Bull. Nov. Comp. Center, Comp. Science, 46 (2022), 27–41
© 2022 NCC Publisher

Architecture of the Cellular Automata Topologies Library*

Yu.G. Medvedev

Abstract. The paper presents new software named Cellular Automata Topologies Library (CATlib). It includes a set of system routines written in the C language, and can be used by application programmers to programmatically implement CAE systems that use the necessary cellular automata as solvers. The CATlib software contains three subsets of system routines: for implementing a preprocessor, a simulator, and a postprocessor.

Introduction

Cellular automata models of computation were proposed in the 1940s [1], and later they began to be used for researches in different physical, biological and chemical processes [2]. Actually, there is a number of cellular automata simulation systems that make working with cellular automata much easier. However, as will be shown below, most of them cannot provide the necessary functionality for thorough process investigations and do not support large cellular arrays. As a result, there is a need for a universal environment for developing software implementations of cellular automata with support of large-sized models.

This environment should meet the following criteria.

1. Simplicity of constructing cellular automata models and convenience of using them.
2. Extensibility, i.e. the system should support different kinds of cellular automata with various transition functions.
3. Platform independence, the ability of system to operate on various computer architectures and OS.
4. Scalability, i.e. as the size of the cellular array increases, the efficiency of using computing resources should remain acceptable.

In this paper, we will distinguish three functional levels of specialists involved in simulation — user, application programmer, and system programmer. The user launches a software product and solves the problem

*This work was carried out under state contract with ICMMG SB RAS (251-2021-0005).

of engineer modeling without modification of any program code. Application programmer develops simulation system and creates program code for this system using software tools, provided to him by the system programmer. These tools makes their job much easier. And finally, the system programmer develops these software tools for the application programmer.

The purpose of this work is to create a software tool having the listed properties, that is, can be used by the application programmer for developing cellular automata implementations.

Section 1 provides an overview of software products that are implementing cellular automata, but no single system is suitable for various reasons. In Section 2 the basic concepts and architecture of the CATlib software are given. In Section 3, the basic functions of the library are described. In Section 4, an example of program cellular automaton implementation obtained using the library is given.

1. Review of software products that implement cellular automata

The simulation ALT system (Animating Language Tools) is intended for experiments with fine-grained parallel algorithms [3]. The system consists of text and graphic tools combinations for visualizing the fine-grained parallel calculating. Simulation of processes comes with the parallel substitution algorithm [4]. For description the constructions of this algorithm in the ALT high level language is used, the prototype of which is C language. In the system there are also various tools for text and graphic editing of models and a number of tools for displaying parameters of calculating process during simulation. Main advantages of using this simulation system for research are ability of monitoring the calculating process and simplicity of creating various models and editing them. However, the ALT does not provide the opportunity of working with large amounts of data, the parallel launch of cellular automata using the system is also impossible. These days the ALT is outdated and not supported by developers, and its launch is only available on devices that are compatible with the DOS operating system.

The WinALT system for simulation of algorithms with fine-grained parallelism is continuation of the ALT system ideas [5, 6]. The system has two versions: console and graphic. Simulation is produced with special language that was developed for description of fine-grained parallel calculations in the system, and it also admits functions usage, created with C and C++ languages.

The WinALT takes over all the advantages the system ALT had. Besides, the system allows choosing the operating mode (synchronous, asynchronous, block synchronous), creating and editing simulating programs. The maximum sizes of the cellular arrays used have been slightly increased.

The system does not provide for specifying user topologies, and the set of topologies implemented in it is very poor. The system includes the ability of calculating on Windows clusters. Launch of the both of versions, console and graphic, is able only on devices with the Windows operating system. Nowadays the system is not supported by developers.

The Mirek's Cellebration (MCell) program was created for the purpose of studying out the existing models and creating new ones of one- and two-dimensional cellular automata [7]. The program supports 14 families of cellular automata. There is a possibility of constructing cellular automata with different types of neighborhoods: the Moore neighborhood, the von Neumann neighborhood, the Margolus neighborhood and the hexagonal neighborhood. In the program, for all of the families of cellular automata there is a number of built-in transition functions: from well-known and well-studied to those developed by the author himself. However, user can add transition functions on one's own with external libraries in .dll format. The presence of a graphical interface allows set cell states fast and easy, and also change these states during an evolution of the cellular automaton. MCell allows to study out the cellular automata with a cellular array size that is not exceeding 100000×2500 cells. There are also tools for collection of various statistics. The program does not allow calculating on the cluster, and actually is not supported by developers. It can be launched only on devices with the Windows operating system.

The Cafun (Cellular Automata Fun) application [8] is a tool for modeling the complex systems, such as social groups, alive organisms, natural processes and so on. It was created for searching general laws of complex systems in order to better understand their evolution, structure and behavior. The ability to make more accurate predictions has been announced as a practical outcome in economics, biology, physics and other scopes where the complexity matters. The concept of complex systems is based on three principles:

- Complex systems consist of many elements with individual properties and behavior.
- Elements' properties are the result of their local environment and their own history. Their behavior determined by a limited and locally available amount of information without any centralization.
- Interaction between elements happens at the same time. There is no predetermined sequence in which they occur.

The program uses original concept notation of cellular automata and object-oriented approach to description of cellular automaton rules. For its work the Java Runtime Environment is required. The program does not involve using on the distributed computing systems and actually is not supported by developers.

The Golly application is developed in order of studying out the behavior of various one-, two- and three-dimensional cellular automata [9]. Despite of the fact that the developers assert the application as an implementation of Conway's game of Life [10], Golly is such a strong tool for building other models. The application has the ability to implement various classes of cellular automata using both built-in software modules and scripts in Python and Lua. The application allows to set various topologies of cellular arrays. Effective using of memory ensures work on cell spaces of virtually unlimited size, provided that most cells are still empty, because of only cells that are not currently empty are processed. The Golly can be launched on the following operating systems: IOS, Android, Windows, Mac and Linux. For August of 2022 the system was supported by developers. It is not possible to perform calculations on a cluster using the application.

The Tiled CA program was developed for simulating using one- and two-dimensional cellular automata [11]. The program allows to split simulating area in different ways, but does not allow user to define a topology. The construction of cellular automata models is carried out using a graphical editor, which allows you to quickly and easily edit the shape and state of cells. The program has few possibilities for constructing new transition functions, since Tiled CA only supports the family of cellular automata of the Game of Life [10]. The program allows the user to set array sizes, but they ca not exceed some fixed value. Tools for expanding the abilities of building the cellular automata are not provided. There is no option to launch the program on a cluster. Actually the Tiled CA is not supporting by developers. Launching is only available on devices with the Windows operating system.

The CelLab web application was developed for studying out different processes using two-dimensional cellular automata [12]. Application allows to create cellular automata models by writing programs on Java and JavaScript languages, and also to display their evolution process. The program allows to set the transition function, the colors of cells depending on their states, to manage the process of displaying and to modify the transition function. Also the developers provide the CelLab Development Kit including an archive with a wide range of ready-to-run transition functions, that is, can simulate different physical, chemical and biological processes. However, there is no option to launch this application on a cluster and to collect statistics.

Considering each of these systems in the light of the criteria given in Introduction, you can see that only the first criterion is satisfied by all systems, most of them satisfy two or three criteria, and none of them satisfy all four. We conclude that these systems are convenient for implementing cellular automata, at least those that have simple topologies, but either produce inefficient code, are limited by the small size of the cellular array, or are unsuitable for running on distributed computing systems.

2. The library architecture

Computer-aided engineering (CAE) systems are usually implemented as the next three functional modules: preprocessor, solver and postprocessor. Using the preprocessor user sets initial and boundary conditions of the problem, including geometry, material parameters and media. The solver finds the solution to the problem and has almost no interaction with the user. In cellular automata methods, the solver is a simulator of the process under study. The postprocessor usually includes a viewer and various converters to present the modeling result to the user in understandable way. If application programmer follows this conventional architectural pattern due to developing CAE, the system programmer needs to provide three sets of system routines: for the preprocessor, for the simulator and for the postprocessor.

Cellular automata are functioning iteratively. The operating mode will be called the order of changing states of the array cells during one iteration [13]. In synchronous mode all of the array cells change their states simultaneously in accordance with the transition function. To implement synchronous mode on a serial computer, you need to select a duplicate cellular array, in which new states of cells will be recorded as they are sequentially traversed. In asynchronous mode, cells change their states randomly, which is realized when going through them sequentially in a random order. The simulator's system procedures ensure the application of the cellular automaton transition function, implemented by the application programmer as a software component, to each cell of the array repeatedly, in accordance with the number of iterations. Also they provide interaction between the cells with each other without the participation of the application software components. The simulator procedures turn out to be quite heavy due to the large number of repetitions, especially when processing the large-sized cellular arrays. Reducing the execution time of these procedures can be achieved by their parallel execution in systems with distributed memory.

The Cellular Automata Topologies Library (CATlib) represents as set of system routines created in C language, that can be used by the application programmer for program implementation of a CAE system that is embodying the required cellular automaton as a solver [14]. Following the conventional architectural pattern described above, the CATlib software contains three subsets of system routines: for preprocessor implementation, for simulator implementation and for postprocessor implementation.

The preprocessor transforms the initial conditions in the physical formulation into cell states, and also initializes a service structure that includes the cell array sizes and its topology, the memory size required to store the state of the cell, the operating mode of the cellular automaton and so on.

The following system routines for implementing the preprocessor are available for the application programmer.

`CAT_InitPreprocessor` receives information about topology, a type of model, a size of additional information, a cell size in bytes, a number of cells per meter and cell array's sizes; initializes the preprocessor neighborhood in the computer memory.

`CAT_PutCell` receives new state of a cell and its indices in the cellular array; records achieved state to the memory area corresponding to this cell.

`CAT_FinalizePreprocessor` receives an output file name; saves the user-specified parameters of the cellular automaton and the user-initialized state of the cellular array to the file in a special format defined by the library, frees the allocated memory.

The simulator iteratively applies transition function of the cellular automaton to all the array cells taking into account a given operating mode while ensuring interaction between neighboring cells according to the topology given.

The following system routines for implementing the simulator are available for the application programmer.

`CAT_InitSimulator` receives an input file name that is containing the parameters of a cellular automaton and a state of the cellular array; initializes the control structure and cellular array with data read from this file.

`CAT_Iterate` receives a pointer to the procedure in which the application programmer implements the cellular automatons transition function; applies the resulting function to the array cells, taking into account the operating mode that was specified during initialization, performing one iteration of the cellular automaton.

`CAT_FinalizeSimulator` receives an output file name; saves a resulting state of the cellular array, frees the allocated memory.

The postprocessor converts the states of the array cells obtained as a result of the operation of the cellular automaton into a format that can be read and used by the user for the further work.

The following system routines for implementing the postprocessor are available for the application programmer.

`CAT_InitPostprocessor` receives an input file name that is containing parameters of the cellular automaton and a cellular array's state; initializes the control structure and the cellular array with data read from this file.

`CAT_GetCell` receives cell indices in the cellular array; returns its state.

`CAT_FinalizePostprocessor` saves simulation results in user-readable formats and frees the allocated memory.

3. Main routines description

CAT_InitPreprocessor
Initialize the preprocessor environment.

Signature

```
int CAT_InitPreprocessor(int arrayTopology, int modelType,
        int cellSize, int globalSize, double cellsPerMeter,
        int arraySizeI, ...)
```

Input Parameters

`arrayTopology` — topology of the cellular array;

`modelType` — type of the model;

`cellSize` — size of value in a cell;

`globalSize` — size of extra info;

`cellsPerMeter` — the number of cells per meter;

`arraySizeI, ...` — sizes of array, number of sizes should be equal to the dimension.

Returns

0 in case of success;

−1 if error occurred when allocating file;

−2 if `arrayTopology` is wrong;

−3 if the number of arguments is wrong.

Attention: the number of array size arguments should be equal to the dimension; if it exceeds the dimension, extra arguments will be ignored; if it is less than the dimension, missing arguments will be equal to 1.

CAT_InitSimulator
Initialize the simulator environment header and the cellular array with information from file.

Signature

```
int CAT_InitSimulator(char *filename)
```

Input Parameters

`filename` — file path.

Returns

0 in case of success;

-1 if error occurred while allocating header or cellular array;

-2 if error occurred while opening file;

-3 if error occurred while reading from file;

-4 if file was damaged (control sum is not the same);

-5 if file name was empty.

CAT_InitPostprocessor
Initialize the postprocessor environment header and the cellular array with information from file.

Signature

```
int CAT_InitPostprocessor(char *filename)
```

Input Parameters

`filename` — file path.

Returns

0 in case of success;

-1 if error occurred while allocating header or cellular array;

-2 if error occurred while opening file;

-3 if error occurred while reading from file;

-4 if file was damaged (control sum is not the same);

-5 if file name was empty.

CAT_GetX, CAT_GetY, CAT_GetZ, CAT_GetT

Signature

```
double CAT_GetX(int i, ...)
double CAT_GetY(int i, ...)
double CAT_GetZ(int i, ...)
double CAT_GetT(int i, ...)
```

Input Parameters

`i, ...` — indices of the cell.

Returns

≥ 0 X, or Y, or Z, or T coordinate of the dot representing the cell;

-6 if topology is not implemented yet;

-7 if topology is not acceptable for this function;

-8 if environment or array were not initiated;

-9 if indices are out of bounds.

CAT_GetI, CAT_GetJ, CAT_GetK, CAT_GetL

Signature

```
int CAT_GetI(double x, ...)
int CAT_GetJ(double x, ...)
int CAT_GetK(double x, ...)
int CAT_GetL(double x, ...)
```

Input Parameters

`x, ...` — coordinates of the dot.

Returns

≥ 0 i, or j, or k, or l index of the cell representing the dot;

-6 if topology is not implemented yet;

-7 if topology is not acceptable for this function;

-8 if environment or array were not initiated;

-9 if indices are out of bounds.

CAT_GetMaxI, CAT_GetMaxJ, CAT_GetMaxK, CAT_GetMaxL

Signature

```
int CAT_GetMaxI()
int CAT_GetMaxJ()
int CAT_GetMaxK()
int CAT_GetMaxL()
```

Returns

≥ 0 maximum i, or j, or k, or l index;

-8 if environment or array were not initiated.

CAT_PutCell
Puts a value in a cell.

Signature

```
int CAT_PutCell(char *cellValue, int i, ...)
```

Input Parameters

`cellValue` — value to put in the cell;

`i, ...` — indices of the cell.

Returns

0 in case of success;

−8 if environment or array were not initiated;

−9 if indices are out of bounds.

Attention: the number of index arguments should be equal to the dimension; if it exceeds the dimension, extra arguments will be ignored; if it is less than the dimension, missing arguments will be equal to 0.

CAT_GetCell
Gets a value from a cell.

Signature

```
int CAT_GetCell(char *cellValue, int i, ...)
```

Input Parameters

`cellValue` — output variable to put the value from the cell in;

`i, ...` — indices of the cell.

Returns

0 in case of success;

−8 if environment or array were not initiated;

−9 if indices are out of bounds.

Attention: the number of index arguments should be equal to the dimension; if it exceeds the dimension, extra arguments will be ignored; if it is less than the dimension, missing arguments will be equal to 0.

CAT_SquareDistance
Calculates a square distance between dots representing two cells.

Signature

```
double CAT_SquareDistance(int i1, ..., int i2, ...)
```

Input Parameters

`i1, ...` — indices of the first cell;

`i2, ...` — indices of the second cell.

Returns

≥ 0 the square distance;

-6 if topology is not implemented yet;

-7 if topology is not acceptable for this function;

-8 if environment or array were not initiated;

-9 if indices are out of bounds.

CAT_Iterate
Applies one iteration on the cellular array.

Signature

```
int CAT_Iterate(int (* cellTransition)(char*))
```

Input Parameters

`cellTransition` — a function applied on the cellular array. It should return 0 in case of success.

Returns

0 in case of success;

$\neq 0$ error code returned from `cellTransition` function.

CAT_IsArrayChanged

Signature

```
uint64_t CAT_IsArrayChanged()
```

Returns the number of changes (in bytes) made in the last iteration.

CAT_GetNumThreads

Signature

```
int CAT_GetNumThreads()
```

Returns the actual number of threads used in a parallel region or the default number of threads to be used for subsequent parallel regions.

CAT_SetNumThreads
Sets the default number of threads to be used for subsequent parallel regions.

Signature

```
int CAT_SetNumThreads(int numThreads)
```

Input Parameters

`numThreads` — the number of threads to set.

Returns

0 in case of success;

-1 if number of threads is out of bounds.

CAT_FinalizePreprocessor, CAT_FinalizeSimulator, CAT_FinalizePostprocessor
Saves environment header and cellular array into file and clears the memory.

Signature

```
int CAT_FinalizePreprocessor(char *filename)
int CAT_FinalizeSimulator(char *filename)
int CAT_FinalizePostprocessor(char *filename)
```

Input Parameters

`filename` — file path.

Returns

0 in case of success;

1 if error occurred during opening file;

2 if error occurred during writing to the file.

4. Example of program implementation

As an example of using the library, we give a program implementation of the simulator of the cellular automaton Game of Life. This automaton uses a two-dimensional cellular array. The topology is a square with eight neighbors (the Moore neighborhood). Every cell can be in one of the two states: $s = 0$—dead cell and $s = 1$—alive cell. The transition function takes the value of 1 in two cases:

- the cell is alive and has 2 or 3 alive neighbors;
- the cell is dead and has 3 alive neighbors.

In other cases, the cell state takes the value of 0.

```
#include <stdio.h>
#include "catlib.h"

const int neighborsNumber = 8;
const int iterationsNumber = 100;

int gameOfLife(void *n)
{
  int *cell = n;
  int sum = 0;
  for(int i = 1; i <= neighborsNumber; i++)
    sum += cell[i];
  if (cell[0] == 0 && sum == 3)
    cell[0] = 1;
  if (cell[0] == 1 && (sum < 2 || sum > 3))
    cell[0] = 0;
  return 0;
}

void main(int argc, char *argv[])
{
  CAT_InitSimulator("inputFileName", CAT_SYNC);
  for(int i = 0; i < iterationsNumber; i++)
    CAT_Iterate(gameOfLife);
  CAT_FinalizeSimulator("outputFileName");
}
```

As we can see from the listing, the application programmer needs to implement transition function of the right cellular automaton to create a fully functional simulator, in this case `gameOfLife()`.

The routine `CAT_InitSimulator()` allocates memory for a cellular array, loads the cellular array and all necessary data from the file named `"inputFileName"`, including the topology selected and prepared by the preprocessor.

The routine `CAT_Iterate()` searches for the values of the neighboring cells and applies the transition function of cellular automaton `gameOfLife()` to each cell in the synchronous mode. This procedure hides all the interactions between the cells from the application programmer.

The routine `CAT_FinalizeSimulator()` saves a result of the simulator into a file named `"outputFileName"` as a new cellular array. This file has the same format as the `"inputFileName"` and it can be used as for processing by the postprocessor and to continue the simulation by restarting the simulator.

Conclusion

The work formulates the criteria for software intended for the implementation of cellular automata. A review of such software was carried out and it was concluded that none of them satisfies all four criteria. The architecture of the CATlib software that meets these requirements and a list of its main routines are presented, and an example of the simulator using this library is given.

References

[1] Von Neumann J. The General and logical Theory of Automata // Cerebral mechanisms in behavior; the Hixon Symposium / L.A. Jeffress ed. — Wiley, 1951. — P. 1–41.

[2] Vanag V.K. Study of spatially extended dynamical systems using probabilistic cellular automata // Physics-Uspekhi. — 1999. — Vol. 42, No. 5. — P. 413–434 (In Russian).

[3] Pogudin Y., Bandman O. Simulating cellular computations with ALT. A tutorial // LNCS. — Springer, 1997. — Vol. 1277. — P. 424–435.

[4] Achasova S., Bandman O., Markova V., Piskunov S. Parallel Substitution Algorithm. Theory and Application. — World Scientific Publ., 1994.

[5] Piskunov S. WinALT — a simulation system for computations with spatial parallelism // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — Novosibirsk: NCC Publisher, 1997. — Iss. 6. — P. 71–85.

[6] Beletkov D., Ostapkevich M., Piskunov S., Zhileev I. WinALT, a Software Tool for Fine-Grain Algorithms and Structures Synthesis and Simulation // LNCS. — Springer, 1999. — Vol. 1662. — P. 491–496.

[7] Mirek's Cellebration, 1-D and 2-D Cellular Automata viewer, explorer and editor. — http://www.mirekw.com/ca/index.html (last accessed on August 1, 2022).

[8] Homeyer A. A Brief Introduction to Cafun. — https://cafun.de (last accessed on August 1, 2022).

[9] Golly Game of Life Home Page. — https://golly.sourceforge.io (last accessed on August 1, 2022).

[10] Gardner M. Mathematical games – the fantastic combinations of John Conway's new solitaire game "life" // Scientific American. — 1970. — Vol. 223, No. 4. — P. 120–123.

[11] Tiled CA. — http://linuxenvy.com/bprentice/TiledCA/TiledCA.html (last accessed on August 1, 2022).

[12] Cellular Automata Laboratory. — https://www.fourmilab.ch/cellab/manual (last accessed on August 1, 2022).

[13] Bandman O. Implementation of large-scale cellular automata models on multicore computers and clusters // Intern. Conf. on High Performance Computing and Simulation (HPCS), Helsinki, Finland. — 2013. — P. 304–310.

[14] Cellular Automata Topologies Library. — https://gitlab.ssd.sscc.ru/medvedev/catlib (last accessed on August 1, 2022).