain a unique sequence of random numbers in each run. Thus, during the initial stages of program testing, when one has not thought to turn on all the bells and whistles, there is no danger of obtaining nondeterministic behavior (which could confound the testing process).

When one has progressed far enough in program testing to want a unique sequence of random numbers in each run, one merely needs to insert a call to the "time-dependent reset operation" in the program before the first call to `Random`. The `Reset` procedure comprises three overloadings, all of which reset the state of the generator given as a parameter; they differ in the nature of the resetting that is performed. When `Reset` is called with no parameters other than a generator, the action is to reset the generator to a time-dependent state; an example is shown in Figure 3. According to the reference manual, two calls to the time-dependent reset operation are guaranteed to establish different states if the calls are made at least one second apart, and not more than fifty years apart; this is certainly sufficient for priming a generator to yield unique sequences in different runs of the application. The time-dependent reset operation supports the third design goal.

One may declare any number of generator objects, and they can be aggregated into arrays or made components of other objects with arbitrary structure. Thus, multiple generators can be created trivially, either within a single task or in each of several tasks.[4] Unless one of the reset operations is used, however,

[3]We decided that learning how to perform explicit numeric conversions was less of a challenge to inexperienced programmers than learning how to use generics. In most cases, such conversions will serve merely to satisfy the strong typing requirements and will not affect the value of the random number; but if the conversion is to a type with greater or lesser precision, the value may be affected. The decision to avoid generics, thereby giving the user no direct way to request random numbers of different precisions, was also made partly on the basis of the implementation burden that it would have created. Floating-point random numbers of exceptionally high precision are required by only a few very specialized applications, and it is reasonable to expect those applications to shoulder the burden of providing high-precision random numbers. They can do so by building on the facilities that have been provided; for example, they can scale and combine the results of two or more calls to `Random` to obtain a single high-precision random number.

[4]We considered an alternative design that uses generics and associates a single, implicit generator with each instance. Having certain advantages as well as disadvantages, the alternative was rejected because it does not allow for generators to be components of other objects, such as arrays, or be allocated dynamically—capabilities that might reasonably be needed in some advanced applications.

all such generators will be started in the same state and will yield the same sequence. While that result may well be what is desired, it is more likely that each generator is intended to yield a different sequence. To satisfy that need, which is expressed by the fourth design goal, we provide an "initiator-dependent reset operation" as one of the overloadings of the `Reset` procedure, the one that takes an integer parameter named `Initiator` in addition to a generator. The idea is to reset each generator with a distinct initiator value before using any of them to generate random numbers; for example, if there are $n$ generators, each may be reset by a different initiator value in the range 1 to $n$. The semantics of the initiator-dependent reset operation are such that, if the characteristics of the implementation permit, each possible value of the initiator will initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is impossible to achieve in a given implementation, then the mapping between initiator values and generator states is required, at least, to be a rapidly varying function of the initiator value.

This technique for starting multiple generators in different states suffices when repeatable program behavior is desired in each run. A more elaborate technique is required when, in addition to being different from each other, one desires the initial generator states to be unique in different runs. For example, one might generate the initiator values randomly, using the discrete random number generator, after having initialized the latter to a time-dependent state. If a wide enough range is requested during the instantiation of the discrete random number generator package, only a small probability exists that a given initiator value will be generated more than once; nevertheless, it would be wise to filter out any duplicates that do happen to be generated, so that each of the floating-point generators can definitely be started in a unique state.

The advanced facilities of `Ada.Numerics.Float_Random` are concerned with saving and restoring generator states and with examining or manipulating generator states in the form of (implementation-defined) strings. Because Ada 9X does not prescribe the RNG algorithm to be used, it also imposes no requirements on the representation of generator states.

The fifth design goal addresses a long-running application's need to checkpoint its state so that it can later be restarted from the same state. Since the state of a random number generator (which is part of the application's state) is implicit, or hidden from the user, operations have been provided to obtain the state from a generator (by means of the `Save` procedure) and to reset a generator to a previously obtained state (by means of the third overloading of the `Reset` procedure). To store a state explicitly, one needs to declare a variable of the type `State`. One way that these advanced facilities can be used to checkpoint and restart a generator's state is illustrated in Figure 3.[5]

It should be emphasized that simple applications, like that illustrated in

[5]For the sake of simplicity, the file in which the generator's state is saved between runs is assumed to exist, in this example.

```
with Ada.Numerics.Float_Random; use Ada.Numerics.Float_Random;
with Ada.Sequential_IO;
procedure Checkpoint_Restart_Application is

   RNG : Generator;
   X : Float;
   RNG_State : State;
   type Run_Types is (Fresh_Start, Restart);
   Type_Of_This_Run : Run_Types;

   package State_IO is new Ada.Sequential_IO (State);
   use State_IO;
   State_File : File_Type;

begin

   Type_Of_This_Run := ...;
   case Type_Of_This_Run is
   when Fresh_Start =>
      Reset (RNG);  --  Time-dependent reset
   when Restart =>
      Open (State_File, In_File, Name => ...);
      Read (State_File, RNG_State);
      Close (State_File);
      Reset (RNG, RNG_State);  --  Reset from previously saved state
   end case;

   ...
   X := Random(RNG);
   ...

   --  Checkpoint the generator

   Save (RNG, RNG_State);  --  Save current generator state
   Open (State_File, Out_File, Name => ...);
   Write (State_File, RNG_State);
   Close (State_File);

end Checkpoint_Restart_Application;
```

Figure 3: Example of checkpointing and restarting a generator state

Figure 2, will have no need to declare variables of type `State`.

Since `State` is a private type, there is no way to examine or manipulate a state that has been exported from a generator. However, in some circumstances, particularly in advanced applications, there may be a need to do so, as enunciated by the sixth design goal. For this purpose, we have provided `Image` and `Value` functions, which are functions that convert state values to string values, and vice versa. The named number `Max_Image_Width` gives an upper bound on the size of the string representation of a state.

Even though the string representation of a state is implementation defined, one can perform some tasks with these strings portably. For example, the strings can be printed. Thus, if one observes aberrant behavior while using a symbolic debugger for program testing, one can obtain the current state of a generator, print its image on the terminal, and write it down for later reference. One can reverse the process by entering interactively a valid state in its string form, converting it to the corresponding state, and then resetting a generator to that state.

In more demanding applications, perhaps involving experimentation with random number generators, one can use information provided by the implementation on how it maps between strings and states to construct an arbitrary state by assembling the corresponding string. Note that the `Value` function must validate the string it is given, and must raise `Constraint_Error` if given a string that is not the image of a state. But this is the only time that state information must be checked for validity. In the more usual type of RNG design, the state information is exposed and often defined in detail, thereby creating the possibility of corruption or, alternatively, requiring that the operation for generating the next random number validate the state each time it is invoked.

The predefined discrete RNG package, `Ada.Numerics.Discrete_Random`, has an almost identical specification (not shown here). The main difference is that it is generic and must be instantiated with a discrete subtype before use. It exports the same entities as `Ada.Numerics.Float_Random`, with the exception that its `Random` function delivers a value of the generic formal subtype `Result_Subtype`, instead of the `Uniformly_Distributed` subtype of `Float`.

Although it is not difficult to convert, or map, floating-point random numbers in the range 0.0 to 1.0 into integers in some range, or into the range of any other discrete subtype, some subtleties arise at the endpoints 0.0 and 1.0 with certain techniques. `Ada.Numerics.Discrete_Random` is provided partly to head off those potential problems and partly to allow implementations to gain efficiency, when the end goal is random integers, by staying entirely within the integer domain. Nevertheless, some unusual applications may not be well served by the discrete RNG package, because it cannot readily be used to generate (say) random integers in a different range on each call; the range of the `Random` function is fixed at instantiation time. An application that has such a requirement would be better off generating random floating-point numbers and mapping them into the desired dynamically varying range. A note in the reference manual gives a reliable technique for performing that mapping. If `FG` is a floating-point generator and `M` has an integer value greater than zero, then the expression `Integer(Float(M) * Random(FG)) mod M` yields an integer uniformly distributed in the range 0 to $\mathtt{M} - 1$.

## References

[1] L. Blum, M. Blum, and M. Shub. *A Simple Unpredictable Pseudo-Random Number Generator.* SIAM Journal of Computing 15(2):364–383, 1986.

[2] ISO/IEC DIS 8652. *Information technology — Programming languages — Ada.*

[3] P. L'Ecuyer. *Efficient and Portable Combined Random Number Generators.* Communications of the ACM 31(6):742–749, 774, 1988.

[4] P. A. Lewis, A. S. Goodman, and J. M. Miller. *A Pseudo-Random Number Generator for the System/360.* IBM System Journal 8(2):136–146, 1969.

[5] G. Marsaglia and A. Zaman. *A New Class of Random Number Generators.* Annals of Applied Probability 1(3):462–480, 1991.

[6] B. A. Wichmann and I. D. Hill. *An Efficient and Portable Pseudo-Random Number Generator.* Applied Statistics 31:188–190, 1982.