Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

A PLUG-IN–BASED ARCHITECTURE FOR RANDOM NUMBER GENERATION IN SIMULATION SYSTEMS

Roland Ewald
Johannes Rössel
Jan Himmelspach
Adelinde M. Uhrmacher

University of Rostock
Albert-Einstein-Str. 21
Rostock, 18059, Germany

ABSTRACT

Simulations often depend heavily on random numbers, yet the impact of random number generators is recognized seldom. The generation of random numbers for simulations is not trivial, as the quality of each algorithm depends on the simulation scenario. Therefore, simulation environments for large-scale experimentation with safety-critical models require a reliable mechanism to cope with this aspect. We show how to address this problem by realizing a random number generation architecture for a general-purpose simulation system. It provides various random number generators (RNGs), probability distributions, and RNG tests. It is open to future additions, which allows the assessment of new generators in a simulation context and the re-validation of past simulation studies. We present a short example that illustrates why the features of such an architecture are essential for getting valid results.

1 INTRODUCTION

Random number generation is a well-studied area in general (Knuth 1981) and also in the simulation realm (e. g., (L'Ecuyer 1990)); and even though this problem is important enough to motivate complete chapters in modeling and simulation textbooks (Law and Kelton 2000), the actual influence of random number generators on simulation results is an often neglected issue in simulation studies. Many stochastic simulation studies are published without even naming the generation algorithm that was used, which may lead to unreproducible results. The reasons for this might be manifold, but one aspect seems to be the frequent re-creation of application-specific simulation environments from scratch. Many components have to be realized for such an environment — e. g., model editor, simulation algorithm, experiment editor, data storage, a. s. o. Their development process is quite time-consuming and often leads to single solutions per required feature, i. e., one simulation algorithm or random number generator only. Most modern programming languages also provide built-in RNGs, which are often used to save development time.

However, since the properties of a generator determine the quality of its output in relation to a *particular* application, simply using a single RNG may introduce additional bias to the simulation study and therefore lead to incorrect results and conclusions — this is common sense in the domain of RNGs (Matsumoto et al. 2007). Moreover, there is no silver-bullet algorithm that generates good random numbers for any kind of application (Hellekalek 1998). A plethora of random number generators have already been proposed, and a typical user has difficulties in selecting one that suits the problem at hand. This is not only due to the underlying mathematical principles, but also because the actual usage of random numbers within the model and the simulator might result in subtle bias that is extremely hard to predict (Grassberger 1993).

In our opinion, neither modelers nor simulation algorithm developers should have to deal with random number generators in detail. This is not their field of expertise. Instead, selecting a generator from a pool of existing implementations should be facilitated, as using alternative RNGs also improves the confidence in the selection (L'Ecuyer 1997).

Another possibility is to test several random number generators in a competitive setup, which could give some insights into classes of problems where certain RNGs are more robust than others – which could lead to a semi-automatic selection of suitable random number generators for a given setup.

In addition, such setups can facilitate teaching and training of RNG realization and usage: students may simply plug-in their generators and compare them with those already available in the system. If the system is a modeling and simulation framework with an RNG architecture, such comparisons can either be done in a more abstract (statistical tests) or a more practical manner (real simulations).

The following section gives a short overview on common deficiencies of random number generators and how these defects may distort simulation results. We then briefly introduce JAMES II and present a random number generation framework that touches several important aspects of modeling and simulation. Section four describes how the effects of random number generators can be tested and quantified, and the last section concludes the paper.

2 BACKGROUND

The relation between random number generators and the field of modeling and simulation is twofold. On the one side, research on random numbers often employs simulation techniques for testing (e. g., Matsumoto et al. 2007). On the other side, stochastic computer simulation is gaining more importance and is applied to ever more application domains. For example, several stochastic approaches from Computational Biology make massive use of RNGs (e. g., Gillespie 1977).

Two general aspects of RNGs are of particular importance in the simulation context: period and seed size. Seed size determines the maximum number of possible trajectories, since all (algorithmic) RNGs are deterministic functions that only generate pseudo-random numbers, the seed being their input parameter. This may lead to problems when the number of possible seeds is much smaller than the number of trajectories that *would* be possible (Marsaglia 2003), e. g., when testing safety-critical applications via simulation. The period of the RNG determines how many random numbers can be generated before the sequence repeats itself. This property is usually uncritical (as the periods of many RNGs are huge, cf. Table 1), but it still needs consideration when executing large stochastic simulations that run several days or weeks. It is also recommended in (Hellekalek 1998) that the amount of generated numbers should usually not exceed the square root of the generator's period, since the generation is equivalent to drawing without replacement (Hellekalek 1998), and hence introduces bias.

2.1 Random number generation

RNG algorithms have already been investigated in the context of simulation (L'Ecuyer 1990). From the bevy of existing approaches, we limit the discussion to the most basic and common family of algorithms, namely the linear congruential generators (LCGs). They serve as an introductory example for terminology and common problems, as they also form a basis for several similar and more advanced methods. An RNG is characterized by an *iteration function* (or transition function) that computes the next *state* of the RNG from the given one. Iterating the function results in a transition within the generator's *state space* and yields a new pseudo-random number (its new state). An LCG is defined by the iteration function

$$x_n = a \cdot x_{n-1} + b \pmod{c}$$

where a, b, and c are the parameters of the LCG, and x_0 is the seed. Other approaches, most notably lagged Fibonacci generators (LFGs), combine several former states x_i, $i < n$. This either requires a larger seed or an additional RNG to initialize the first states randomly. The initialization of more complex RNGs is one of the scenarios where LCGs are still widely used. A very popular approach that also requires prior initialization is the Mersenne Twister (Matsumoto and Nishimura 1998), which is known for its very large period (cf. Table 1), speed, and high-quality output. Additionally, RNGs define an *output function* that often simply normalizes the current state of the RNG to generate a number in $[0, 1)$. In case of the aforementioned LCG, the output function could calculate $\frac{x_n}{c}$ (L'Ecuyer 1997).

2.2 RNG properties and defects

(Hellekalek 1998) identifies the theoretical properties of RNGs as period length, structural properties, and correlations. The period length is related to the size of a generator's state space, but does not need to be equivalent. Additive lagged-Fibonacci generators (ALFGs), for example, have a toroidal state space. While the iteration function moves the state along one dimension, moving along the other dimension will yield new seeds (i. e., states) for RNG streams with another period of same length (Mascagni and Srinivasan 2004). Structural properties refer to intrinsic structures of the generated numbers. For example, d-tuples of multiplicative RNGs are known to lie in a limited number of $(d-1)$-dimensional hyperplanes (Marsaglia 1968). Finally, correlations subsume all interdependencies of generated subsequences, combinations of RNGs, or certain initialization schemes (Hellekalek 1998).

Initialization is particularly problematic for RNGs with a large state space. Usually, another RNG is initialized with the actual seed and then creates the initial state. This can lead to various correlations, or *defects*, especially when both generators have similar structural properties or even the same parameters (Matsumoto et al. 2007). A defect can be *transient*, i. e., only temporary, until a certain "warm–up" phase of the algorithm has been finished, or *persistent*.

In (Matsumoto et al. 2007), two related defects have been analyzed and identified in several algorithms: *affine dependence* and *difference collision*. Affine dependence allows to predict $x_n(s) \in \{0, 1, \ldots, M-1\}$, the n-th output of an RNG initialized with seed s, by $x_n(s) = a_n \cdot s + c_n \; mod \; M$, where a_n and c_n are parameters independent of s. All LCGs have a persistent affine dependence (Matsumoto et al. 2007). Other algorithms have nearly affine dependence, which permits a small prediction error $e_n(s)$. In a simulation, this defect may introduce a huge bias, e. g., if stochastic model entities are initialized with subsequent seeds (cf. Section 4.1). Difference collision refers to the similarity of differences between subsequent random numbers from RNGs initialized with different seeds. Such collisions may easily render a stochastic simulation invalid. For example, consider exploring various timed interactions of stochastic entities: with difference collisions, the time spans of certain actions may become heavily correlated, even though this was not intended by the modeler.

Unfortunately, many defects can only be discovered in practice, e. g., when highly stochastic models like the Ising model are investigated via Monte-Carlo simulation (Grassberger 1993). Although no RNG is able to provide uncorrelated random numbers for all kinds of simulation application (Hellekalek 1998), empirical tests can be used to complement theoretical findings and to discover the most severe defects in common setups.

2.3 RNG testing and simulations

In (Srinivasan et al. 2003), RNG tests are categorized as either *statistical* or *application-based*. Several statistical test suites have been proposed, e. g., NIST 800-22 (Rukhin et al. 2001), Diehard (Marsaglia 1995), or FIPS 140-1 (Federal Information Processing Standards and Technology 1982), and new approaches are subject of current research (e. g., Duggan et al. 2005). Statistical testing looks for certain correlations within an RNG's output. If a given RNG passes a test, the tested kind of correlation can be ruled out. On the other hand, statistical testing is *not* sufficient to ensure an RNG's suitability in a certain simulation scenario, as it is usually unknown which correlations are particularly harmful. Even the characterization of what defines a high-quality sequence of random numbers is up for debate in the RNG community (super-uniformity vs. LIL-uniformity, cf. Hellekalek 1998). In some sense, statistical tests merely express common simulation requirements and constitute *"prototypes of simulation problems"* (Hellekalek 1998, p. 494).

Therefore, it is regarded as mandatory to apply application-based testing to RNGs, i. e., to use an RNG for simulation applications and to analyze the outcome (Srinivasan et al. 2003). Conversely, no stochastic simulation result can be validated without using *different* RNGs, each having different characteristics and different defects (Hellekalek 1998). Due to the explorative character of many simulation studies, RNG defects may not be recognized otherwise. In (Grassberger 1993), for example, defects and long-range correlations in LFGs and other RNGs could only be identified because of theoretical *a priori* knowledge. Such knowledge is often not available for stochastic models.

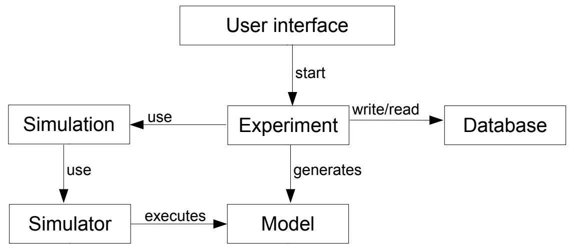


Figure 1: Framework controlled experiment execution

2.4 JAMES II

The simulation framework JAMES II is a very lean system consisting of a set of core classes. The core, which forms the basis of the scalable modeling and simulation framework JAMES II, is the central and most rarely changed part of the framework. The main parts of the core are: User interface, Data, Model, Simulator, Simulation, Experiment, and Registry. The `Experiment` package is central in the design of JAMES II (see Figure 1).

We used common software engineering techniques for the creation of the framework, e. g., the model-view-controller paradigm for decoupling its parts (Gamma et al. 1994). The strict differentiation between different concerns allows, e. g., to switch the simulation engine (even during runtime) and to exchange the data structures used for the executable models – an essential feature for a scalable framework. In addition, this adds the possibility to use JAMES II for reliable evaluations of new simulation algorithms. In combination with an XML-based model component plug-in, this flexibility enables the freedom of choice in regards to model data type, simulator code (algorithm as such; or sub-algorithms, e. g., event queues, random number generators), visualization, and runtime environment.

Whenever possible and useful, JAMES II tries to provide a default behavior for tasks. That's in particular true for the lower levels with a larger distance to the user.

2.4.1 Extension points

The "Plug'n simulate" approach (Himmelspach and Uhrmacher 2007) has been developed for supporting, on the one hand, a variety of solutions which may be provided by third parties, and on the other hand for enabling yet unforeseen types of plug-ins. Functionality not included in the core classes, especially modeling formalisms and simulation algorithms, can be added in form of plug-ins. The scalability of JAMES II relies on these extension points as well as on the availability of extensions for these. Extension points in the core are, e. g., different modeling formalisms & languages, and random number generators.

3 RANDOM NUMBERS IN JAMES II

Following its basic principles, JAMES II provides an extension point for random number generators, by which any RNG can be provided as a plug-in. Thus, stochastic models and simulation algorithms for JAMES II are "future-proof", i. e., their re-validation with more advanced RNGs later on is inherently supported. Such re-validation is required when new knowledge on RNG defects or correlations is available, or if better methods have been developed. Those can be easily integrated into the framework, and all experiments could then be repeated. This feature of JAMES II will also help to provide a large suite of application-based tests for RNG evaluation without much additional effort.

3.1 Requirements

An RNG mechanism for a general-purpose simulation system like JAMES II needs to fulfill several additional requirements. First of all, RNGs usually generate pseudo-random numbers that are uniformly distributed. To better support stochastic modeling, it is necessary that additional probability distributions can be defined. These should convert uniformly distributed numbers to the target distribution, so that arbitrary combinations of RNGs and probability distributions are possible.

As already pointed out, no stochastic simulation result can be considered valid if only one RNG has been used. RNG selection and setup (seed, initialization scheme, parameters) have to be part of the explicit experiment description — this makes experiments reproducible, while allowing the user to select different RNGs for testing. One should therefore regard the RNG setup as part of the experimental frame — the setup is inseparably connected to the simulation results. Such a repeatable experimental setup also allows to test proposed enhancements of existing methods (e. g., L'Ecuyer and Touzin 2000), i. e., their effect on simulation results and simulation speed. It is also important that some classic RNGs are available, which allows re-validating past results.

JAMES II supports parallel and distributed simulation, which implies that multiple *streams* of random numbers need to be generated on distinct hosts. There are two basic approaches to parallel random number generation (PRNG): *cycle division* and *parameterization* (Srinivasan et al. 2003). Cycle division splits the cycle of subsequently generated numbers (which has the size of the RNG's period), and then initializes all required RNGs to generate different parts of that cycle. While this approach is rather straightforward to implement, it does not scale to large amounts of uncorrelated RNG streams. Parameterization aims at generating uncorrelated RNG streams of full period size by adjusting the parameters of the RNG's iteration function. It is often harder to achieve and requires thorough mathematical analysis and exhaustive testing. For example, the ALFGs mentioned in Section 2.2 can be used for parameterization (Mascagni and Srinivasan 2004). A flexible RNG architecture for simulation needs to support both PRNG schemes.

Sometimes it is also required to sample a sublist out of a given one. Sampling can be done with or without replacement, i. e., resulting in lists that contain only unique entries or not. Subset sampling is an old problem (e. g., (Bentley 1999)) and efficient solutions are well known. They should be provided by a modeling and simulation framework, so that validated implementations can be reused whenever necessary.

3.2 Architecture

The basic RNG architecture of JAMES II consists of three extension points: *RNG generators*, *RNGs*, and *Probability distributions* (see Figure 2). They provide the main components whose combinations fulfill the requirements from Section 3.1.

RNG generators abstract from different parallel RNG approaches in providing an interface that allows to create as many RNGs as necessary — this is the well-known factory pattern (Gamma et al. 1994). Since RNG instantiation is now hidden from the user, both cycle division and parameterization can be implemented without affecting other code. The parameters of an RNG generator have to determine the exact order and parameterization of generated RNGs, so that repeatability is ensured.

Random number generators are required to implement the `IRandom` interface, which provides the user with methods to read and write the RNG's seed. The RNG seed is of type `java.io.Serializable`, which allows to store it to arbitrary data sinks. Again, this is a precondition to repeatability, and also to distributed simulation, where RNG seeds and parameters need to be sent over the network. `IRandom`'s sole method to generate random numbers returns uniformly distributed values in $[0,1)$.

Uniform random numbers in $[0,1)$ are of limited use in real-world models or simulations. Most often, they need to be distributed according to a certain probability distribution. This has already been considered several times and there are solutions for this problem in several simulation software packages (e. g., SSJ (L'Ecuyer et al. 2002)). All probability distributions in JAMES II are subclasses of `AbstractDistribution`, which is initialized with an instance of `IRandom`, so that any probability distribution can be combined with any RNG. Parameterization of distributions is handled by their corresponding factories, which are part of their plug-in definition (see Section 2.4). Finally, the class `RandomSampler` supports developers by implementing sampling functions based on the `IRandom` interface. JAMES II provides sampling support because it

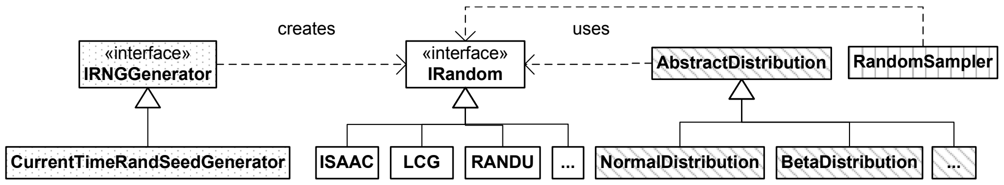


Figure 2: Plug-in based RNG architecture for JAMES II. Each background pattern denotes a single extension point in JAMES II (except for the random sampler, which is an auxiliary class).

is a feature required by several modeling approaches (e. g., in micro simulations).

All in all, this simple RNG architecture has three advantages: Firstly, RNG creation is encapsulated in RNG generators and therefore completely separated from application code. This allows to implement and evaluate different approaches to create parallel RNGs. Secondly, the central `IRandom` interface allows to combine all available plug-ins and eases the extension of this framework for other mechanisms that rely on random numbers. Thirdly, the plug-in based approach of JAMES II allows to reuse many existing RNG libraries by wrapping them into new plugins — the wheel should not be invented twice. Any new RNG generator, RNG, or probability distribution becomes instantly available for all experimental setups.

3.3 Implementation

JAMES II includes two "built-in" RNGs: a wrapper for `java.util.Random`, which is an LCG, and an implementation of the Mersenne Twister (Matsumoto and Nishimura 1998). To prove our architecture, we also implemented the ISAAC (Jenkins 1996) generator, the "Mother of all RNGs" (Marsaglia 1995) (a multiply-with-carry generator), all of which are available as plug-ins. The RANDU generator, a classic generator that is typically no longer in use, has also been implemented. This is for two reasons: firstly, for assessing the effect that such a generator, known for its highly correlated output, has on our current applications (see Section 4.1). Secondly, we aim at reproducing results of former simulation studies based on RANDU. Finally, a parameterizable LCG is available for evaluation. Our modular approach makes it trivial to add further methods.

Several probability distributions are already provided by JAMES II (see Table 2 for a selection). The ones shown here are well-known in the field of event queues (e. g., Rönngren and Ayani 1997), where they are typically used for testing and evaluation.

In Figure 4, it can be seen that the results of probability distributions may be quite dependent from the used RNG. In all three setups, the same seed has been used to generate 100.000 random numbers with a camel distribution of $(2,0.2,0.5)$. The results of the Java default RNG (1)

Table 1: RNGs realized for JAMES II.

Name		Period
a	Java Random	2^{48}
b	Mersenne Twister	$2^{19937}-1$
c	ISAAC	at least 2^{40}, on avg. 2^{8295}
d	Mother of all RNGs	$\approx 2^{160}$
e	RANDU	2^{31}
f	LCG	2^{48}
g	Java SecureRandom	$2^{?}$

Table 2: Selection of probability distributions realized in JAMES II (cf. Figure 3).

Distribution		Expression to compute random values
1.	Exponential	$-ln(rand)$
2.	Uniform 0.0–2.0	$2*rand$
3.	Biased 0.9–1.1	$0.9+0.2*rand$
4.	Bimodal	$0.95238 * rand +$ if $rand < 0.1$ then 9.5238 else 0
5.	Triangular	$1.5*rand^{0.5}$
6.	Neg Triangular	$1000*(1-\sqrt{rand})$
7.	Camel $(2,0.8,0.2)$	see (Rönngren and Ayani 1997)
8.	Camel $(2,0.999,0.001)$	see (Rönngren and Ayani 1997)

and the Mersenne Twister (2) are as expected and approximate the desired probability density function. In contrast, RANDU's results (3) are lacking any values in $\approx [0.6,0.8]$. This again shows that an easy-to-use framework for plugging in *different combinations* of RNGs and probability distributions is essential for thorough experimental analysis with stochastic simulation and algorithms/datastructures (e. g., event queues).

3.4 Testing random number generators

Although using "good" generators is no sufficient prerequisite for valid stochastic simulation results (see Section 2.3), all RNGs should be tested carefully with statistical means.

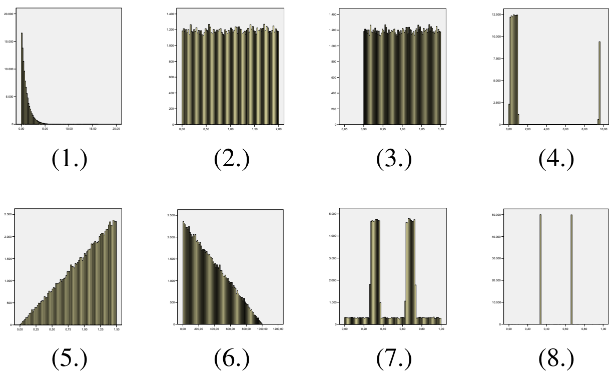


Figure 3: Pictures of the random distributions from Table 2. Generated by using the corresponding plug-ins in JAMES II and the default Java RNG.

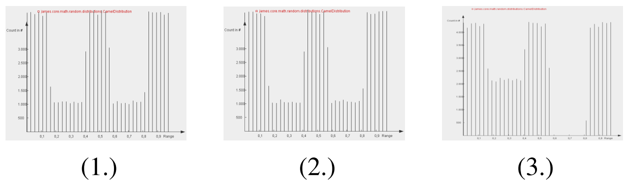


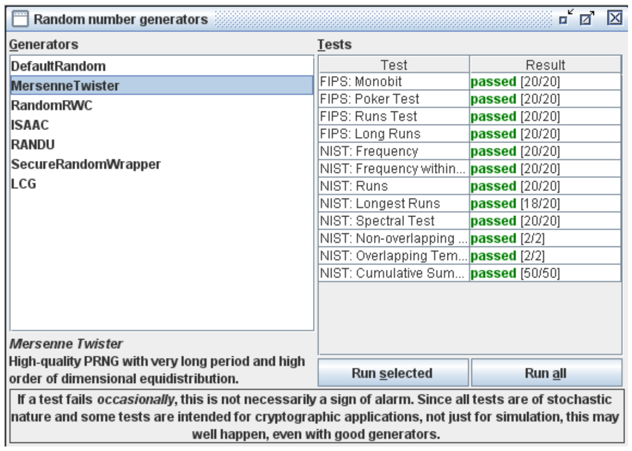
Figure 4: Camel distribution fed by three RNGs. The third picture (from RANDU) shows strong defects. The pictures have been generated using a simple distribution test dialog integrated into the JAMES II GUI.

This will at least hint at their performance in realistic scenarios and may reveal serious defects. To integrate easy empirical RNG testing in JAMES II, we created an interactive testing framework with which any user can test available RNGs via a GUI (see Figure 5). This should increase the confidence of the user in the selected algorithm.

The testing framework currently supports all tests described in Table 3. However, as the testing framework is based on the plug-in schema as well, anyone can extend this list easily, e. g., by integrating existing test suites (such as TestU01 by L'Ecuyer and Simard 2007). Even tests which are no longer considered to be trustworthy contribute to the system, since JAMES II is used for teaching and training as well.

3.5 Integration with JAMES II

An important design aspect of this plug-in based architecture is its seamless integration with JAMES II. This does not only refer to technical compatibility, which is ensured by the plug-in schema, but also to the conceptual level. As already mentioned in Section 3.1, we regard the RNG setup as a crucial aspect of the experimental frame. Therefore,



Random number generators

Generators
- DefaultRandom
- MersenneTwister
- RandomRWC
- ISAAC
- RANDU
- SecureRandomWrapper
- LCG

Tests

Test	Result
FIPS: Monobit	passed [20/20]
FIPS: Poker Test	passed [20/20]
FIPS: Runs Test	passed [20/20]
FIPS: Long Runs	passed [20/20]
NIST: Frequency	passed [20/20]
NIST: Frequency within...	passed [20/20]
NIST: Runs	passed [20/20]
NIST: Longest Runs	passed [18/20]
NIST: Spectral Test	passed [20/20]
NIST: Non-overlapping ...	passed [2/2]
NIST: Overlapping Tem...	passed [2/2]
NIST: Cumulative Sum...	passed [50/50]

Mersenne Twister
High-quality PRNG with very long period and high order of dimensional equidistribution.

Run selected | Run all

If a test fails *occasionally*, this is not necessarily a sign of alarm. Since all tests are of stochastic nature and some tests are intended for cryptographic applications, not just for simulation, this may well happen, even with good generators.

Figure 5: Screenshot of the RNG testing framework integrated into JAMES II.

all RNG generator parameters need to be stored in the experiment definition, and the seeds and parameters of RNGs used in a single simulation run need to be handled as meta information by our data storage system for result saving. Otherwise, experiments with JAMES II would not be repeatable. Another challenge is to save valid snapshots of distributed simulation runs, which can be used to resume the simulation after a hardware breakdown. It requires to store *all* RNGs as well, including their current state, seed, parameters, and additional information to attach each RNG to the same probability distribution when resuming. Some of these integration issues are still open and subject of future research.

4 EXPERIMENTS

This section subsumes some empirical and application tests that have been conducted with the currently implemented RNGs. At least the empirical tests enumerated in Table 3 should be used to check newly developed RNGs before they are used for experimentation. The results of the test are given in Table 4. The LCG (f) has been parametrized with a bad initial multiplier value of 2 to illustrate the usage of the already realized tests. Users and students can thus experiment with parameters on the fly and may get a feeling for better and worse combinations — especially if this is accompanied with some theoretical background. Further tests may advance this idea to a new level if some of the other generators start failing as well.

4.1 An application example

To illustrate the importance of a reliable RNG mechanism in day-to-day simulation, we now present a brief example along the argumentation line of Matsumoto et al. (2007).

Table 3: Random number generator tests.

	Name	Short description
	FIPS 140-1	
1	Monobit Test	Frequency of 1s and 0s in bit stream.
2	Poker Test	Frequency of four-bit values.
3	Runs Test	Runs of identical consecutive bits.
4	Long Runs Test	Checks that abnormally long runs do not appear.
	NIST SP 800-22	
5	Frequency Test	Proportion of 0s and 1s, akin to the FIPS Monobit test.
6	Freq. Test in Blocks	Proportion of 1s in M-bit blocks.
7	Runs Test	Number of runs of various lengths.
7	Longest Run of 1s in Block	Longest run of 1s in M-bit blocks.
8	Spectral Test	Periodic features near each other.
9	Non-Overlapping Template Matching	Occurrences of pre-specified aperiodic patterns.
10	Overlapping Template Matching	Like above, but counts overlapping matches.
11	Cumulative Sums	Interprets the bits as a random walk and looks for abnormal deviations.

We created a simple coupled PDEVS-model (Chow and Zeigler 1994) with a variable number of n atomic models that conduct iterated coin-tossing, i. e., they randomly decide on executing action a or b at each time $t = 0, 1, \ldots, 100$. The atomic models were initialized with single RNGs and seeds $s = 0, \ldots, n-1$. Note that subsequent seeds are *not* equivalent with subsequent states of an RNG, as these are determined by its iteration function. Figures 6 and 7 show results for $n = 10$ and $n = 1000$ as the difference $|m_a - m_b|$ of models choosing action a (m_a) or b (m_b).

A value of 10 in case $n = 10$ has a probability of $2 * ((0.5)^{10}) \approx 0.2\,\%$, as this is only possible if *all* models choose either a or b. This means, the higher a data point in

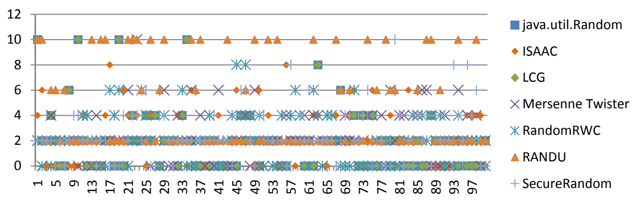


Figure 6: Difference between $n = 10$ models choosing a or b per point in time.

Table 4: Test results of using the eleven tests from Table 3 on the seven random number generators from Table 1.

	a	b	c	d	e	f	g
1	✓	✓	✓	✓	✓		✓
2	✓	✓	✓	✓	✓		✓
3	✓	✓	✓	✓	✓		✓
4	✓	✓	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	✓		✓
6	✓	✓	✓	✓	✓		✓
7	✓	✓	✓	✓	✓		✓
8	✓	✓	✓	✓	✓		✓
9	✓	✓	✓	✓	✓	✓	✓
10	✓	✓	✓	✓	✓	✓	✓
11	✓	✓	✓	✓	✓		✓

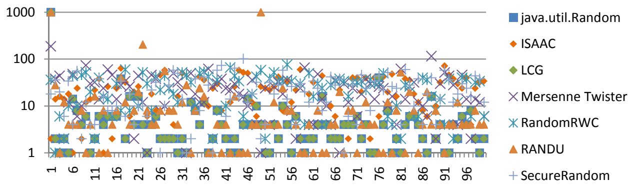


Figure 7: Difference between $n = 1000$ models choosing a or b per point in time.

the plot, the lower its probability. As can be seen in both plots, all LCGs (RANDU, Java's Random, and our LCG implementation) exhibit correlations for time 0. In case $n = 1000$ (Figure 7), the probability of such an outcome would be $\approx 2 \cdot 10^{-299}\,\%$. Simulation with RANDU is also clearly correlated at times 22 and 49.

This behavior is anything but new (see (Matsumoto et al. 2007) for a detailed discussion), but merely illustrates again that choosing the wrong RNG may have a huge impact on simulation results. Another example of large bias could be randomly placed particles in a 3D-simulation with a poorly configured LCG, since three-tuples of LCGs form two-dimensional planes in three-dimensional space (cf. spectral test, Knuth 1981).

5 RELATED WORK

Due to the overwhelming number of different solutions in the fields of "random number generation" and "simulation software", the following overview lists only a few packages strongly related to what we have presented.

SSJ (L'Ecuyer et al. 2002, L'Ecuyer and Buist 2005) is based on several years of research on random number generators and tests. It is a Java library which can be reused from different Java applications. SSJ is not as easy to extend as JAMES II but already provides diverse random number generators, distributions, statistical functions, and event queue implementations, which makes it to some degree comparable with JAMES II. Similar to JAMES II, SSJ can also be used for discrete, continuous, or hybrid simulations, but in contrast to JAMES II reuse is not based on a plug-in-based schema.

Colt (CERN) is an “Open Source Library for High Performance Scientific and Technical Computing in Java”. It provides diverse classes generally applicable in modeling and simulation applications (among them random number generation, distributions, and sampling), but is not solely dedicated to those.

Diehard (Marsaglia 1995), TestU01 (L’Ecuyer and Simard 2007), and many other test suites support empirical RNG testing. Most often these are standalone packages not strictly associated to certain random number implementations (e. g., they are fed by CSV files). JAMES II allows the integration of any test suite or RNG package by exploiting the plug-in schema once more. This enables every user to apply all available tests to all available RNGs without having to leave the system. Most of the existing packages could be added to JAMES II via a wrapper schema, which enables anyone to compare results over a broad range of RNGs.

6 CONCLUSIONS AND OUTLOOK

We presented a plug-in based architecture for the integration of random numbers in a general modeling and simulation framework and derived a generic RNG architecture from the requirements motivated and analyzed in Section 2 and 3.1. By exploiting the plug’n simulate concept, random number generation in such a framework becomes an exchangeable and well documented service. This forms a base of “trust” for random numbers and simulation results likewise. The importance of such an approach is demonstrated by a brief example in Section 4.1. The proposed architecture has been realized and evaluated in JAMES II. It currently provides support for different RNG generators, RNGs, probability distributions, random sampling algorithms, and test methods. The RNG generators allow to implement different approaches for parallel RNG, such as *cycle division* and *parameterization* for distributed simulation. To the best of our knowledge, such an approach — based on a plug-in based schema and fully integrated into a general modeling and simulation framework — has not been developed yet.

As known from the literature (Grassberger 1993), the quality of RNGs has a large effect on the quality of the simulation results — a framework as the one proposed in this paper helps to get aware of the problem and can be easily used to check the influence of different generators by exchanging them. It should therefore help to achieve simulation results of better quality. Like other experimental disciplines, simulation strives to reduce the number of potential error sources. Therefore, a modeling and simulation framework should provide as many pre-factored and well validated methods as possible.

Furthermore, a flexible RNG architecture can help pointing out the importance of random number generation in the simulation context and may put forward questions concerning model validity and artifacts stemming from the experimental techniques like stochastic simulation. This aspect is closely related to teaching and training RNG implementation and usage. Although the framework in principle now provides all prerequisites for practical teaching and training (a GUI, clean interfaces, flexibility), its effectiveness for increasing understanding by hands-on experience has still to be investigated.

Finally, RNG execution speed is always an issue, especially when dealing with large-scale stochastic simulations. Different simulation studies may require different random number generators — there might be simulations which require “high-quality” random numbers and others for which slightly correlated numbers (e. g., from a well parameterized LCG) are sufficient and lead to recognizable speedup. We also plan to add existing RNG packages (such as SPRNG, see Mascagni and Srinivasan 2000) and test frameworks to JAMES II, so that we can quantify the real impact of the algorithm’s theoretical differences on simulation results. Due to the number of modeling formalisms and the corresponding simulation algorithms supported by JAMES II, these experiments can be done for many different settings and models.

REFERENCES

Bentley, J. 1999. *Programming pearls*. 2. ed. Addison-Wesley Professional.

CERN. Colt. `http://dsd.lbl.gov/~hoschek/colt/`. Accessed 08/25/08.

Chow, A. C., and B. P. Zeigler. 1994. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proc. of the 1994 Winter Simulation Conference*, ed. J. D. Tew and S. Manivannan, 716–722. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Duggan, M. J., J. H. Drew, and L. M. Leemis. 2005. A test of randomness based on the distance between consecutive random number pairs. In *Proc. of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 741–748: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Federal Information Processing Standards, N. I. o. S., and Technology. 1982, April. Security requirements for cryptographic modules (FIPS 140-1). `http://www.itl.nist.gov/fipspubs/fip140-1.htm`.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, USA.

Gillespie, D. T. 1977. Exact Stochastic Simulation of Coupled Chemical Reactions. *J. Phys. Chem.* 81 (25).

Grassberger, P. 1993. On correlations in "good" random number generators. *Phys. Lett. A* 181 (1): 43–46.

Hellekalek, P. 1998. Good random number generators are (not so) easy to find. *Math. Comput. Simul.* 46 (5-6): 485–505.

Himmelspach, J., and A. M. Uhrmacher. 2007, March. Plug'n simulate. In *Proceedings of the ANSS*, 137–143: IEEE Computer Society.

Jenkins, B. 1996. ISAAC, a fast cryptographic random number generator. http://www.burtleburtle.net/bob/rand/isaacafa.html. Accessed 08/25/08.

Knuth, D. E. 1981, November. *Art of computer programming, volume 2: Seminumerical algorithms (2nd edition)*. Addison-Wesley Professional.

Law, A., and D. W. Kelton. 2000. *Simulation modeling and analysis*. 3. ed. Mc GrawHill.

L'Ecuyer, P. 1990. Random numbers for simulation. *Commun. ACM* 33 (10): 85–97.

L'Ecuyer, P. 1997. Uniform random number generators: a review. In *Proc. of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 127–134. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P., and E. Buist. 2005. Simulation in Java with SSJ. In *Proc. of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 611–620: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P., L. Meliani, and J. Vaucher. 2002. SSJ: a framework for stochastic simulation in Java. In *Proc. of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, Volume 1, 234–242: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P., and R. Simard. 2007, August. Testu01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.* 33 (4).

L'Ecuyer, P., and R. Touzin. 2000. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In *Proc. of the 2000 Winter Simulation Conference*, ed. J. A. Joines and R. R. Barton, 683–689: Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Marsaglia, G. 1968, September. Random numbers fall mainly in the planes. *Proc. of the National Academy of Sciences of the USA* 61 (1): 25–28.

Marsaglia, G. 1995. The Marsaglia random number CDROM including the Diehard battery of tests of randomness. http://www.stat.fsu.edu/pub/diehard/. Accessed 08/25/08.

Marsaglia, G. 2003. Seeds for random number generators. *Commun. ACM* 46 (5): 90–93.

Mascagni, M., and A. Srinivasan. 2000. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.* 26 (3): 436–461.

Mascagni, M., and A. Srinivasan. 2004, July. Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing* 30 (7): 899–916.

Matsumoto, M., and T. Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul.* 8 (1): 3–30.

Matsumoto, M., I. Wada, A. Kuramoto, and H. Ashihara. 2007, September. Common defects in initialization of pseudorandom number generators. *ACM Trans. Model. Comput. Simul.* 17 (4).

Rönngren, R., and R. Ayani. 1997. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.* 7 (2): 157–209.

Rukhin, A., J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. 2001, May. NIST Special Publication 800-22, a statistical test suite for random and pseudorandom number generators. http://csrc.nist.gov/groups/ST/toolkit/documents/rng/SP800-22b.pdf.

Srinivasan, A., M. Mascagni, and D. Ceperley. 2003, January. Testing parallel random number generators. *Parallel Computing* 29 (1): 69–94.

AUTHOR BIOGRAPHIES

ROLAND EWALD is a PhD candidate employed in the Modeling and Simulation Group of Prof. Uhrmacher. His research interest is the automatic configuration of simulation software.

JOHANNES RÖSSEL is a computer science student pursuing his MSc at the University of Rostock.

JAN HIMMELSPACH is employed as a (post-doc) researcher in the Modeling and Simulation Group of Prof. Uhrmacher. His research interest is simulation software engineering and efficient algorithms for simulation.

ADELINDE M. UHRMACHER is head of the Modeling and Simulation Group at the University of Rostock. Her research interests are in modeling and simulation methodologies, and their applications.