A Unified Simulation Framework for Spatial Stochastic Models

J. Mayer [a,b,*], V. Schmidt [b], F. Schweiggert [a]

[a] *University of Ulm, Department of Applied Information Processing, D-89069 Ulm, Germany*

[b] *University of Ulm, Department of Stochastics, D-89069 Ulm, Germany*

Abstract

For spatial stochastic models, a lot of programs exist which deal with the simulation of specific models. But, combining them is not that easy and usually requires greater effort. This paper presents an object-oriented framework, i. e. a set of collaborating abstract and concrete classes, dealing with the simulation of such models. The selected fundamental models are only illustrating examples for the general concept. From the Java implementation of this framework, two code examples are shown, which could also be implemented similarly in any other object-oriented programming language. All interfaces and a lot of concrete classes can be implemented dimension-independent. The design and implementation problems arising in the context of static and dynamic plus sampling are specifically discussed.

Key words: Simulation, Spatial stochastic models, Java, Stochastic geometry

1 Introduction

Spatial stochastic-geometric models are random geometric structures, such as random configurations of points, overlapping (random) geometric objects at random locations, and random mosaics among others. They can be used to model global patterns and have proven useful in various fields, such as communication networks [1], materials science [15], medicine [9,10], physics [12], and astrophysics [13].

* Corresponding author. Tel.: +49-731-50-23573; fax: +49-731-50-23975.
Email address: `jmayer@mathematik.uni-ulm.de` (J. Mayer).

The combination of the existing simulation programs (cf. e. g. [15,16]) is not an easy task if possible at all. Furthermore, those implementations depend on the dimension of the simulated model. So, to simulate the model (for which a simulation program is given) in a higher dimension, at least changes are required, or a new program has to be developed.

The framework for the simulation of spatial stochastic-geometric models presented in this paper gives a possible solution to these problems and provides an unified approach. There are several orthogonal models in the framework which can be arbitrarily combined. This leads to combinatorial complexity of possible applications. All models are designed as interfaces, which are dimension-independent. Therefore, the framework is basically independent of the dimension. Only for a part of the concrete classes it would be much more complicated to give a dimension-independent implementation. So, for such classes a separate implementation has to be provided for each dimension. The framework is implemented in Java. It could, however, be implemented similarily in any object-oriented programming language. Two code examples of concrete models are presented, demonstrating the advantages of the framework design. All parts of the framework are explained using UML class diagrams [20].

The paper is organized as follows. Section 2 contains a brief description of fundamental spatial stochastic-geometric models considered as illustrating examples in the presented framework. Related work and common problems occuring during the combination of existing simulations are addressed in Section 3. The framework is described in Section 4, where sample implementations for two models are shown, followed by the summary and conclusions in Section 5.

2 Spatial Stochastic-Geometric Models

In this section, several fundamental spatial stochastic-geometric models are described, which are considered as illustrating examples in the present paper. The description is only as detailed as necessary for the design and non-formal — besides the little paragraph on basic morphological operations. The main focus is on the simulation of these models, but also not more precisely than required for the framework. The description is for the two-dimensional case, but can be extended to higher dimensions. An overview of spatial stochastic-geometric models is given, for example, in [19].

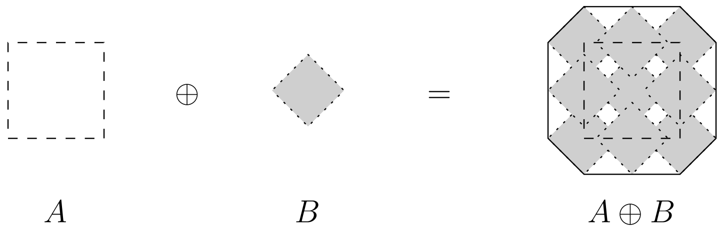


Fig. 1. Illustration of the Minkowski addition

2.1 Basic Morphological Operations

For simulation without edge effects, some basic operations from the field of mathematical morphology are necessary, which are described here shortly according to [15,18]. The *Minkowski addition* $A \oplus B$ of two sets A and B is the pointwise addition of the elements of both sets,

$$A \oplus B = \{x + y : x \in A, y \in B\}.$$

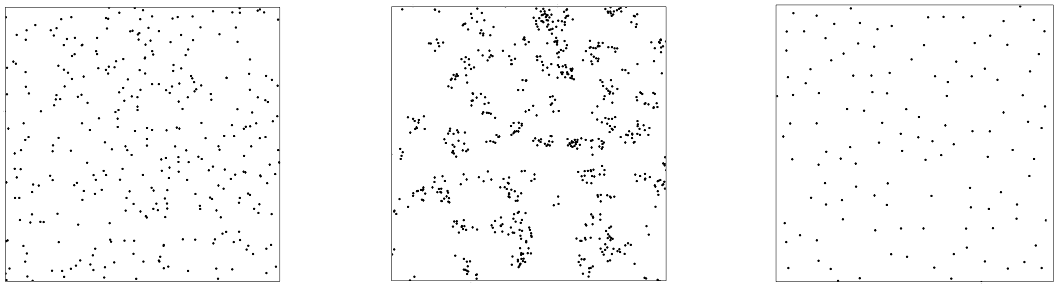
It holds $A \oplus B = \bigcup_{x \in A} B_x$, where B_x is the set B translated by x. With the latter equation, the example shown in Figure 1 — where only a few translations of B are depicted and the resulting set is outlined by the solid line — can be understood, since the result of $A \oplus B$ is the union of all translations of B into all points of A. With $\check{B}$ we denote the reflection of the set B at the origin, i. e. $\check{B} = \{-x : x \in B\}$. The *dilation* $\delta_B(A)$ of the set A with the so-called *structuring element* B is the Minkowski addition of A and the reflected structuring element $\check{B}$, i. e. $\delta_B(A) = A \oplus \check{B}$. For symmetric structuring elements, like B being a disc with center at the origin, dilation and Minkowski addition coincide.

2.2 Fundamental Spatial Stochastic-Geometric Models

2.2.1 Point Processes

A *point process* is a random variable whose realizations are (locally finite) series of points $\{x_1, x_2, \ldots\}$. Figure 2 shows typical realizations for various examples of point processes.

One well-known example is the *homogeneous Poisson process* (cf. Fig. 2(a)). The points of this process within a given sampling window are independently uniformly distributed, provided that the number of points within the sampling window is given. (The number of points within the sampling window is Poisson distributed and the parameter of this distribution is proportional to the area of the window.) Homogeneity means that the distribution of this point process is invariant under arbitrary translations of the origin. In the following,



(a) (b) (c)

Fig. 2. Samples of point processes

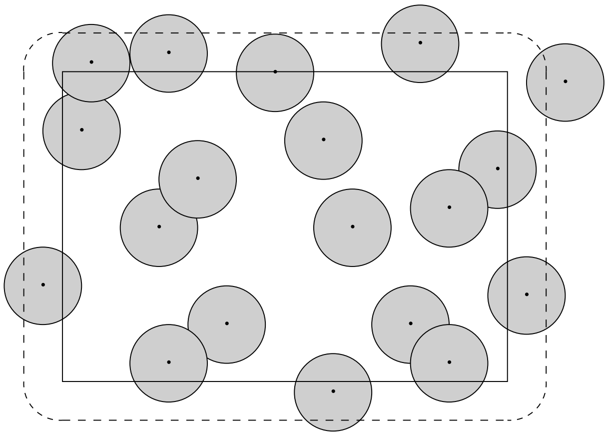
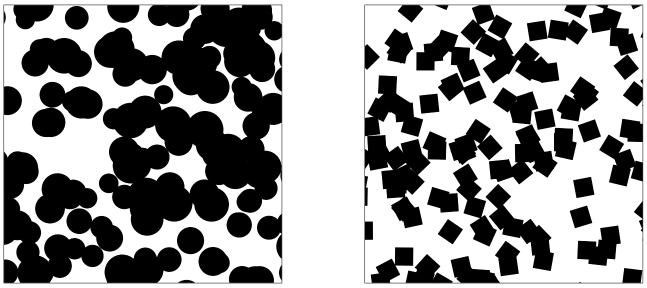


Fig. 3. Illustration of plus sampling

the considered Poisson processes are always assumed to be homogeneous. One possibility to simulate a homogeneous Poisson process is via its "conditional uniformity" property mentioned above. Furthermore, a *radial simulation* algorithm which produces the points with increasing distance from the origin is presented in [17]. This method is needed when the size of the sampling window is not known in advance — e. g., depends on the generated points.

A *Matérn cluster process* (cf. Fig. 2(b)) is generated by two Poisson processes. First the so-called parent process is simulated. For each point of the resulting pattern, the second process — the child process — is simulated within a disc around this point. The radius of the disc is called *cluster radius*. The union of all child points gives a realization of this process. In order to guarantee that the simulation is free of edge effects, the parent process must be simulated in an enlarged window, the sampling window dilated with the disc with the cluster radius and centered at the origin. The simulation within the enlarged window is called *plus sampling*. It is said to be static in this case, since the enlarged sampling window can be constructed before the simulation of the parent point process starts. In Figure 3, no parent points outside of the enlarged sampling window surrounded by the dashed line can have any influence on the result within the sampling window bounded by the solid line, where the radius of the discs is equal to the cluster radius.



(a) (b)

Fig. 4. Samples of the Boolean model

Another example is the *Matérn hard-core process* (cf. Fig. 2(c)). First a Poisson process is simulated. Then, a random number uniformly distributed within $(0,1)$ is assigned (independently) to each point. These assigned numbers are the marks of the points. Thereafter, all those points are removed which have at least one neighbor, i. e. a point within a given disc around it, with a smaller mark. The radius of the disc is here called *hard-core radius*. Since points out of the sampling window can force the removal of points within the window (cf. Fig. 3), plus sampling has to be applied here, too. The enlarged sampling window can be constructed as in the case of the cluster process.

It is obvious that cluster and hard-core processes can be generated not only by Poisson processes, but by arbitrary point processes.

2.2.2 The Boolean Model

Based on point processes, interpreted as random configurations of *germs*, further classes of spatial stochastic models can be defined. One example is the *Boolean model* (cf. Fig. 4) which is based on a Poisson process whose points are marked by random closed sets, the so-called *grains*. Into each point a realization of the *primary grain* is translated. Examples for primary grains are discs with uniformly distributed — within a certain finite range — radius (cf. Fig. 4a) and squares with random rotation (cf. Fig. 4b).

Grains on points outside of the sampling window can be partially within the window. Therefore, to avoid edge effects, plus sampling has to be applied. A dilation of the sampling window with the union of all possible realizations of the primary grain yields the enlarged window. In case of a disc with uniformly distributed radius, the union is the disc with the maximal radius, and in the case of a square with random rotation, the union is the disc with diameter equal to the diagonal length of the square.

The general model which is not restricted to the Poisson process is called *germ-grain model*.

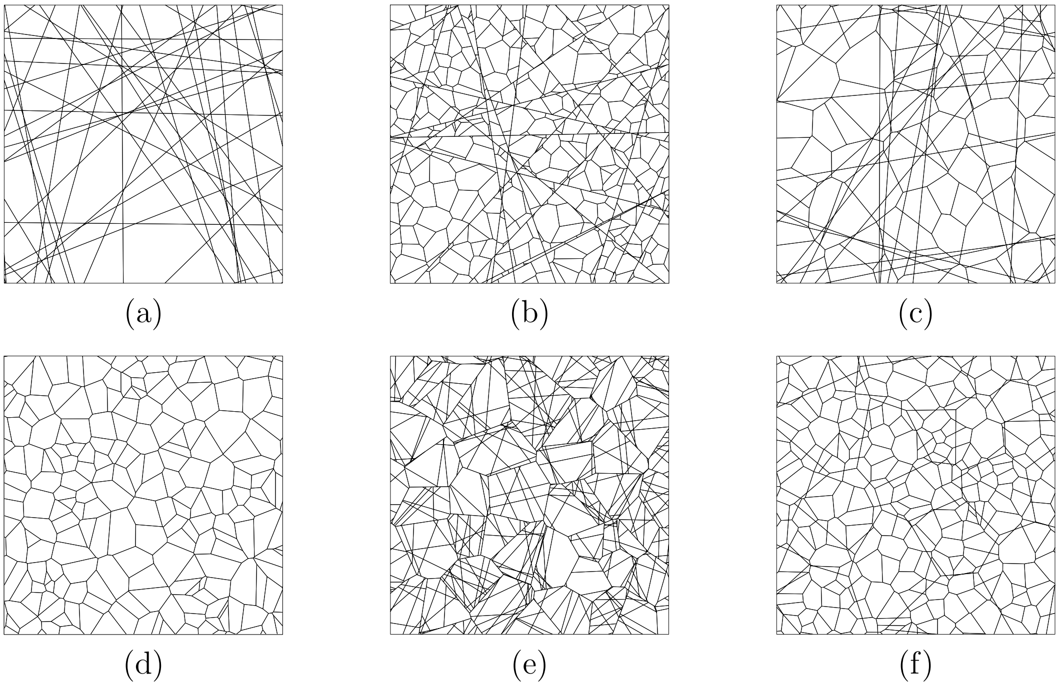


Fig. 5. Samples of tessellations

2.2.3 Random Tessellations

A *tessellation* subdivides the whole space into non-overlapping *cells*. The term mosaic is probably better known. Examples are shown in Figure 5. The *cells* of a tessellation are convex polygons.

A *Poisson line tessellation* (cf. Fig. 5(a)) is generated by lines with random orientation. The number of lines whose distance to the origin is within a certain range is Poisson distributed. The simulation of the line process is similar to the radial simulation of (two-dimensional) Poisson processes, since a line process is based on a one-dimensional Poisson process. Afterwards, the half-planes induced by the lines must be intersected to construct the cells. An arbitrary distribution for the orientation may be chosen in order to generalize this model.

Another type of random tessellations is the *Voronoi tessellation* based on a point process. If a Poisson process is chosen, it is called *Poisson-Voronoi tessellation* (cf. Fig. 5(d)). The Poisson process can be simulated on the sampling window, followed by the construction of the Voronoi tessellation. This approach, however, is not free of edge effects. Furthermore, static plus sampling cannot be applied, since the cells can be arbitrarily large. Radial simulation of the Poisson process, however, enables dynamic plus sampling and thus solves this problem. After each newly generated point the tessellation is adapted accordingly and all cells which lie within the sampling window are tested (with an adaption of the method described in [5]) whether they may change through further points (whose distance from the origin must be greater than that of all previous points, due to the radial simulation).

Iteration, i. e., embedding a *component tessellation* within the cells of an *initial tessellation*, yields further models [7,8]. *Independent iteration* or *nesting*, i. e. embedding an independent realization of the component tessellation within each cell of the initial tessellation, is shown in Figures 5(b) and 5(e). Examples for *superposition* of tessellations or dependent iteration are depicted in Figures 5(c) and 5(f). In all these examples, the initial and component tessellations are Poisson line or Poisson-Voronoi tessellations.

Characteristics, such as the number of vertices, area, and perimeter, of the cells of a tessellation are of particular interest. Via counting within a given window the number of all cells and the number of all cells with a given property, the probability that a randomly chosen cell has this property can be estimated. As the window approaches the whole space, the quotient approaches the exact (deterministic) value. A random polygon such that the probability for all cell properties is equal to the deterministic limit of quotients is called the *typical cell* of the tessellation. There are local simulation algorithms for the typical cell of the Poisson-Voronoi and the Poisson line tessellation. This means that in these cases the tessellation has not to be simulated on a huge window to estimate the probability of a typical cell property, but samples of the typical cell can be generated and thus used for the estimation; cf. [7].

3 Motivation

3.1 Some Related Work

The Image Analysis Simulation Statistics (IASS) C library version 0.1 developed at the ITWM Karlsruhe permits the simulation of Poisson and hard-core point fields — as described in the online documentation [1]. Furthermore, the Boolean model with discs of fixed radius can also be simulated. However, these simulations cannot be combined to form more powerful simulations.

The Pascal programs in the appendix of [11] are designed for the simulation of Poisson and cluster point processes, Poisson line processes and the Boolean model with discs. These programs are tailored to the described tasks.

Programs [2] implemented in the programming language C++, which simulate colored Voronoi tessellations described in [14], are freely available on the In-

[1] http://www.itwm.fhg.de/mab/employees/lecturews01/doc/index.html
[2] http://www.nr.no/~skare/vorSim.html

ternet. They can handle both, the two-dimensional and the three-dimensional case.

The spatstat package[3] which is written in R, a programming language for statistical computations, is intended for the statistical analysis of point patterns. It also contains functions to simulate homogeneous Poisson, hard-core, and cluster processes.

Splancs[4] is a S-Plus module for spatial point pattern analysis. Among others, this package provides the basis for the simulation of various point processes. Similarly, the S-Plus module S+SPATIALSTATS [4] also provides means to simulate several point process models.

The program Stochastic Geometry for Windows STG 4.1[5] has been developed at the Technical University of Freiberg. It is a Windows program intended for the simulation of various point fields, segment processes, germ-grain models, and tessellations. The combination of such simulations — for example, the germ-grain model based on a hard-core process — is only possible if the intermediate results are stored in files.

Apart from the field of stochastic-geometric models, Java is widely used for simulation purposes. The Hydra library [21] written in Java provides means to do Markov Chain Monte Carlo (MCMC) simulation. This library may be used to extend the framework presented in this chapter to provide more sophisticated simulations, employing methods from Markov Chain Monte Carlo. The SSJ Java library [6] is designed for discrete event simulation — as a number of other Java libraries.

3.2 Problems

When trying to combine already existing simulation programs, the following common problems may occur:

- **Different Data Structures:**
  Different programs do not use the same data structures most often. Thus, if such programs should be combined, the data structures have to be converted into each other.
- **Special Cases:**
  Often, simulation programs are designed for a special case, such as the simulation of a Poisson-Voronoi tessellation or a Boolean model with discs of

[3] http://www.maths.uwa.edu.au/∼adrian/spatstat.html
[4] http://www.maths.lancs.ac.uk/Software/Splancs/
[5] http://www.mathe.tu-freiberg.de/math/inst/stoch/Stoyan/programm.html

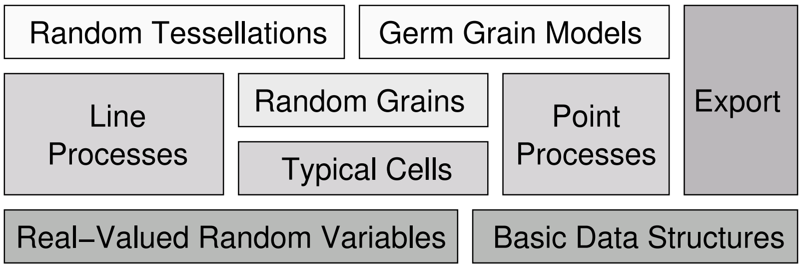


Fig. 6. The layers of the framework

constant radius. Thus, combination with an arbitrary point process or grain is not possible directly. If a Voronoi tessellation of a point process other than the Poisson process or the germ-grain model have to be simulated, substantial changes to the existing programs are necessary.

- **Adequate Abstractions:**
  Due to missing adequate abstractions and uniform interfaces, several, say, point processes may not be treated equally. Therefore, one point process may not easily be replaced by another.

The framework presented in the following section is a unified solution for the simulation of arbitrary spatial stochastic models addressing the mentioned problems in particular.

4 The Simulation Framework

In this section a framework for the simulation of spatial stochastic models is presented which has been implemented in Java. After a short overview, the individual parts are shown in detail. Some code examples are given to illustrate the implementation and the usage of the framework.

4.1 An Overview

The framework for the simulation of spatial stochastic-geometric models consists of several layers shown in Figure 6. The fundamental data structures belong to the basis of the whole framework. Important examples of these data structures are the (deterministic) geometric objects and the set representation of a realization of a spatial stochastic model. Furthermore, the simulation of real-valued random variables is also part of the lowest layer. The simulations of typical cells, line processes, and random grains are based on them (and, of course, on the basic data structures — which is true for all simulations of spatial stochastic models). The simulation of point processes requires only the basic data structures. The implementation of random grains uses as well the simulation of typical cells, which are sometimes used as grains, as the real-valued random variables, which are used as parameters of geometric objects, such as the radius of a disc. The simulation of random tessellations is based on line processes (e. g. Poisson line tessellation) and point processes (e. g. Poisson-Voronoi tessellation). And the simulation of the germ-grain model is a combination of the simulations of point processes and random grains. The export layer provides means to convert the data structure of a realization into, for example, a binary image or an encapsulated PostScript image. Whereas the data structures contain an exact representation of the simulation result, discretization may take place at this stage, for instance, when transforming the fundamental data structures into a binary image.

A UML class diagram of the classes and interfaces for the simulation of spatial stochastic models and the real-valued random variables is shown in Figure 7. This diagram gives only an overview of all the classes, while details are omitted. All the fundamental data structures, the export classes and some concrete classes are also skipped. The individual parts of this diagram are explained in detail in Section 4.5. All models are designed as interfaces and only interfaces should be referred to within implementations (as far as possible).

4.2 Naming Conventions

In order to make it easier to use the framework, each model has a method realize() that simulates the model and returns a realization. The sampling window, if necessary, is specified as argument of this method. Classes which simulate real-valued random variables have also such a method.

4.3 Basic Classes

The interfaces of the whole framework are independent of the dimension. Therefore, the utility classes DoubleBox, which represents a hyper cuboid, and ConvexPolytope, which stands for a convex polytope, must be implemented for arbitrary dimensions, since they are used in the interfaces.

Dilation necessary for the plus sampling is quite difficult for arbitrary sets. Therefore, the minimal enclosing hyper cuboid is determined and the dilation of the cuboids is computed, as mentioned in Section 4.4. DoubleBox has, for this reason, a method which can be used to dilate one DoubleBox by another, necessary for plus sampling.

The result of the simulation of a model is an object of the class RandomSet, which is bounded by a DoubleBox or a ConvexPolytope object and contains

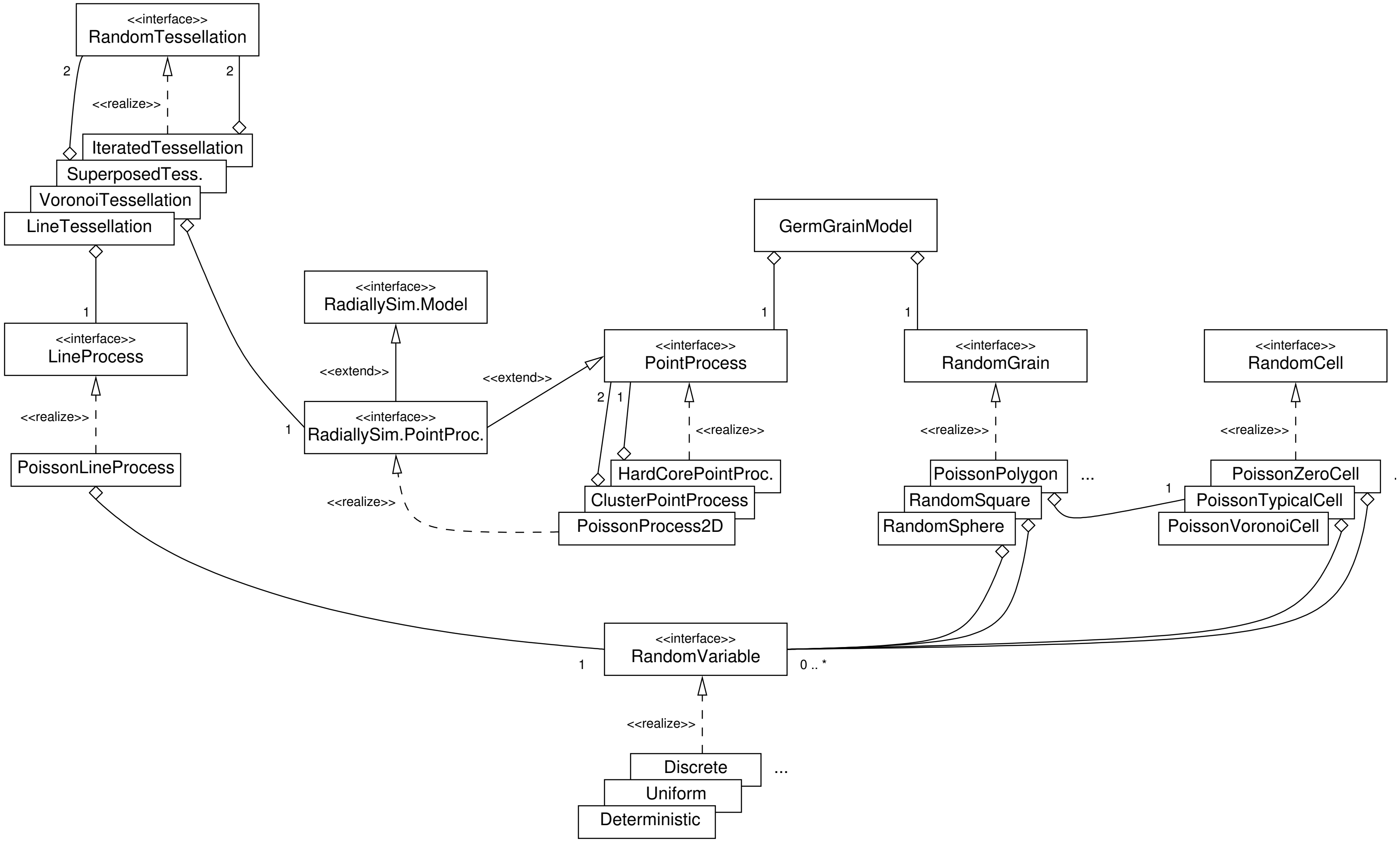


Fig. 7. An overview of the framework

elements of type RandomSetElement. It can be regarded as a *deterministic* set representing the union of all contained RandomSetElements. These elements are deterministic geometric objects, such as points (class Point), lines (class Line), and so on. A RandomSet itself is also a RandomSetElement, so iterated tessellations, for example, are represented as trees, where each level contains the cells of one iteration level. The methods of RandomSet are common container methods.

The classes for the export of a RandomSet object are based on the *Strategy Pattern* [2]. Examples are the classes BinaryImageExport, which converts the RandomSet object into a binary image — a discrete version —, and EpsExport, which produces an encapsulated PostScript file. By the way, the images of spatial stochastic models included in this paper were produced by the class EpsExport.

4.4 Basic Problems

In this section, problems are discussed that arise when programs for the simulation of the described models are developed.

4.4.1 Static Plus Sampling

One problem regarding static plus sampling is that the sampling window must be dilated with a disc (cluster and hard-core processes) or the union of all possible realizations of the primary grain (Boolean model). So, the structuring element can be a complex geometric object. Furthermore, the sampling window has not to be a rectangle which complicates the situation a little bit more. So, to compute the exact geometric object may be arbitrarily difficult. A possible way out is to work with the so-called bounding boxes of the sampling window and the structuring element. The bounding box of a geometric object is the smallest rectangle that contains the object. Dilation of a rectangle with another rectangle yields a rectangle. The width of the resulting rectangle is the sum of the widths and the height is the sum of the heights. In that way, dilation is simple.

The typical cell of the Poisson line or the Poisson-Voronoi tessellation may be used as the primary grain for the Boolean model. This is nothing special. But, these typical cells cannot be bounded. So, the union of all possible realizations would be the whole space. Simulation without edge effects is, therefore, in this case impossible with static plus sampling. But, a box can be computed such that the typical cell is contained within this box with very high probability. Then, all realizations of the typical cell that are not contained within this rectangle are discarded. The box should be chosen so big that only very few cells are rejected. In this case, the simulation results fit well the theoretical values.

4.4.2 Dynamic Plus Sampling

When a Poisson-Voronoi tessellation has to be simulated, the Poisson process cannot be simulated on a previously constructed window, since the cells can be arbitrarily large. The window within which the points are to be simulated depends on the points generated before. So, the radial simulation should be designed so that the points can be simulated one by one. Here the well-known iterator concept (cf. [2]), which is heavily used in the C++ standard template library, can be used to design such a simulation. Each point is simulated by an individual call to a method of the iterator. So, it is possible to stop the simulation when all cells of the tessellation cannot change through further points.

4.5 Detailed Description

In the following sections, parts of the class diagram shown in Figure 7 are described in detail.

4.5.1 Real-Valued Random Variables

Real-valued random variables are needed for several purposes. The parameters of random grains are one example, such as the random radius of a disc (uniformly distributed within a finite range), or the random rotation and width of a square. But, some parameters of random grains may also be deterministic. Furthermore, the orientation of the lines of a Poisson line process is a real-valued random variable. Clearly, this applies also to the simulation of the typical and the zero cell of a Poisson line tessellation. Finally, many implementations of concrete tessellations rely on the simulation of real-valued random variables.

Plus sampling which depends on the random grain is necessary for the simulation of a germ-grain model. As discussed, the bounding box of the union of all realizations of the random grain must, therefore, be constructed. This problem can be reduced to the problem of finding the minimal interval which contains all realizations of a parameter of the random grain.

The design of real-valued random variables is shown in Figure 8. A concrete real-valued random variable must implement the common interface RandomVariable. With the method realize(), a new realization of the random variable

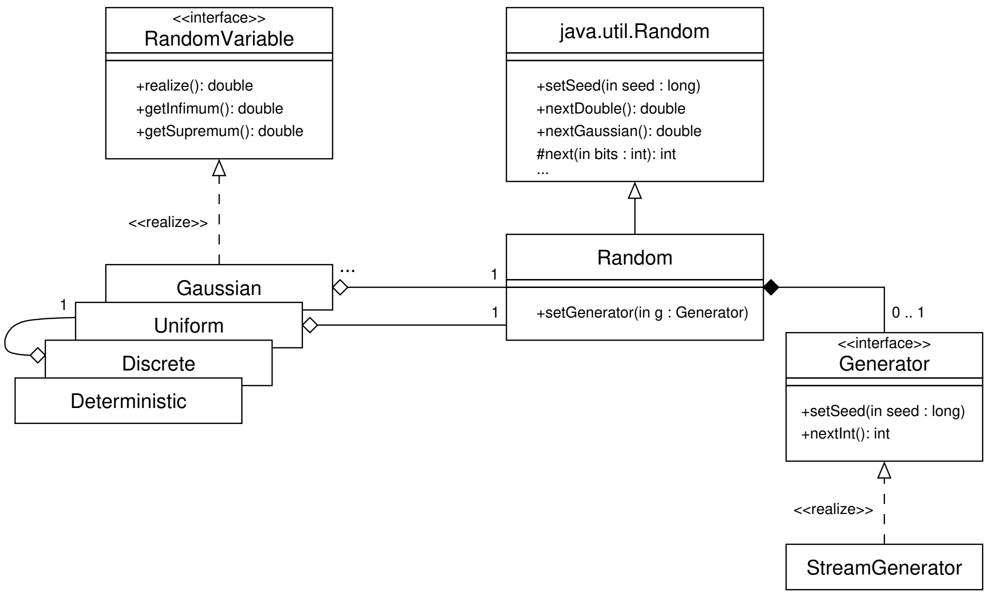


Fig. 8. Classes and interfaces for the simulation of real-valued random variables

can be obtained. The methods getInfimum() and getSupremum() return the lower and upper bounds, respectively, of the range of the random variable. So, this solves the plus sampling problem for the simulation of germ-grain models. Since an implementation of a deterministic random variable is provided by the class Deterministic, real-valued random variables can also act as double variables. Thus, a random grain can have a mixture of random and deterministic parameters for a maximum of flexibility.

The implementation of the concrete classes for real-valued random variables is based on the Java class java.util.Random. To allow further pseudo-random number generators, a sub-class called Random of the mentioned class was introduced. This class provides a method to set the pseudo-random number generator (of type Generator). The sample generator StreamGenerator reads the random numbers from a stream. So, in case another generator should be used or the Java pseudo-random number generator proves bad, it can easily be replaced. For this reason, the class Random should only be used for the implementation of Uniform and Gaussian. The implementation of all other classes must rely on random variables (of type RandomVariable), such as Uniform.

4.5.2 Point Processes

The design of point processes is shown in Figure 9. The interface PointProcess is common to all point processes. It provides the method realize() which simulates the point process within the given bounding box and returns the resulting RandomSet object. This RandomSet contains only Point objects. Additionally, there is a method to determine the dimension of the process. It can be used to test, whether two processes have the same dimension and can,

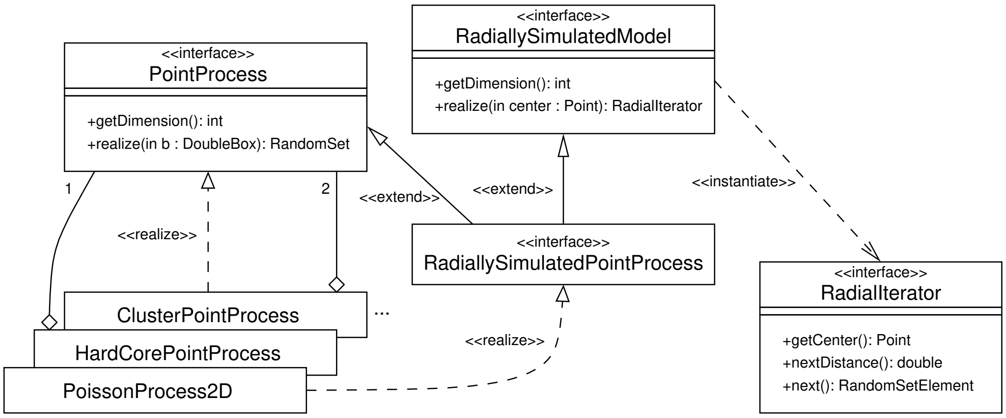


Fig. 9. Classes and interfaces for the simulation of point processes

therefore, be combined.

As discussed in Section 4.4, radial simulation can be designed using the Iterator Pattern [2]. The interface RadiallySimulatedModel should be implemented by all classes for models which permit radial simulation. For such point processes, as for example the Poisson process, there is a special interface RadiallySimulatedPointProcess. The number of objects generated by the RadialIterator, which is returned when calling the method realize() for a radially simulated model, is — theoretically — unlimited. The simulated objects have increasing distance from the specified center point. The distance of the next object, which will be returned by the following call to next(), can be questioned through a call of nextDistance().

The concrete classes HardCorePointProcess and ClusterPointProcess are implemented in a generic way independent of the dimension of the given point processes. The simulation of the Poisson process is however not easily done by one class for all dimensions. So, such a concrete class is implemented separately for each dimension.

4.5.3 The Boolean Model

For an implementation of the Boolean model, or, more generally, the germ-grain model, the simulations of point processes and primary grains are needed. Figure 10 shows the interfaces and classes used for grains. A primary grain must implement the interface RandomGrain. This interface contains the method realize() which simulates the grain. Additionally to the method getDimension(), there is a method to retrieve the bounding box within which all realizations of the primary grain are guaranteed to lie. This method is used to compute the enlarged sampling window for the plus sampling within the simulation of the germ-grain process. The parameters of the implementations of RandomGrain have type RandomVariable, as mentioned, and the bounding box of the grain

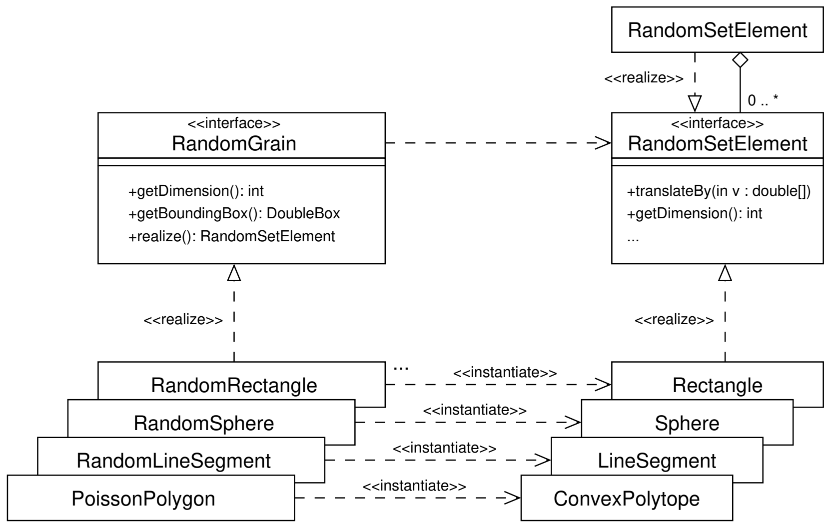


Fig. 10. Classes and interfaces for the simulation of primary grains

can be determined through the range of its parameters, which is provided by RandomVariable. The realization of a primary grain is a RandomSetElement, which provides the method translateBy() to translate it. This is necessary to translate a realization of the primary grain into a germ.

For the implementation of the germ-grain model (class GermGrainModel in Figure 7) the design proves beneficial — as in the case of cluster and hard-core point processes. The implementation of the germ-grain model, whose method realize() follows, is completely independent of the dimension of the underlying models.

```
private RandomGrain primaryGrain;    // initialized elsewhere
private PointProcess germProcess;    // initialized elsewhere

public RandomSet realize(DoubleBox boundingBox) {
    // the bounding box must have the same dimension
    // as the process itself
    if (boundingBox.getDimension() != getDimension())
        throw new IllegalArgumentException("Wrong Dimension");

    // compute enlarged bounding box (plus sampling)
    DoubleBox grainBox = primaryGrain.getBoundingBox();
    DoubleBox box = boundingBox.plusSampling(grainBox);

    // realize the germ process
    RandomSet germRandomSet = germProcess.realize(box);

    // RandomSet for the result
    RandomSet grainRandomSet = new RandomSet(boundingBox);
```

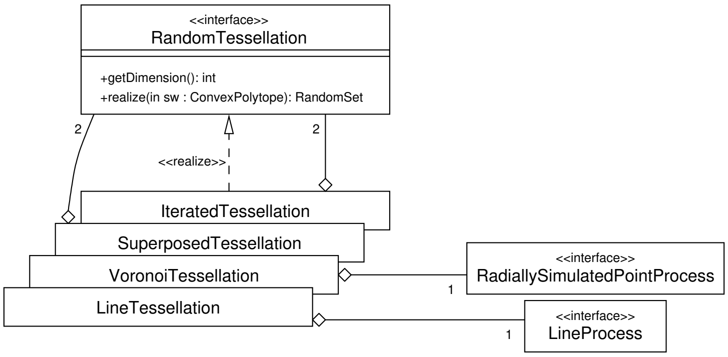


Fig. 11. Classes and interfaces for the simulation of random tessellations

```
    Point germ;
    RandomSetElement grain;
    for ( Iterator  i = germRandomSet.iterator(); i.hasNext();) {
        // for every germ ...
        germ = (Point) i.next ();
        // simulate the primary grain
        grain = primaryGrain.realize ();
        // ... and translate it into the germ
        grain . translateBy(germ.getCoordinates());
        // finally , add it to the resulting set , if the bounding
        // box of the grain intersects with the given bounding box
        if (grain . getBoundingBox().intersectsWith(boundingBox))
            grainRandomSet.add(grain);
    }

    return grainRandomSet;
}
```

4.5.4 Random Tessellations

The interface RandomTessellation (cf. Fig. 11) is used for random tessellations whose cells are convex polytopes. The RandomSet object returned by a call to realize() represents a tree with ConvexPolytope objects as leafs and further RandomSet objects as (inner) nodes. The difference to previous models is that the used sampling window can be an arbitrary convex polytope, and not only a box. The implementations LineTessellation and VoronoiTessellation are tailored to a certain dimension, in order to make it feasible. But, the implementation of IteratedTessellation, which does independent iteration, can be completely independent of the dimension of the models. In the following, the implementation of the method realize() of the class IteratedTessellation is shown:

```
private RandomTessellation initial;     // initialized elsewhere
```

```
private RandomTessellation component; // initialized elsewhere

public RandomSet realize(ConvexPolytope samplingWindow) {
    // RandomSet for the result of the simulation
    RandomSet result = new RandomSet(samplingWindow);

    // first , simulate the initial tessellation
    RandomSet initialTessellation = initial . realize (samplingWindow);

    // iterate all elements of the initial tessellation
    for ( Iterator i = initialTessellation . iterator (); i .hasNext();) {
        RandomSetElement cell = (RandomSetElement) i.next();

        // ... cell is iterated by the component tessellation
        if ( cell instanceof ConvexPolytope)
            result .add(component.realize((ConvexPolytope) cell));
        else
            result .add( cell );
    }

    return result;
}
```

Obviously, several iterated tessellations may be nested. Then, the initial tessellation must not be an iterated tessellation, since the resulting RandomSet typically contains further RandomSet objects in this case. However, this is no real restriction since the component tessellation can be an iterated tessellation.

4.6 Using the Framework

The simulation framework described in the present section provides everything to perform simulations of basic stochastic-geometric models. So, a possible usage of the framework is to simply combine the available classes to do the required simulation. For example, a sample of the two-dimensional Boolean model with discs, whose radius is randomly chosen, can be generated as follows:

```
PointProcess germs = new PoissonProcess2D(0.001);
RandomGrain grain = new RandomSphere(new double[] {0.0, 0.0},
                                     new Uniform(10.0, 30.0));
GermGrainModel g = new GermGrainModel(germs, grain);
DoubleBox box = new DoubleBox(new double[] {0.0, 0.0},
                              new double [] {400.0, 400.0});
```

```
RandomSet r = g.realize(box);
```

The class RandomSphere represents a hyper sphere — independent of the dimension. So, the first parameter of the constructor which specifies the center also determines the dimension of the sphere — or disc, in this case. The second parameter of the constructor gives the distribution of the radius — here, a uniform distribution within the interval $[10, 30]$ has been chosen.

The sampling window for the simulation is a DoubleBox object which represents a rectangle with lower left corner $(0, 0)$ and upper right corner $(400, 400)$.

The call of the method realize() generates the sample of the Boolean model with germ intensity 0.001 and discs with radius randomly chosen from the interval $[10, 30]$.

Due to the use of interface types, it is possible to replace the Poisson process and the primary grain by any other (two-dimensional one).

Another example is the simulation of independently iterated tessellations:

```
RandomTessellation child = new IteratedTessellation(
        new LineTessellation(new PoissonLineProcess(0.05)),
        new VoronoiTessellation(new PoissonProcess2D(0.005))
    );
RandomTessellation tess = new IteratedTessellation(
        new LineTessellation(new PoissonLineProcess(0.02)),
        child
    );
double n = 100.0;
ConvexPolytope sw = new ConvexPolytope(new Point[] {
        new Point(new double[] {n,n}),
        new Point(new double[] {-n,n}),
        new Point(new double[] {-n,-n}),
        new Point(new double[] {n,-n})
    });
RandomSet r = tess.realize(sw);
```

An iterated tessellation being a nesting of three tessellations is constructed. The tessellation is a Poisson line tessellation with intensity 0.02 whose cells are subdivided by another Poisson line tessellation with intensity 0.05 that in turn contains a nested Poisson-Voronoi tessellation with intensity 0.005.

The sampling window for the simulation of tessellations must be a convex polytope. The, in this case, convex polygon is specified by its vertices (in counter-clockwise order).

Iterated tessellations may be nested arbitrarily deep. As in the previous example, the concrete tessellations, point and line processes may be replaced by any other.

For the simulation of more complex models, the framework may easily be extended. If the type of the new model already exists, say PointProcess, another class providing this interface has to be implemented. For the implementation all classes of the framework can be used, simplifying this task. In case the type of the new model does not yet exist, a new interface for this type has also to be made. This interface should adhere to the following conventions:

(1) It has to contain a method realize() that triggers the generation of a new sample. This method may have a parameter specifying the sampling window which should either be an object of type DoubleBox or an object of type ConvexPolytope. It should return an object of type RandomSet being the result of the simulation.
(2) Furthermore, it should also provide a method getDimension() which returns the dimension of the model. (This is necessary to support the combination of models while ensuring that they have the same dimension.)
(3) It is important that the interface is independent of the dimension. This implies that no classes may be used as parameter types or return types of methods that depend on the dimension. (However, the classes DoubleBox, ConvexPolytope, and RandomSet are independent of the dimension and can thus be used.)

4.7 Contributions

Summarizing, the main contributions of the presented framework are:

- All interfaces of the framework are similar. A simulation can be done by calling the method realize(), eventually providing a sampling window.
- To allow for the simple and correct combination of models, the interfaces also contain the method getDimension() for the determination of the dimension.
- To support plus sampling for the germ-grain model without great effort, the methods getInfimum() and getSupremum() are introduced for the simulation of real-valued random variables.
- For the edge-effect free simulation of a Voronoi tessellation, the underlying point process has to be simulated radially, i. e. the points have to generated one by one with increasing distance to the origin. Therefore, the interface RadiallySimulatedPointProcess has been introduced which produces a RadialIterator object. The latter generates the points one by another.
- The choice of a convex polytope as the sampling window for the simulation of tessellations has proven useful. Thereby, it is possible to simulate iterated

tessellations without great effort.

- The simulation of real-valued random variables has been equipped with a replaceable pseudo-random number generator. Thus, an arbitrary pseudo-random number generator may be chosen.

5 Summary and Conclusions

The dimension-independent framework design for the simulation of spatial stochastic models presented in this paper — illustrated only with a few basic models for ease of understanding — gives a unified and simple way to generate samples of such models. Further models may be added without difficulties. For their implementation, the available classes and interfaces can be used, which makes this task easier. So, the framework helps to speed up the development of new simulation programs. In addition, the chosen design leads to better source code which is readable without great effort, as shown for the Boolean model and independently iterated tessellations. The basic problems, such as edge effects and plus sampling, arising when simulating spatial stochastic models have been treated in the present paper and possible solutions were given. Although, the framework has been implemented in Java, it can be implemented similarily in any other object-oriented programming language. The contributions of this paper are more general and not restricted to an implementation.

Acknowledgment

The authors are grateful to Matthias Grabert and Ingo Melzer for helpful comments. They would also like to thank the anonymous referees for several useful remarks which helped to improve the presentation of the results.

References

[1] F. Baccelli, M. Klein, M. Lebourges, and S. Zuyev, Stochastic Geometry and Architecture of Communication Networks, Journal of Telecommunication Systems 7 (1997) 209–227.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, Reading, 1995).

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha, The Java Language Specification Second Edition (Addison-Wesley, Reading, 2000).

[4] S. Kaluzny and S.C. Vega, S+SPATIALSTATS (Springer-Verlag, New York, 1997).

[5] N.C. Kenkel, J.A. Hoskins, and W.D. Hoskins, Local Competition in a Naturally Established Jack Pine Stand, Canadian Journal of Botany 67 (1989) 2630–2635.

[6] P. L'Ecuyer, SSJ — A Java Library for Stochastic Simulation: User's Guide, Department of Computer Science and Operations Research, University of Montreal, January 2003. [6]

[7] R. Maier, J. Mayer, and V. Schmidt, Distributional Properties of the Typical Cell of Stationary Iterated Tessellations, Mathematical Methods of Operations Research 59 (2004). (to appear)

[8] R. Maier and V. Schmidt: Stationary Iterated Tessellations, Advances in Applied Probability 35 (2003) 337–353.

[9] T. Mattfeldt, H. Frey, and C. Rose, Second-Order Stereology of Benign and Malignant Alterations of the Human Mammary Gland, Journal of Microscopy 171 (1993) 143–151.

[10] T. Mattfeldt, V. Schmidt, D. Reepschläger, C. Rose, and H. Frey, Centered Contact Density Functions for the Statistical Analysis of Random Sets, Journal of Microscopy 183 (1996) 158–169.

[11] J. Mecke, R.G. Schneider, D. Stoyan, and W.R.R. Weil, Stochastic Geometry, DMV Seminar 16 (Birkhäuser, Basel, 1990). (in German)

[12] K. Mecke, Integral Geometry and Statistical Physics, International Journal of Modern Physics B 12 (1998) 861-899.

[13] K. Mecke, T. Buchert, and H. Wagner, Robust Morphological Measures for Large-Scale Structure in the Universe, Astron. Astrophys. 288 (1994) 697–704.

[14] J. Møller and Ø. Skare, Bayesian Image Analysis with Coloured Voronoi Tessellations and a View to Applications in Reservoir Modelling, Research Report 6, MaPhySto, Århus, 2001.

[15] J. Ohser and F. Mücklich, Statistical Analysis of Microstructures in Materials Science (John Wiley & Sons, Chichester, 2000).

[16] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams (John Wiley & Sons, Chichester, 2000).

[17] M.P. Quine and D.F. Watson, Radial Generation of n-dimensional Poisson Processes, Journal of Applied Probability 21 (1984) 548–557.

[18] P. Soille, Morphological Image Analysis: Principles and Applications (Springer-Verlag, Berlin, 1999).

[19] D. Stoyan, W.S. Kendall, and J. Mecke, Stochastic Geometry and its Applications (John Wiley & Sons, Chichester, 1995).

[6] http://www.iro.umontreal.ca/~dift6561/java/guide.pdf

[20] Unified Modeling Language (UML), Version 1.5, Object Management Group (OMG), formal/03-03-01, 2003.
Available at http://www.omg.org/technology/documents/formal/uml.htm

[21] G.R. Warnes, Hydra: A Java Library for Markov Chain Monte Carlo, Journal of Statistical Software 7 (2002).