# MODELLING AND MONTE-CARLO SIMULATIONS FOR A CONCURRENCY PROBLEM

Doina LOGOFĂTU, Felix DIETRICH
Department of Computer Science and Mathematics
University of Applied Sciences
Lothstrasse 64, 80335, Munich
{doina.logofatu, felix.dietrich}@hm.edu

**ABSTRACT**
This paper describes Monte-Carlo simulation techniques that calculate how effective the risk of a blockage in case of concurrent processes is. We start describing a common problem in current computer science, the deadlock. This is followed by a mathematical abstraction of the problem. Three solution models are presented for it, two of them designed for multidimensional cases. These models are then tested in experiments and compared against the exact solutions.

**KEY WORDS**
Deadlock, Continuous Probability Theory, Geometry, Monte-Carlo Simulation, Parallelization, Concurrency, and Threading

## 1. INTRODUCTION

A lot of systems use nowadays locking mechanisms for concurrency control. A lock of an object acquires when a transaction needs an assurance that some object will not change in some unpredictable manner [4, 9].

Problems with this approach arise when an object cannot be divided up and is desired by more than one party. In this case, a compromise can be found by defining disjunctive time spans where only a single party holds the object. When the time is over, it is handed on to another party. This approach directly leads to concurrency if the time spans are not set in advance [4, 7, 9].

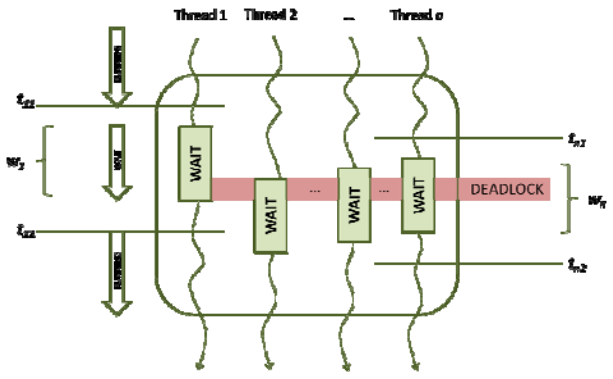



**Figure 1. Process containing several threads with the risk of a deadlock, thread *i* starts waiting for $w_i$ seconds between $t_{i1}$ and $t_{i2}$**

Concurrency in computer science means that several computations are executing simultaneously, potentially interacting with each other, and eventually sharing the same resources (memory, CPU). This often leads to a serialization of access, what implies a ranking of the participating events. In many cases it is unimportant in which order the events are executed or if they are processed simultaneously [4, 9].

Let us consider the scenario of several threads running in parallel (Fig. 1, state RUNNING) and further thread 1 waiting for thread 2 to release the lock of some object (Fig. 1, state WAIT). This waiting starts in a time span $[t_{11}, t_{12}]$ with equal probability distribution and takes some time $w_1$. Let now thread 2 also start waiting in a time span $[t_{21}, t_{22}]$ for an object thread 1 has to release. This takes the time $w_2$. It sometimes occurs that both threads are in the WAIT state at the same time. In this situation, none of them can proceed without the other, what is then called a *deadlock*. The application cannot operate any further and must at least shut down the two threads. The problem we will discuss in this paper is to minimize the probability of this event.

To solve this problem, we first describe it in a more abstract, mathematical way. We start with $X_i$ being a continuous random variable with uniform distribution, representing the thread $i$. Let $t_{i1}$ and $t_{i2}$ be the first/last possible points in time where thread $i$ might start waiting. The density $f_i$ of $X_i$ can now be described by

$$f_i(x) = \frac{I_{\{t_{i1}, t_{i2}\}}(x)}{t_{i2} - t_{i1}} \tag{1}$$

with $x \in \mathbb{R}$ and $I$ an indicator function on $\mathbb{R}$. Let then $w_i$ be the time thread $i$ needs to wait for another one to release the lock of an object. For simplicity, all $w_i$ are equal and constant for all $i$ (=$w$). To define the problem, we need a combination $C$ of the random variables. Let $n$ be the number of random variables and $i, j \in \{1, \ldots, n\}$, $i \neq j, x_i \in \{t_{i1}, t_{i2}\}$. Then $C$ can be defined as

$$C = \{c \in \bigcup_i \{t_{i1}, t_{i2}\} \mid \forall j : x_j - w \leq c \leq x_j + w\} \tag{2}$$

When $C \neq \emptyset$, the probability $p$ of a deadlock is nonzero and can be written as

$$p = \frac{|C|}{\left|\bigcup_i x_i\right|} \tag{3}$$

For two random variables $X_1$ and $X_2$, there is a geometric interpretation of this probability. Let $t_{11}$ and $t_{12}$ be the boundary points of the range of $X_1$ and let $t_{21}$ and $t_{22}$ the boundary points of the range of $X_2$. Then $C$ is the area of a rectangle $A$ in a two-dimensional space formed by $X_1$ and $X_2$ bordered by two lines $y_1 = x + w$ and $y_2 = x - w$ with $x \in X_1$ and $y_1, y_2 \in X_2$.

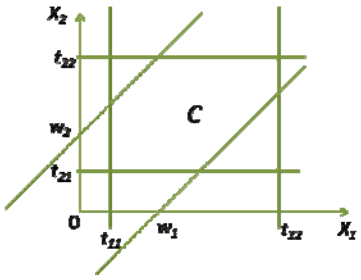



**Figure 2. *C* for two random variables**

The probability of a deadlock is

$$p = \frac{C}{A} \tag{4}$$

We will use this formula for our solutions in the next part.

## 2. SOLUTION MODELS

To calculate the area of $C$ in the two-dimensional case, we will now introduce three solution models – the Monte Carlo Simulation, a numerical method and the exact method. The first two models were chosen because they can be extrapolated very easily. This will be an advantage when more than two random variables are chosen to generate $C$.

### 2.1 Solution by Monte Carlo Simulation

Monte Carlo methods rely on repeated random sampling to compute their results. The idea came from Enrico Fermi in the 1930s, when he used Monte Carlo in the calculation of neutron diffusion [13]. Monte Carlo methods are mostly used if finding an exact solution to a problem is very complex or impossible. A classic example of using the method is the approximation of π [8]. There are many variants of Monte Carlo methods, but all of them follow a particular pattern:

1. For the originally mathematical model a stochastic model must be found which describes the problem.
2. A sequence of random numbers must be generated. These values should simulate possible real situations.
3. There must be found estimations from the random values for the original problem.

Monte Carlo methods are used in many different areas like mathematics, physics, according permission on the rivers or in the financial sector [3, 5, 6, 10, 11]. In mathematics, they are often used for evaluating definite integrals with complicated boundary conditions. Especially for multidimensional integrals the Monte Carlo integration can be particularly efficient [14]. For example, two of the most used Monte Carlo methods for integration were compared by Hörmann and Leydold in 2005 [14]. In the financial sector, Monte Carlo methods are used, among others, to reduce the uncertainty involved in estimating future outcomes [3]. A wide area of application for Monte Carlo methods can also be found in physics [5, 6]. Zabenkov and Kochubey have used the Monte Carlo simulation to study the dependence of the spatial resolution of a luminescent object inside the skin [13]. Below, we are using Monte Carlo simulation to calculate the area of $C$.

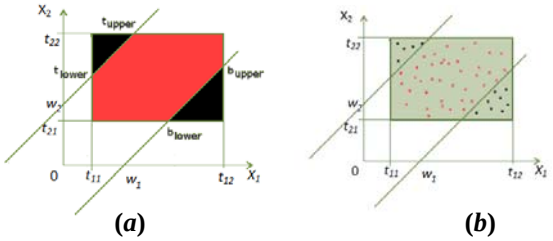
In our solution model, we generate random points in the rectangle formed by the two random variables $X_1$ and $X_2$ as defined before.

```
void MCSimulation::Execute()
{
        double counter = 0;

        for(int i=0; i<points; i++)
        {
                if(InSolution(GeneratePoint()))
                        counter++;
        }
        result = counter;
}
```

**Figure 3. Code in C++ for Monte-Carlo method**

### 2.2 Solutions by numerical and exact integration

If we do not choose random points but divide the rectangle in equally spaced subparts, we can approximate $C$ numerically. The points lie on the edges between the parts and the number of red/all points is used to approximate the probability. As the problem is in two-dimensional space, we also can use normal integration to solve it exactly.




***(a)*** ***(b)***

**Figure 4. Two solution models (a) Exact solution by integration (b) Numerical solution**

```
member this.solve (problem:Problem, solvers:int)=
        let w = problem.Bandwith
        let mutable counter = 0

        let xlist = [ t11 .. stepSize .. t12 ]
        let ylist = [ t21 .. stepSize .. t22 ]
        for x in xlist do
           for y in ylist do
              if x <= y+w && y<= x+w then
                  counter <- counter + 1

        (float)counter/
        (float)(xlist.Length*ylist.Length)
```

**Figure 5. Code in F# for the numerical integration**

The red area shown in Figure 4 (a) represents $C$. It can be computed by integrating the lines [20].

This is done by setting values for the integral ranges:

$$b_{lower} = \max(\min(t_{12}, t_{21} + w), t_{11}) \quad (5)$$

$$b_{upper} = \max(\min(t_{12}, t_{22} + w), t_{11}) \quad (6)$$

$$t_{lower} = \min(\max(t_{11}, t_{21} - w), t_{12}) \quad (7)$$

$$t_{upper} = \min(\max(t_{11}, t_{22} - w), t_{12}) \quad (8)$$

Then $C$ can be computed by integrating the lines.

$$C = \int_{t_{lower}}^{t_{upper}} (x + w)dx - \int_{b_{lower}}^{b_{upper}} (x - w)dx + \\ + \max(0, (t_{12} - t_{upper}) \cdot (t_{22} - t_{21})) - \\ - \max(0, (t_{12} - b_{upper}) \cdot (t_{22} - t_{21})) \quad (9)$$

The complete area $A$ is $(t_{12}-t_{11}) \cdot (t_{22}-t_{21})$.

## 3. EXPERIMENTS

We will now compare the different solution models. In the experiments, only one parameter is changed at a time, the others remain constant. Interesting parameters are $t_{11}$, $t_{12}$, $t_{21}$, $t_{22}$, $w$, the number of MCS points, and the step size of the numerical integration. $t_{11}$, $t_{12}$, $t_{21}$ and $t_{22}$ will remain constant in all experiments as they do not significantly change the performance of the methods compared to each other. The parameter $w$ is called "bandwidth" in all experiments, as it defines the half-distance between the two lines in two-dimensional space.

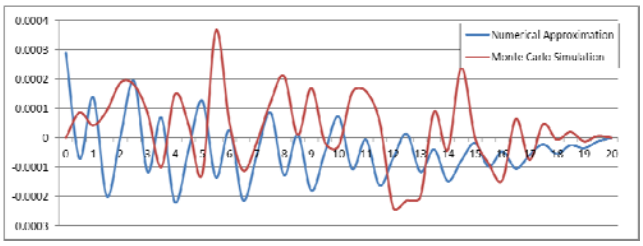
The initial problem model can also be transferred in other areas such as database transactions or synchronization. In a multithreaded environment knowing the probability of a deadlock can significantly reduce the overhead in detecting critical situations.

```
member this.solve (problem:Problem, solvers:int)=
 let w = problem.Bandwith

 let xBottom = Math.Max(Math.Min(t12,t21+w), t11)
 let xBottomTop=Math.Max(Math.Min(t12,t22+w),t11)

 let xTop = Math.Min(Math.Max(t11, t22-w), t12)
 let xTopBottom=Math.Min(Math.Max(t11,t21-w),t12)

 // integrate the upper line
 let A = 1.0 / 2.0*xTop*xTop +
         w * xTop - t21 * xTop -
         (1.0/2.0*xTopBottom*xTopBottom +
         w* xTopBottom - t21*xTopBottom) +
         Math.Max(0.0, (t12-xTop)*(t22-t21))

 // integrate the lower line
 let B = 1.0 / 2.0*xBottomTop*xBottomTop -
         w * xBottomTop - t21 * xBottomTop -
         (1.0/2.0*xBottom*xBottom –
         w* xBottom - t21*xBottom) +
         Math.Max(0.0,(t12-xBottomTop)*(t22-t21))

 // the area we search for is the difference
 // between A and B
 let area = A-B
 let completeArea = (t12-t11)*(t2-t1)
 // return the probability
 area / completeArea
```

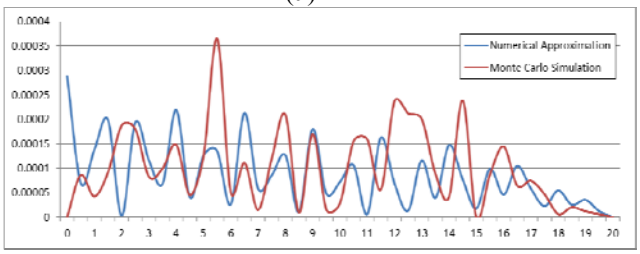
**Figure 6. Code in F# for the exact integration**

### 3.1 First experiment: Changing the bandwidth w

Letting all other parameters constant, $w$ is changed during this experiment from 0 to 20 in steps of 0.5. The output is the probability $p$. All methods are applied to these problems, giving the following results:




(*a*)

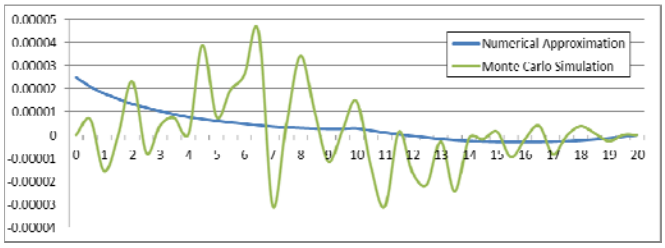



(*b*)

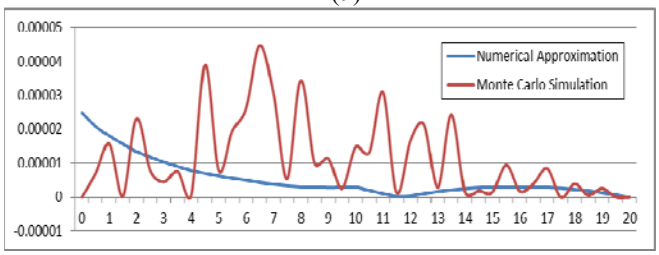
**Figure 7. Experiments with change of the bandwith. Number of points in both numerical method and Monte Carlo Simulation: 6e6.**
**(a) Difference to the exact value**
**(b) Absolute difference to the exact value**

```
// define the random variables used in the experiments
let X1 = new RandomVariable(0.0,10.0)
let X2 = new RandomVariable(0.0,10.0)
let xList = [ X1; X2 ]

// define the solvers
let NISolver = new NumericalIntegration.Solver(0.02);
let MCSolver = new MonteCarloSimulation.Solver(500000);
let EISolver = new ExactIntegration.Solver();

// Experiment 1 - Changing the bandwith
let bandwidths = [ 0.0 .. 0.5 .. 10.0 ]

// Numerical Integration
let NIexperiments =
  [| for i in bandwidths
  do yield new Experiment(new Problem(xList,i), NISolver)
  |]

let NIresults =
  NIexperiments
  |> Array.map (fun exp -> exp.execute)

// Monte Carlo Simulations
let MCSexperiments =
  [| for i in bandwidths
  do yield new Experiment(new Problem(xList,i), MCSolver)
  |]

let MCSresults =
  MCSexperiments
  |> Array.Parallel.map (fun exp -> exp.execute)

// Exact Integration
let EIexperiments =
  [| for i in bandwidths
  do yield new Experiment(new Problem(xList,i), EISolver)
  |]

let EIresults =
  EIexperiments
  |> Array.map (fun exp -> exp.execute)
```

**Figure 8. Experimental scenario in F#**




(*a*)




(*b*)

**Figure 9. Monte-Carlo Simulations with change of the bandwith. Number of points in both numerical method and Monte Carlo Simulation: 8e8.**
**(a) Difference to the exact value**
**(b) Absolute difference to the exact value**

An interesting point here is the comparison of the two numerical approximations with 6e6 and 8e8 points. In Figure 7, the values oscillate, whereas in Figure 9, they approach zero nearly monotonic.

Figures 10 and 11 show Monte-Carlo simulation results for scenarios in two and three dimensions. For the last one, no exact solution has been applied, whereas the simulation could be set up very easily.

### 3.2 Second experiment: Scenario with two threads and ascending number of points

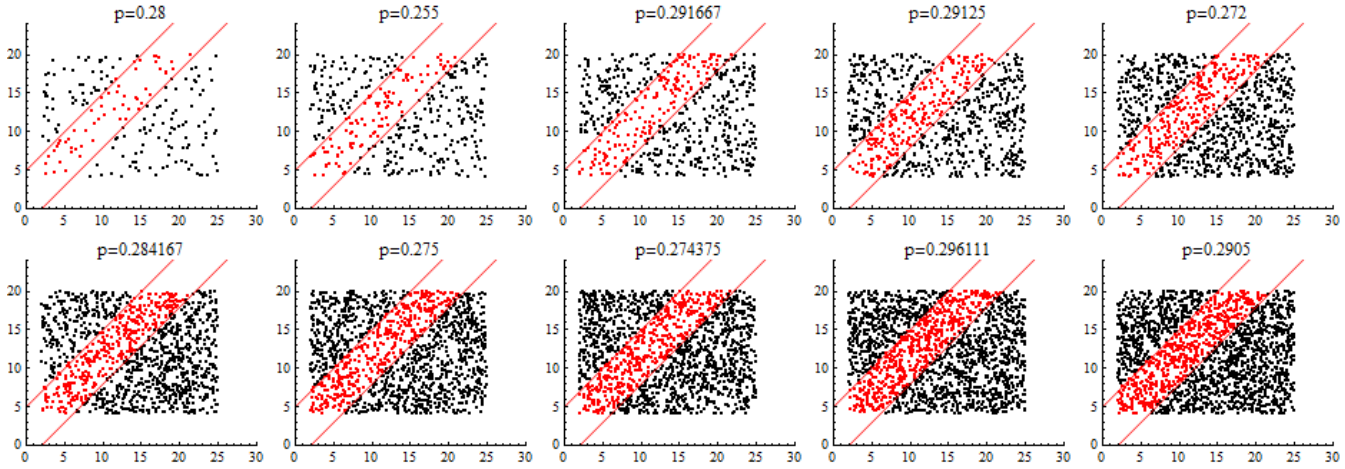



**Figure 10. Plots of Monte-Carlo Simulations with point numbers ranging from 200 up to 2000; $s_1$=2, $t_1$=4; $s_2$=25; $t_2$=20; $w_1$=5, $w_2$=2; Exact probability: $p_{exact}$=0.272891**

### 3.3 Third experiment: Three threads scenarios

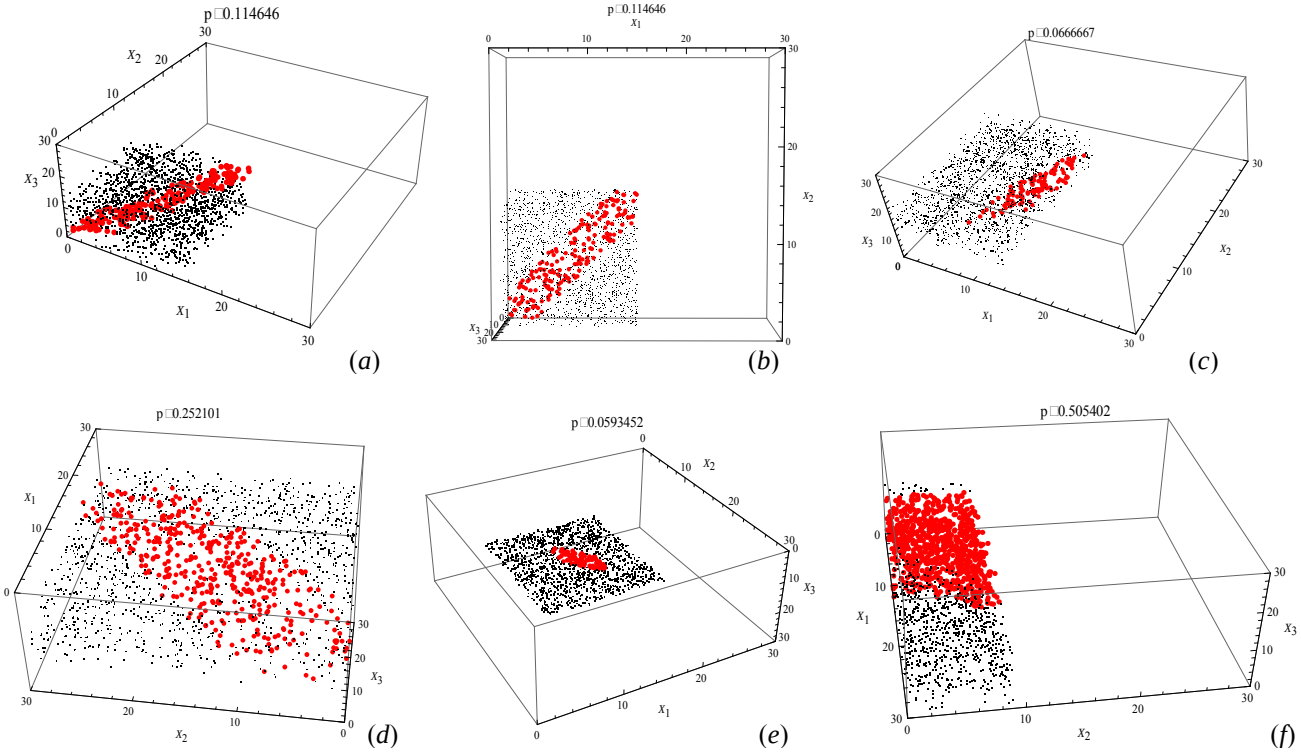



**Figure 11. Three dimensional Monte-Carlo simulations. The number of points is always 5e3.**

| | $t_{11}$ | $t_{12}$ | $t_{21}$ | $t_{22}$ | $t_{31}$ | $t_{32}$ | $w$ | $p_{approximated}$ |
|---|---|---|---|---|---|---|---|---|
| **(a)** | **0** | **15** | **0** | **15** | **0** | **15** | **3** | **0.114646** |
| **(b)** | **0** | **15** | **0** | **15** | **0** | **15** | **3** | **0.114646** |
| **(c)** | **0** | **30** | **0** | **30** | **9** | **19** | **3** | **0.0666667** |
| **(d)** | **0** | **30** | **0** | **30** | **9** | **16** | **8** | **0.252101** |
| **(e)** | **5** | **21** | **5** | **21** | **13** | **14** | **2** | **0.0593452** |
| **(f)** | **0** | **30** | **0** | **9** | **0** | **16** | **12** | **0.505402** |

## 4. CONCLUSION

The two approximation models described in this paper do not have the same accuracy as the exact method, though they produce output very close to the desired one. This is getting important when exact solutions are not possible or much too expensive in needed computational power. Furthermore, if there are outside conditions, that have to be taken into account, the Monte Carlo method seems more flexible, as these can be simply added by modifying the simulation. As for the integration methods, in that case a new mathematical model has to be defined and implemented addressing these new conditions. A different probability distribution, for example, could not be easily modeled by numerical integration. In this paper, we gave a basic example, able to address the three solution methods. The described scenario can be extended in a variety of ways. For example, we can look at more than two parallel threads or consider different wait times ($w_i$) for the locking (Figure 1). In this case we would have to deal with multidimensional integrals for calculating the exact probability.

## 5. USED TOOLS

All computation was performed on an Intel Core 2 Duo T7500 Processor. The code for the experiments was written in F# and C++, using the .NET 4.0 Platform and Visual Studio 2010 Beta 2. The graphics were made in Mathematica 7.0 [12], GIMP 2.0 and Microsoft Excel 2010.

## REFERENCES


[1] P. L'Ecuyer, A.B. Owen (Eds.): *Monte Carlo and Quasi-Monte Carlo Methods*, Springer, Berlin/Heidelberg, 2009.

[2] J.E. Gentle: *Random Number Generation and Monte Carlo Methods*, 2nd ed., Springer, Berlin/Heidelberg, 2003.

[3] P. Glasserman: *Monte Carlo Methods in Financial Engineering*, Springer, Berlin/Heidelberg, 2003.

[4] B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, T. Peierls, *Java Concurrency in Practice*, Addison-Wesley Longman, Amsterdam, 2006.

[5] W. K. Hastings, *Monte Carlo Sampling Methods Using Markov Chains and Their Applications*, *Biometrica*, Vol. 57, No. 1, pp 97-109, 1970.

[6] D. P. Landau, K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, New York Cambridge University Press, 2005.

[7] D. Logofătu, F. Dietrich, E. Pavlidis, D. Wilfert: Modelling the Probability of Deadlocks in a Multithreading Process, *10th International Conference on Development and Application Systems*, Suceava, 2010.

[8] D. Logofătu: *Eine praktische Einführung in C*, pp. 209-226, entwickler-press, München, 2008.

[9] O. Maslennikov: Systematic Generation of Executing Programs for Processor Elements in Parallel ASIC of FPGA-Based Systems and Their Transformation into VHDL-Descriptions of Processor Element Control Units, *Lecture Notes of Computer Science 2328*/2002, p. 272, 2002.

[10] S. M. Mesbah, R. Kerachian, M.R. Nikoo: Developing real time operating rules for trading discharge permits in rivers: Application of Bayesian Networks, *Environmental Modelling and Software 24* (2), pp. 238-246, Elsevier Ltd., 2008.

[11] N. Metropolis, S. Ulam, "The Monte Carlo Method", *Journal of the American Statistical Association* 44 (247), pp. 335-341, 1949.

[12] H. Ruskeepaa: *Mathematica Navigator: Mathematics, Statistics and Graphics*, Academic Press, 3rd Edition, 2009.

[13] I. V. Zabenkov, V.I. Kochubey: "Monte Carlo simulation of the recording of fluorescent objects in the skin", In: *Optics and Spectroscopy*, vol. 107, nr. 6, pp. 898-902, Springer, 2009.

[14] W. Hörmann, J. Leydold: "Monte Carlo Integration using Importance Sampling and Gibbs Sampling", In: H. Dag and Y. Deng (eds.), *Proceedings of the International Conference on Computational Science and Engineering*, pp.92-97, Istanbul, 2005.