
Evolvable Random Number Generators: A Few Ants in Your Hardware could be a Good Thing.

Jason C. Isaacs

Robert K. Watkins

Simon Y. Foo

Department of Electrical Engineering
FAMU-FSU College of Engineering,
Tallahassee, FL 32310 USA

Abstract - We have recently succeeded at using a Genetic Algorithm (GA) to evolve Random Number Generators (RNG) on a Field Programmable Gate Array (FPGA). In our experiments, decoded chromosomes produce bit streams, which are then tested for “fitness” as RNG. The GA's fitness function is a metrized version of the Federal Information Protection Standards (FIPS)-140. Once identified, fit chromosomes “breed” preferentially, and over generations, the RNG evolve. Offline testing with the Diehard test battery provides more stringent evidence of the evolved RNGs' fitness.

In our search for a PRNG that can be efficiently evolved in hardware, we began with well-known RNG, for example, Linear Congruential (LCGs), Multiply with Carry (MCGs), Linear Feedback Shift Registers (LFSRs). Although many of these algorithms map well to our hardware, they either consistently fail statistical tests or are limited in their evolvable characteristics. This failure led us to develop an Evolvable Characteristic-based RNG (ECRNG). In our encoding for the ECRNG, 26 integer-encoded genes, range [0, 15], represent mathematical operators and seed/parameters. Acting upon an initial seed, with reference to the genetic code (beginning at gene one), digraphs in the odd positions produce seed/parameters, while the even positions determine operations. Many of the ECRNG solutions performed outstandingly on all of our statistical tests (routinely passing all Diehard tests). However, this ECRNG system has proved difficult to implement in hardware.

With hardware limits in mind, we focused on generators that rely on hardware friendly computations (*i.e.* simple arithmetic and Boolean operators). Cellular Automata (CA)-based PRNG and Ant Colony Simulations (ACS)-RNG algorithms both fit this criteria. While CA are easily encoded in hardware, we have not been able to identify an evolvable coding scheme that passes any of our statistical tests. We have, however, designed and evolved several ACS-RNG. Our first method (ACS-1) used chromosomes containing three genes: the first gene encoding for individual ant movement ranges, integer range [1,3]; the second and third genes each comprise a single ten-bit binary encoding, with a totalistic decoding scheme allowing a range from 0 to 10—these genes represent the colony's food pickup and drop-off scaling factors, respectively.

While attempting to encode our ACS-1 in hardware, we discovered a simplified algorithm, ACS-2, which out performs ACS-1. An ACS-2 chromosome consists of a single gene, either length 20 (ACS-2a) or 32 (ACS-2b), whereby each locus represents the movement ranges for an individual ant. These ants have increased movement ranges, with six possible states for each ant. ACS-2 ants operate on a single cylindrical array, with movements limited to left/right choices (depending on the value of a single bit CA-QRNG controller). Food distribution in ACS-2 is governed by a simple rule: if an ant has food and there is at most one piece of food in its cell, then the ant drops the food. If an ant has no food and there is food in its cell, then the ants gathers the food. Polling the ants, registering a 1 if an ant has food, and a 0 if not, produces the random bit stream. Thus far, evolved ACS-2 RNG nominally pass 14 of 18 Diehard tests.

In summary, we have succeeded at evolving very high-quality RNG in software. Our off-line testing indicates that evolved generators of the ECRNG family routinely pass all tests in the Diehard battery. However, these are not implementable in hardware at this time. We have also demonstrated that it is possible to evolve ACS-RNG in hardware. Our method produces generators that pass many stringent statistical tests for randomness (failing only a few of the most stringent, theoretical, Diehard tests). These are superior to many known, non-evolvable generators (for contrast, the C rand function routinely fails all 18 Diehard tests). We will continue to develop our methods—with an eye towards optimizing the compactness of the hardware implementation, specifically, minimizing gate count and maximizing switching frequency.