Translations of Cellular Automata for Efficient Simulation

Jörg R. Weimar*
Institute of Scientific Computing,
Technical University Braunschweig,
D-38092 Braunschweig, Germany

Cellular automata can be described in many different ways, one of which is to use a special purpose description language. Here, the language CDL is used as the source for translations into Java or C code for computer simulations. Several coding styles are generated automatically: The state transition function can be coded as Java, as C with stubs for integration into a Java simulation environment, as a lookup table, or as Java code consisting of boolean functions which allow the parallel simulation of 32 or 64 cells on one processor. The coding styles are compared for several examples and it is found that the boolean function style (also called multispin-coding) realized in Java is often, but not always, significantly more efficient than even native C code.

1. Introduction

Since John von Neumann invented the concept of cellular automata (CA) more than 50 years ago [1], many software systems have been written for simulating them. Most of these were created to simulate one specific CA, but many are capable of simulating a large class of CA. An overview of some of these programs can be found in [2]. Here, we describe a subsystem of the CA simulation environment JCASim [3–5], which is implemented purely in Java to provide the best portability. In this system, CA can be specified in Java, *cellang* [6], CDL [7], or a CDL-related XML dialect CAXL [8]. Descriptions in these special-purpose languages are translated into Java for efficient execution in the simulation environment. In this translation, several different coding styles can be used, which severely impact the efficiency of the resulting simulation. We describe these different translation options and measure the efficiency of the resulting code for several examples. Only the speed of the evolution of a CA for many times steps is used as a criterion, since initialization is done only once, and the speed of display depends very much on the operating system, and should be considered separately.

*Electronic mail address: J.Weimar@tu-bs.de.

Complex Systems, **14** (2003) 175–199;

https://doi.org/10.25088/ComplexSystems.14.2.175

This paper is organized as follows. First we describe the simple translation of CDL code into Java. We discuss a number of coding options that are considered good practice in object-oriented programming, but carry severe performance penalties. Section 3 describes the possibility of using a look-up table for the state transition rule. In section 4, this table is translated into a set of logical functions. This in itself does not lead to a performance improvement, but if it is combined with a packed coding, where the bits needed to store many cells are packed into one integer, this option can lead to a dramatic time saving. Section 5 describes how performance can be improved by translation into C and native code. This eliminates the overhead of current Java programming environments for calculating the state transition function, while retaining portability and flexibility for the remaining portions of the code. Section 6 then describes measurements to compare the generated code using the different coding styles and demonstrates that the logic coding approach is significantly more efficient than the other approaches, when it is applicable, and generates reasonably small boolean functions.

2. Translating CDL into Java

The language CDL is a Pascal-like special-purpose language for the description of CA [7, 9, 10]. It contains language constructs for the definition of a structured cell type, for defining the state transition function using normal imperative programming language constructs, and some special constructs to facilitate working with neighborhood cells.

The description of a CA in CDL can be translated fairly directly into Java, since most of its language constructs have direct equivalents in Java. These are: arithmetic and logical operators, assignment, if-statement, blocks, and case-statements. Most primitive data types can also be translated into Java primitive types, but Java does not have primitive enumeration or range types. The range types of CDL can be translated into suitable primitive integer types. The enumeration types can be translated into integers as well, or into a type-safe system of subclasses, which however is less memory-efficient. As an example, the following translation is shown. More Java code than shown here is generated for initialization and display.

CDL

```
cellular automaton T4;

type celltype = record
    a: boolean;
    b: 0..3;
end;

rule begin
    *[0].a := ( *[-1].a AND *[0].a ) XOR *[1].a;
```

```
    if  *[0].a AND *[-1].a AND *[1].a then
        *[0].b := ( (*[1].b) + (*[0].b DIV 2) ) MOD 4
    else
        *[0].b := *[0].b;
end;
```

Java

```
public class T4 extends State {
    protected celltype mystate;
    protected class celltype implements Serializable {
        boolean a;
        /*0..3*/ int b;
    }

    public T4(){
        mystate = new celltype();
    }
    /** The state transition function
    */
    public void transition(Cell cell){
        final State [] neighbors = cell.getNeighbors();
        {
            mystate.a = (((T4)neighbors[1]).mystate.a
                  && ((T4)neighbors[0]).mystate.a)
                   ^ ((T4)neighbors[2]).mystate.a;
            if (( (T4)neighbors[0]).mystate.a
              && ((T4)neighbors[1]).mystate.a
              && ((T4)neighbors[2]).mystate.a
            ){
              mystate.b =((((T4)neighbors[2]).mystate.b)
                        + (((T4)neighbors[0]).mystate.b/2))%4;
            }else{
                mystate.b = ((T4)neighbors[0]).mystate.b;
            };
        }
    }
}
```

2.1 Groups

One concept of CDL which does not exist in Java is that of a "group." This is similar to a vector in Java, with iterators defined to loop through all elements of the group (in CDL: `for`, `one`, `all`, `num`, and `sum`). There are two possible ways to translate this feature of CDL into Java. One way is to translate it into vectors and iterators. The second possibility exploits the fact that the group is (in our language definition) fixed; that is, all elements are known, and thus the action of iterators can be expanded explicitly into a sequence of statements (`for`) or expressions. Both possibilities are used in JCASim: The first is used for groups with the name "`neighbors`," which makes it possible for CDL to be written independently of the dimension and neighborhood of the lattice (selection of the neighborhood is done in the simulation environment).

https://doi.org/10.25088/ComplexSystems.14.2.175

The second possibility is used for all other cases, since it is more efficient for small groups.

As an example, the following CDL-code is translated into Java using the two different options.

CDL

```
cellular automaton TestGroup;
  type celltype = 0 .. 2;
  group neighbors = {*[0],*[-1],*[1]};
  var n : celltype;

rule begin
    for n in neighbors do
       if n > 0 then
           .....
end
```

Java with array

```
public void transition(Cell cell){
   int n;
   final State [] neighbors = cell.getNeighbors();
   for (int i=0; i<neighbors.length; i++){
      n = ((TestGroup)neighbors[i]).mystate;
      if (n > 0){
          .....
      }
   }
}
```

Java directly expanded

```
public void transition(Cell cell){
   final State [] neighbors = cell.getNeighbors();
   {
      if (((TestGroup)neighbors[0]).mystate > 0){
          .....
      };
      if (((TestGroup)neighbors[1]).mystate > 0){
          .....
      };
      if (((TestGroup)neighbors[2]).mystate > 0){
          .....
      };
   }
}
```

2.2 Statements in expressions

A small complication appears if we use the vector translation option for the expressions `one`, `num`, `all`, and `sum`. The Java code uses statements, but in CDL these are expressions, which can occur as part of a more complicated expression. Java, different from C in this respect, does not allow statements to occur inside expressions. One possible solution is to use anonymous inner classes, but it is more efficient to

generate temporary variables that are assigned a value directly before the expression in which they are used. Here is an example.

CDL

```
r := (1+num( n in neighbors: n = 1)) mod 3;
```

Java directly expanded

Here no difficulty appears, as the num-expression is expanded into another expression.

```
r = (1 + (
    (((TestGroup)neighbors[0]).mystate == 1?1:0)
   +(((TestGroup)neighbors[1]).mystate == 1?1:0)
   +(((TestGroup)neighbors[2]).mystate == 1?1:0)
  )) % 3;
```

Java with array

Here a temporary variable is used, since the num-expression expands into statements.

```
int temp1_=0;
for (int i=0; i<neighbors.length; i++){
    n = ((TestGroup)neighbors[i]).mystate;
    temp1_+= ( n == 1)?1:0;
}
r = (1 + temp1_) % 3;
```

Note that the translation system has detailed type information for determining the type of the temporary variable as boolean, integer, or float.

2.3 Record types

CDL record and union types are translated into inner classes in Java. The components of a record then become the member variables of the class. Variables (which in CDL are temporary objects used in the calculation of the transition function, but are not conserved from one iteration of the CA to the next) of such a record type would normally be translated into variables of the new class type, and be local to the transition function. In this case they need to be instantiated (created) at the beginning of the transition function. Unfortunately, creating objects is a very expensive operation in Java, therefore we have to avoid any object creation in the inner loop of the simulation. This can be done by re-using all objects, for example, by making record variables *static*, unless multithreaded simulation is intended.

2.4 Neighbor access

The access to neighboring cells, which in CDL uses a *relative* address, can be done through two different mechanisms. One possibility is to translate each access to a neighboring cell into a call of the

https://doi.org/10.25088/ComplexSystems.14.2.175

method "`cell.getNeighborRelative (dx,dy)`". This method must check the coordinates and access the lattice of cells to return the appropriate `State`-object. These operations involve a certain amount of overhead. Another possibility is to collect all neighboring states into an array once, and access them using this array. In JCASim this array is actually preserved since the neighborhood does not change between time steps. This option is called "cache neighborhood" in the JCASim system and can be turned on or off, since caching the neighborhood involves a cost in memory but improves performance if enough memory is available.

3. Table-lookup coding

A CA in the classical definition is a regular lattice of cells, each of which contains a state selected from a finite set of states. The cells are updated according to a state transition function depending on a finite number of neighbors of each cell. Since both the set of states and the neighborhood are finite, the number of different possible local configurations which can occur as inputs to the state transition function is also finite. If this number is small enough, we can store the result of the local state transition function in a table, and use this table in the simulation instead of the directly translated transition function.

The translation process proceeds along the following steps in the JCASim system.

- Determine the active neighborhood.
- Calculate the size of the input configuration.
- Generate masks and shifts for the neighbor components.
- Fill the table.
- Generate Java code to use the table.

3.1 Active neighborhood determination

First, the system collects the list of variables used as inputs to the state transition function. These are just those components of the state of the cell and of its neighbors, which appear as read-accesses in the transition function. In many cases the relevant configuration does not contain all parts of the state for all neighbors, but just one or a few components of a neighbor's state.

3.2 Calculation of the input configuration size

From the list of accessed (neighboring) state variables the size of the input configuration is determined: For each distinct part, the number of

bits used to store this state component are calculated, and these numbers are added for all parts of the active neighborhood. If the result is at most 22 bits, this means that the look-up table has at most $2^{22} = 4194304$ entries, which makes it feasible to store the entire table, and also makes it possible to store the entire state of a cell in one integer variable.

3.3 Generation of masks and shifts

In order to use a look-up table, a method must be found to calculate an index into the table (address) from the local configuration. For the calculation, each active neighborhood component is assigned a bit-mask and a shift for composition into an address. For the example T4 from section 2, the following properties are calculated.

name	size (bits)	mask	shift1	shift2
`*[ 0].a`	1	1	0	0
`*[ 0].b`	2	6	1	0
`*[-1].a`	1	1	–	3
`*[ 1].a`	1	1	–	4
`*[ 1].b`	2	6	–	4

Here shift1 is the bit position of this component in the integer representation of the state. shift2 is the shift needed to place the masked bit pattern into the address for access to the look-up table. The seven-bit address into the table encodes the active neighborhood as follows.

⋯	`*[1].b`		`*[1].a`	`*[-1].a`	`*[0].b`		`*[0].a`
⋯	6	5	4	3	2	1	0

The address is calculated as the disjunction of expressions of the form `(name & mask) << shift2` for each component of the active neighborhood, in this example:

```
int adr = (*[0] & 7)
    | ((*[-1] & 1)<<3)
    | ((*[ 1] & 1)<<4)
    | ((*[ 1] & 6)<<4);
```

where the first line collects all relevant components of the cell itself, and the other lines refer to the neighbors.

3.4 Calculation of all table entries

To fill the look-up table, we use the state transition function translated into Java in the normal way. We set up a special `Cell` object which is given an index of the table and provides the appropriate states as neighbors. The transition-method of the original Java class is then called once to update the state. The resulting state is converted into an

https://doi.org/10.25088/ComplexSystems.14.2.175

integer (using the mask and local shifts calculated before) and stored in the table. Even though this process must be performed for each table entry, it takes at most a few seconds.

3.5 Java code generation

Finally, a new Java class is generated that uses the look-up table in the state transition function, but otherwise behaves exactly as before (e.g., for initialization and display). This is achieved by subclassing the previously translated state-class. The `transition`-method is overridden, and additional methods are provided for conversion between the representation using one integer and the representation using separate variables for the state components as in this example.

```
public class T4Table extends T4 {
protected int mys;

static final int table[] = {
 0,0,2,2,4,4,6,6,0,1,2,3,4,5,6,7
.......
,1,1,3,3,5,5,7,7,1,6,3,6,5,0,7,0
};

public void transition(Cell cell){
    final State [] neighbors = cell.getNeighbors();
    int adr = (mys & 7)
        | ((((T4Table)neighbors[1]).mys & 1)<<3)
        | ((((T4Table)neighbors[2]).mys & 1)<<4)
        | ((((T4Table)neighbors[2]).mys & 6)<<4);
    mystatetable = table[adr];
}
private void toOriginal(){
    mystate.a = (((mys & 1) >> 0)==1);
    mystate.b = (((mys & 6) >> 1));
}
private void toTable(){
    mys = ((mystate.a?1:0) << 0)
        | ((mystate.b) << 1) ;
}
.....
```

3.6 Limitations

The translation from CDL and Java code into the table form has a number of limitations. Most importantly, the size of the active neighborhood may be limited, since tables larger than about 2^{22} entries are impractical to generate and store. A second limitation is that probabilistic rules are difficult to handle. If only one binary probabilistic choice appears in the transition function, and this choice has a fixed probability (as opposed to a data-dependent probability), then this choice can be incorporated as an extra input bit in the transition table, and a bit with the appropriate statistics can be generated for each table-lookup operation.

This approach only extends to a few binary choices or one probabilistic choice with very few alternatives and no data dependency. Otherwise, the table would become prohibitively large.

Obviously, the table method is not applicable to cell states that contain floating-point numbers or integers without range limitation.

4. Coding as a logic function

The preceding section showed how the transition function can be translated into a lookup-table. In this section we describe a transformation of this table into a set of logical formulas. The table can be regarded as a function

$$f : [0 \ldots 2^k - 1] \mapsto [0 \ldots 2^n - 1] \tag{1}$$

$$f(x) = y \tag{2}$$

where k is the number of input bits, and n is the number of output bits (usually also the number of bits in the cell state). Alternatively, consider the binary representation of x and y:

$$x = \sum_{i=0}^{k-1} 2^i \, x_i \qquad y = \sum_{j=0}^{n-1} 2^j \, y_j \tag{3}$$

and decompose the function f into boolean functions

$$g_j(x_0, x_1, \ldots, x_{k-1}), \qquad j = 0, \ldots, n-1 \tag{4}$$

such that

$$f(x) = y = \sum_{j=0}^{n-1} 2^j \, g_j(x_0, x_1, \ldots, x_{k-1}). \tag{5}$$

We can now try to find a compact representation of the functions g_j in terms of the logical functions available on a computer, namely AND (`&`), OR (`|`), NOT (`~`), and XOR (`^`).

4.1 Logic minimization

The task here is to find a compact representation for a set of boolean functions that are given as a set of input values, for which the output (the jth bit in the table output) is 1 (true). There are several approaches to this problem. Older methods, such as the Quine–McCluskey procedure [11] or the approach used in the espresso software package [12] aimed at producing a minimal two-level representation, for example, as a disjunctive minimal form. In the case of computer code implementing the logic function, where operations are applied sequentially, a representation using more levels does not carry a performance penalty, and

https://doi.org/10.25088/ComplexSystems.14.2.175

is most likely more compact. Such a representation can be found by algorithms that manipulate a data structure called the binary decision diagram (BDD).

4.2 Compact representation of logic functions with binary decision diagrams

BDDs [13, 14] are a graph-based representation of boolean functions. A node in the BDD is labeled by a variable of the boolean function. The node represents the Shannon decomposition of a function with the cofactors f_{low} and f_{high} represented by the two child nodes v_{low} and v_{high} of node v. Using the conditional expression, node v labels with variable x_i represent `xi?f_h:f_l`. When implemented using logic functions it represents `(xi & f_h) | (~xi & f_l)`, where `f_h` and `f_l` are the expressions for nodes f_{high} and f_{low}.

Two terminal nodes labeled 0 and 1 represent the constant functions 0 and 1. The BDDs we consider here are ordered, that is, the variables occur at most once in each path and in the same order in all paths from the roots to the terminal nodes.

Figure 1 shows the BDD representation of a number of simple boolean functions. A BDD can be reduced by the following two operations.

- *Type 1*. If two subgraphs are isomorphic, they are identified.
- *Type 2*. If the two cofactors of a node are identical, that is, the *low* and the *high* child of the node are identical, it can be deleted and all references to it can be replaced by references to the child node.

Applying these two operations until a fixed point is reached leads to a *reduced* graph. For a given variable ordering, this graph is a unique representation of the boolean function. Efficient algorithms exist for the manipulation of such a BDD [14].

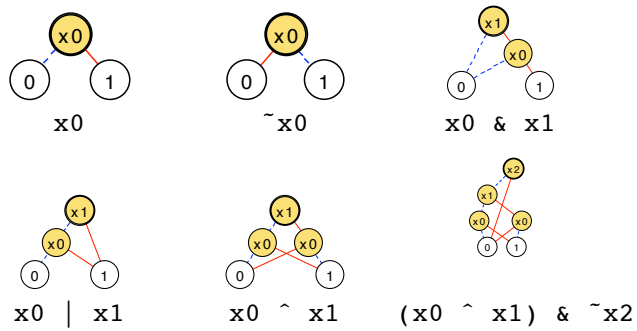

Figure 1. BDDs for simple boolean functions. The high edge is red (full line), the low edge is blue (dashed line). These BDDs have only one root, marked by a double circle.

4.2.1 Minimization of binary decision diagrams

The size of a BDD depends very much on the ordering of variables. Therefore one necessary optimization step is to find a good ordering which leads to a small number of nodes in the graph [14]. The task to find the best ordering has exponential running time, since there are $n!$ different orderings of n variables. An alternative is the sifting algorithm. In this approach the variables, which correspond to levels in the diagram, are ordered by the level size, which is the number of nodes labeled with each variable. Then the variable with the largest level size is shifted to all possible positions and the position which results in the smallest overall BDD size is kept. This operation is then repeated for the other variables. The algorithm only needs the exchange of neighboring levels as its basic operation. The disadvantage of this algorithm is that closely coupled variables can prevent the algorithm from finding a good ordering, since only one variable can be shifted at a time the other closely coupled variables can prevent it from moving away.

An alternative is the window permutation algorithm. Here a window size is selected (e.g., 4) and this window is shifted over the variables. Within the window all permutations of the variables are tested (window size 4: 24 permutations), and the best permutation is kept. The window is then shifted by one position and the process repeated. If during one pass of this algorithm an improvement occurred, the algorithm is repeated. The permutations are arranged in such a way that two successive permutations are related by an exchange of two neighboring variables. This is because the exchange of two variable levels in the BDD can be implemented efficiently, while implementing an arbitrary reordering of variables is very complicated.

Here are some examples for such permutation sequences.

- Window size 2: (0,1) - (1,0).
- Window size 3: (0,1,2) - (1,0,2) - (1,2,0) - (2,1,0) - (2,0,1) - (0,2,1).
- Window size 4: (0,1,2,3) - (1,0,2,3) - (1,2,0,3) - (1,2,3,0) - (2,1,3,0) - (2,1,0,3) ....

As a result, this algorithm leads to a simplified BDD in most cases with a reasonable computational effort. A practical example can be found in Figure 2.

4.2.2 Mapping into program code

Usually BDDs are used in the verification of logical circuits [15] (where the uniqueness of the representation is important) or in the synthesis of circuits. In this circumstance, the process of creating logical gates from the BDD is called *technology mapping*. In our case, we want to generate Java code with the logical operators available in the Java language, which are AND (`&`), OR (`|`), NOT (`~`), and XOR (`^`). The code should

https://doi.org/10.25088/ComplexSystems.14.2.175

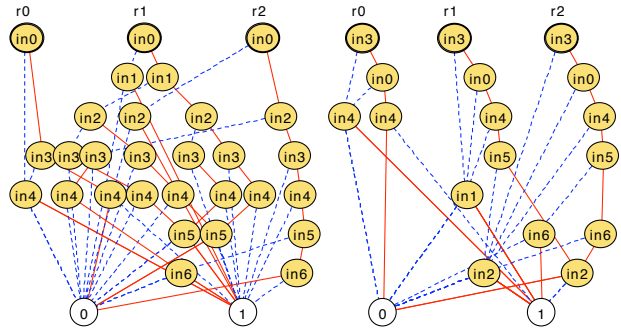

Figure 2. BDD for the example program T4. On the left, the initial variable ordering produces a BDD with 29 nodes, while the permutation search produces a variable ordering for which the BDD has only 17 nodes (right). The three roots correspond to the three bits of the output and common expressions are shared between the different functions.

store common subexpressions in temporary variables and generate one expression for each output bit of the multi-valued function. One possible approach is to directly translate the BDD according to the definition and generate code such as `(~xi & xi_low) | (xi & xi_high)` for node x_i with children $x_{i,\mathrm{low}}$ and $x_{i,\mathrm{high}}$, where `xi_low` would be the expression resulting from the translation of node $x_{i,\mathrm{low}}$. This approach can be simplified by recognizing common patterns and translating them directly into more efficient code. Figure 3 shows some patterns that are recognized. The following table contrasts the long version of all recognized patterns with the reduced code.

Long version	Short
`(~xi & 0) | (xi & 1)`	`xi`
`(~xi & 1) | (xi & 0)`	`~xi`
`(~xi & 0) | (xi & y)`	`xi & y`
`(~xi & y) | (xi & 1)`	`xi | y`
`(~xi & y) | (xi & 0)`	`~xi & y`
`(~xi & 1) | (xi & y)`	`~xi | y`
`(~xi & y) | (xi & ~y)`	`xi ^ y`
`(~xi & y) | (xi & ((~xk & y) | (xk & 0)))`	`y & (~xi | ~xk)`
`(~xi & y) | (xi & ((~xk & y) | (xk & 1)))`	`y | (xi & xk)`
`(~xi & y) | (xi & ((~xk & 0) | (xk & y)))`	`y & (~xi | xk)`
`(~xi & y) | (xi & ((~xk & 1) | (xk & y)))`	`y | (xi & ~xk)`
`(~xi & ((~xk & y) | (xk & 0))) | (xi & y)`	`y & (xi | ~xk)`
`(~xi & ((~xk & y) | (xk & 1))) | (xi & y)`	`y | (~xi & xk)`
`(~xi & ((~xk & 0) | (xk & y))) | (xi & y)`	`y & (xi | xk)`
`(~xi & ((~xk & 1) | (xk & y))) | (xi & y)`	`y | (~xi & ~xk)`

4.3 Storing a cellular automaton for logic coding

The logic representation of a transition table can be directly applied to the cell state as it is stored in the table update method. In this case

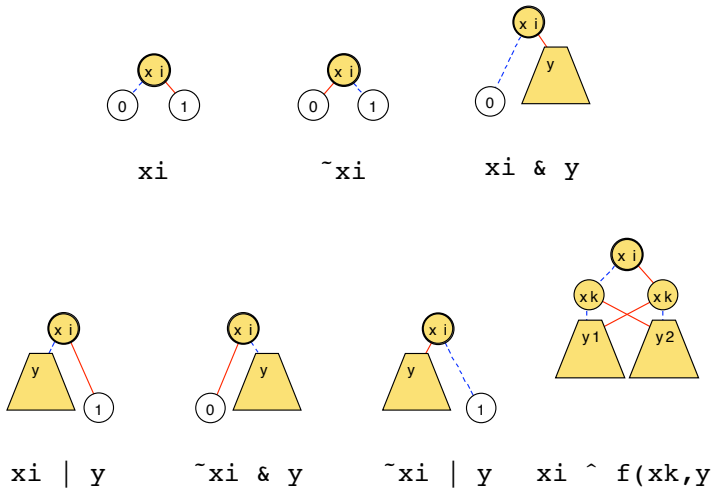

Figure 3. BDD patterns recognized in the translation process. The triangular shapes represent arbitrary subgraphs. Below each graph the translated form is shown.

the different input bits are extracted from the address calculated for the table.

```
int in0 = ((adr>>0)&1);
int in1 = ((adr>>1)&1);
.....
int in6 = ((adr>>6)&1);
```

Then the logic function is applied, which calculates the output bits `out0`, `out1`, ..., `out`n – 1, that are then combined to yield the new state

```
mystatetable = out0  | (out1<<1) | (out2<<2);
```

In this example the automatically produced logic function is:

```
 int t0 = ~in2;
 out0 =  ((~in3 & in4) | (in3 & (in0 ^ in4)));
 out1 =  ((~in3 & in1) | (in3 & ((~in0 & in1)
    | (in0 & ((~in4 & in1) | (in4 & (in5 ^ in2)))))));
 out2 =  ((~in3 & in2) | (in3 & ((~in0 & in2)
    | (in0 & ((~in4 & in2) | (in4 & ((~in5 & in6)
    | (in5 & (in6 ^ in2)))))))));
```

Manual simplification is much too complicated already in this small example.

Of course the resulting transition function is much more complex than a simple table lookup, and therefore is very inefficient. The only

https://doi.org/10.25088/ComplexSystems.14.2.175

case where it might be competitive is where the table is very large, memory access is very slow but the logic representation is short and logic operations are very fast.

4.4 Multispin storage

Fortunately, there exists an alternative storage scheme for which the logic coding is very efficient. Note that the logic function in the naive storage scheme above uses only one bit of the integers used for the calculation. But the logic operations used here have the property that they operate on all 32 or 64 bits of an integer in parallel and without interference between one bit and another (unlike operations such as + and *, which generate carry bits). We can use this property and store the information of 32 or 64 cells in one integer. Then one application of the logic function updates 32 or 64 cells in parallel, which more than offsets the inefficiency of the logic coding. This coding style is also called multispin coding and has been used extensively in the study of Ising spin systems [16–20], of the HPP lattice gas [21], and of traffic simulations [22, 23], but an automatic conversion to this coding style has to the author's knowledge not previously been described. A similar conversion appears in the preparation of a CA for simulation on programmable hardware, such as the CEPRA machines [9, 10, 24, 25], where much of the optimization is carried out by the (commercial) hardware synthesizer.

In the one-dimensional example automaton T4, the declaration for the memory reads

```
int numberOfLongs = (lx+63)/64;
long mysOld[numberOfLongs][k];
long mysNew[numberOfLongs][k];
```

where `k` is the number of bits needed for storing the cell state and `lx` is the size of the CA. Then the inputs for the transition function are

```
long in0 = mysOld[x][0];
long in1 = mysOld[x][1];
long in2 = mysOld[x][2];
long in3 = (mysOld[x][0]>>>1);
long in4 = (mysOld[x][0]<<1);
long in5 = (mysOld[x][1]<<1);
long in6 = (mysOld[x][2]<<1);
```

Here, `in0` ... `in2` are accessing the cell itself, while accesses to neighboring cells (in the first dimension) translate to a shift of the integer. Note that those cells on the border of a 32/64 cell block need special care for those neighbors which lie on adjacent blocks. Therefore the following code is added:

```
if (x < numberOfLongs-1){
    in4 |= ((mysOld[x+1][0]>>63)& 0x1L);
```

```
        in5 |= ((mysOld[x+1][1]>>63)& 0x1L);
        in6 |= ((mysOld[x+1][2]>>63)& 0x1L);
    }
    if (x > 0){
        in3 |= (mysOld[x-1][0]<<63);
    }
```

The coding style described here is only applicable on the level of a lattice, since it groups different cells together. This is realized in the simulation system by generating a special subclass of the `Lattice` class which implements the state transition function itself instead of calling the method `transition` of the `State` objects. This results in performance that for some rules is comparable to the performance of native C code, and is 10 times faster than any other Java coding style.

An alternative layout would be to pack cells together that are 64 sites distant from one another into one word. This is the approach used in most of the multispin-coding for Ising systems. It replaces the use of bit shifts to access the neighbors by more memory accesses. On modern CPUs with large register sets and memory caches, it is more efficient to use the shifts than to use more memory words.

The method as described so far is only applicable to deterministic CA. However, it is possible to extend it to probabilistic automata if the number of probabilistic choices is small. For a discussion of such approaches in the context of Ising systems see [16, 19].

5. Native C code

In the same way that Java code can be generated from CDL code, it is also possible to generate C code. The differences in the core transition function are actually very small, since C and Java use the same syntax for most expressions and statements. Nevertheless, it would take considerable effort to translate the complete simulation system into C. In addition, one would probably lose platform-independence. Therefore, a good interface between fast C code and a portable Java environment must be found. To place the interface at the individual cell level leads to very inefficient code, since the conversion of the cell state from Java to C and back takes more time than the actual calculation of the transition function. Therefore we use specialized `Lattice` classes which do the conversion once for all cells, then execute a number of time steps completely in C, and finally convert back to Java for displaying the data. The interface between C and Java uses the *Java Native Interface* [26]. The C code can be split into two parts: subroutines that are common to all translated native codes; and subroutines and data type declarations specific to the CA being simulated.

https://doi.org/10.25088/ComplexSystems.14.2.175

5.1 General code for native transition functions

The code for the use of transition functions in C consists of a Java-class `Lattice2DNative`, which has methods that are declared as native and implemented in C.

```
protected native void resetNative(int lx, int ly);
public native void beginBlockBorderNative();
public native void endBlockBorderNative();
public native void beginBlockNative();
public native void endBlockNative();
public native void transitionNative();
public native void backupNative();
```

A simulation proceeds through the following steps.

- `Reset` loads the native library and allocates data for all cells.
- `beginBlock` converts Java cells into C cells.
- `backup` copies *new* C cells into *old* C cells.
- `transition` executes one time step for each cell.
- `endBlock` converts C cells back to Java cells.

The additional methods `beginBlockBorder` and `endBlockBorder` are used to interconvert just those cells between Java and C which are located at the boundary, so that the different boundary handlers of the simulation system (which operate on Java data types) function correctly. Since the number of boundary cells is small compared to the total number of cells, converting these cells at each time step is feasible.

As an example of the general routines, we show the code for the conversion from Java to C cells, together with the relevant declarations. One can see that only the subroutine for converting a Java cell into a C cell must be specially coded for each CA. Note that the C array is larger than the Java array by `dist` cells on each side. These cells are used to store the neighboring cells (they are filled by `beginBlockBorder`) with those states that the appropriate `BoundaryHandler` delivers.

```
int lx, ly;           /* Size of the lattice  */
int dist;             /* boundary width       */
celltype **Cstate;    /* Array of C-cells     */
celltype **Cold;      /* old state of C-cells */
jobject **Jstate = 0; /* Array of Java-cells  */

/* Routine for conversion of one cell */
/* specific to each CA model. */
void java2c(JNIEnv *env, jobject jc, celltype *cc);

/* write Java fields into C fields.
*/
JNIEXPORT void JNICALL
```

```
Java_casim_Lattice2DNative_beginBlockNative
(JNIEnv *env, jobject this)
{
    int x,y,xx,yy;
    jobject Jstatel;
    celltype *Cstatel;

    if (Jstate == 0) collectStaterefs(env,this);

    for(x=dist; x<lx+dist; x++){
    for(y=dist; y<ly+dist; y++){
        java2c(env,Jstate[x][y],&(Cstate[x][y]));
    }}
}
```

5.2 Cellular automaton specific code

For each CA, some specific code is generated from the CDL description. This code includes declaring a structure for storing the cell content for access in the C code, conversion routines between C and Java, and backup and transition functions, which do most of the simulation work.

The example automaton T4 is translated into the following declarations.

```
#include <jni.h>
#include <Lattice2DNative.h>

typedef struct ct {
    char a;
    int b;
} celltype;

#include <Lattice2DNative.c>
static jfieldID fid_a, fid_b;
```

The conversion routines then use JNI-methods to access data in the Java cells.

```
void java2c(JNIEnv *env, jobject jc, celltype *cc){
    cc->a = (*env)->GetBooleanField(env, jc,fid_a);
    cc->b = (*env)->GetIntField(env, jc,fid_b);
}
```

The transition function makes a state transition for each cell of the lattice. Access to neighbor cells is translated directly into accesses of the neighboring elements of the d-dimensional array used to store the cells.

```
JNIEXPORT void JNICALL
Java_casim_Lattice2DNative_transitionNative
  (JNIEnv *env, jobject this)
{
  int x,y;
  for (x=1; x<lx-1+2; x++){
  for (y=1; y<ly-1+2; y++){
```

```
      Cstate[x+0][y+0].a =
        (Cold[x+-1][y+0].a && Cold[x+0][y+0].a)
        ^ Cold[x+1][y+0].a;
      if (Cold[x+0][y+0].a && Cold[x+-1][y+0].a
           && Cold[x+1][y+0].a
      ){
          Cstate[x+0][y+0].b =((Cold[x+1][y+0].b)+
                      (Cold[x+0][y+0].b / 2)) % 4;
      }else{
          Cstate[x+0][y+0].b = Cold[x+0][y+0].b;
      };
  }}
}
```

The result of this combined approach using general routines and CA specific code is that very little additional code needs to be generated for each CA. The resulting code is very efficient while retaining the full functionality of the JCASim simulation system. Of course, this efficiency can only be observed when the simulation runs over many time steps without needing to convert back to Java. Comparative measurements are presented in section 6.

A disadvantage of this approach is that platform-independence is reduced, since the C code must be compiled separately for each platform (but this is possible for all platforms on which Java is available). Nevertheless, if the library with the compiled C code is not available, the system automatically uses the Java code, which has the same functionality, but is somewhat slower.

6. Measurements

In order to demonstrate the different compilation options, we have measured the performance for a number of example CA. Table 1 summarizes some information on the different CA.

The different compilation methods are all automated in the translation from CDL and can be summarized as follows.

- *Normal*. The CDL source is translated into Java code, creating a subclass of `State`. This Java class is then compiled into byte-code using the standard `javac` compiler (with optimization). The byte-code is further compiled into machine code by the Just-In-Time (JIT) compiler of the Java virtual machine. The strictly object-oriented structure of the simulation system leads to the fact that each cell consists of a number of objects.

- *Lattice*. One optimization is to directly compile code that subclasses `Lattice2D` and includes a loop that runs over all cells. The updating of each cell is then performed directly on the data values of the cell, without calling methods of the objects comprising the cell. This approach violates the object-oriented philosophy, but leads to some improvements in speed. Since this translation is performed automatically, the turnaround-time is not significantly affected.

Name	Cell state	
	CDL	**Java**
MFTest42	-1..1	int
GH3	0..3	int
Bact2	2 boolean, 4× 0..10	2× boolean, 4× int
T4	boolean, 0..3	boolean, int
vonNeumann3	0..5, boolean, 2× -1..1, 0..6	boolean, 4× int
DiffAverage3	0..15	int
SpeedTest1	0..4	int

Name	Bits		Size		Function calls		Logic
	Cell	**Neigh**	**bytec**	**t**	**getN**	**rand**	**ops**
MFTest42	2	6	161	d	1	1	12
GH3	2	10	127	y	1	0	19
Bact2	18	90	618	n	1	4	
T4	3	7	163	y	1	0	37
vonNeumann3	10	50	2144	n	1	0	
DiffAverage3	4	12	79	d	1	1	5677
SpeedTest1	2	2	18	y	0	0	3

Table 1. Properties of the seven examples used for speed measurements. Column headings: **Bits Cell**, bits for storing the cell state; **Bits Neigh**, bits for storing the complete neighborhood (needed for table addressing); **Size bytec**, size of byte-code for the transition function, giving an indication of complexity; **t**, can be translated into table form (y: yes, d: deterministic, loose probabilistic part, n: no); **Function calls**, number of function calls; **getN**, getNeighbors(); **rand**, Random()/Prob(); **Logic ops**, number of logic operations in the logic coding style.

- *Table*. The conversion to table-lookup coding is described in section 3. The current implementation is limited to tables with less than about 22 inputs, which excludes the test automata "Bact2" and "vonNeumann3," which would need 90 inputs (about 10^{28} bytes) and 50 inputs (10^{16} bytes), respectively. In addition, the current implementation cannot handle probabilistic rules correctly. Instead, it uses a different probabilistic choice for each neighborhood configuration. For some automata, this can approximate the probabilistic rule, since different choices are stored in the table for similar configurations, which might occur with similar probabilities (see [27] and [28] for discussions on "error diffusion"). The table coding implies that the state of one cell is stored in one integer, which reduces memory used for storing cells. On the other hand, the size of the table increases memory usage. If the frequently used portion of the table exceeds the available cache size, performance degrades (see below).

- *Logic*. If the table is converted into a logic function operating on one cell (see section 4), the additional storage needed for the table is avoided.

- *LogicLattice*. The big advantage of logic coding is that one logic operation can operate on many bits at once. This advantage is realized by creating a subclass of `Lattice` which uses an array of `long` variables to store 64 cells each. The logic operations than operate on all 64 cells at a time. This bit-level parallelism offsets the speed disadvantage of the logic methods for many CA. This method is not useful if the cell state needs to be displayed regularly, since it must be converted back into the normal storage mode before display, and this is a fairly expensive operation. The logic coding method currently is based on the transition table, so it has the same limitations. The multispin coding method can handle very large lattices, since the memory is used very efficiently (almost all bits are used). In the measurements, we achieved the best speed with the largest lattices tested (500×500 cells).

- *Native*. Translation into native (C) code is described in section 5. In the current measurements, many time steps were used (on the order of 30000), so the time for the conversion between Java and C-storage modes, which is done once at the beginning and at the end, does not influence the measurements.

- *TableNative*. The table coding method can also be converted to native (C) code, using exactly the same table as in the Java version.

The speed of a CA simulation depends on a number of factors. We consider the most important ones and try to give estimates or measurements of their influence. Measurement results are shown in Table 2 and Figure 4 for the different coding styles applied to all test cases.

- *CPU speed*. The speed of the CPU and the processor architecture play an important role. We can estimate this influence by comparing simulation speeds of the same CA coded in C on different architectures. For the two architectures used in measurements here, a SGI O2 (200 MHz R10000 processor) and an Intel-based Linux system (700 MHz P-III), the difference is about a factor of 3 to 6.

Style	MFT	GH3	B2	T4	vN3	DA3	ST1
Normal	0.48	0.36	1.7	0.44	0.54	0.74	0.38
Lattice	0.24	0.12	1.4	0.20	0.27	0.53	0.11
Table	0.38	0.40	–	0.38	–	0.38	0.34
Logic	0.40	0.44	–	0.45	–	10	0.34
Multispin	0.007	0.008	–	0.010	–	0.35	0.005
Native	0.14	0.048	0.89	0.11	0.18	0.31	0.045
TNative	0.05	0.066	–	0.057	–	0.050	0.041

Table 2. Measured speed (in μ s/cell update) on the Intel platform with IBM JDK 1.3.0. Abbreviations: MFT: MFTest42, B2: Bact2, vN3: vonNeumann3, DA3: DiffAverage3, ST1: SpeedTest1. For comparison: one function call to Random() or getNeighbors() takes about $0.2..0.5\mu$ s.

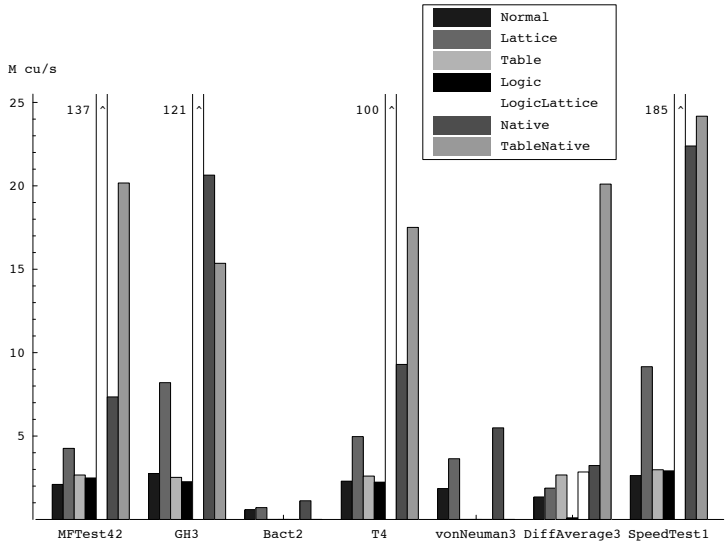

Figure 4. Speed (in millions of cell updates per second) comparison for different CA and different coding schemes. Measurements were made on a Linux system (800 Mhz, IBM JDK 1.3.0) and the result for the fastest CA size (between 30×100 and 1000×1000) was used.

- *Java VM*. The virtual machine and the JIT compiler used in the VM play an important role, but are difficult to judge. From the test we conclude that the difference between the two environments (SGI JDK 1.2.1 and IBM JDK 1.3.0) is about another factor of 2 (IBM Java 1.3.0 is 2 times faster) [29].

- *Memory usage*. All modern processors use one or several levels of cache to reduce memory latency. Both machines tested have a secondary cache size of one megabyte. This means that simulations which need less than a total of one megabyte during the cell updating phase fit completely in the cache. Bigger simulations lead to a complete replacement of the cache content during every time step. The amount of memory needed differs greatly between the different coding styles. For the simple Java style, this is around 200 bytes per cell, which means that simulations with less than about 50000 (or 50×100) cells run fastest. For the native coding methods, between 8 and 40 bytes per cell are used, and correspondingly bigger lattices are handled efficiently. The difference in speed between cache-based simulations and memory-based simulations is a factor of 2 for the SGI system and a factor of 4 for the Intel system (which is much faster overall).

- *Overhead*. During each simulation step some time is lost due to the special treatment of the borders, which leads to reduced performance (if

measured in cell updates per second) for smaller lattices compared to the biggest lattice that still fits into the cache. If a lattice size of 30×30 is compared to a lattice size of 100×100, we observe a difference of a factor of 2 for the native coding methods, up a factor of 7 to 20 for the LogicLattice coding, and only small differences (less than 15%) for the other Java methods.

- *Coding styles*. The speed differences between different coding styles depend very much on the memory usage pattern. Compared to the normal coding, we find the following differences.
  - *Lattice*. The Lattice coding is faster by a factor between 1.2 and 2.5 with a median around 2.
  - *Table*. The Table coding is only applicable to five out of the seven models, and neglects the probabilistic choices for two other models (see Table 1). Only in these cases does it give a performance advantage, in the other cases the speed is at most 10% faster than the normal coding.
  - *Logic*. The simple logic coding is about as fast as the normal coding, except for the "DiffAverage3" model with its 5677 logical operations, where it is slower by a factor of 6.
  - *LogicLattice*. The multispin coding which collects different cells together is the fastest option for most automata where it is applicable. For the automata with simple logic functions ("MFTest42," "GH3," "T4," and "SpeedTest1"), it is faster by a factor between 30 and 70.
  - *Native*. The native coding is faster than the Java coding by factors of 1.8 to 8 (SGI) or 5.7 to 15 (Linux). It is applicable to all CA coded in CDL, and treats probabilistic choices correctly.
  - *TableNative*. The table method coded in C is only slightly faster than the native method. It has the advantages of using a more compact storage (only one integer per cell) and through the fact that no random numbers are generated.

7. Conclusion

We have demonstrated how cellular automata (CA) descriptions coded in a language like CDL or *cellang* can be translated into different coding styles for use in a CA simulation environment. Direct translation of all language constructs into equivalent Java constructs is the easiest option. For some automata, a lookup table can be constructed, so that the transition function consists only of collecting the state of all cells in the neighborhood into an address and looking up the result in a table. This option turns out to be not more efficient than the simple translation. A huge performance gain can be achieved by using the multispin storage method, where each word contains 32 or 64 bits for storing part of the state of 32/64 cells. The transition function must be converted

into a boolean logic function, which can be done automatically. A conversion to native C code is not as efficient as this multispin coding, but by far outperforms the other possible Java coding styles. We have compared the different options on a number of examples and discussed the limitations of each approach. When CA are visualized at every time step, the visualization is the limiting factor, and the simple translation is adequate.

References

[1] J. von Neumann, *Theory of Self-Reproducing Automata* (University of Illinois Press, Urbana, IL, 1966, edited and completed by A. Burks).

[2] Thomas Worsch, "Simulation of Cellular Automata," *Future Generation Computer Systems*, **16**(2) (1999) 157–170.

[3] Uwe Freiwald and Jörg R. Weimar, "The Java Based Cellular Automata Simulation System—JCASim," *Future Generation Computing Systems*, **18**(7) (2002) 995–1004.

[4] Jörg R. Weimar, "Simulating Reaction-diffusion Cellular Automata with JCASim," in *Discrete Modelling and Discrete Algorithms in Continuum Mechanics*, edited by T. Sonar (Logos-Verlag, Berlin, 2001).

[5] Jörg R. Weimar, "JCASim," `www.jcasim.de`.

[6] J. Dana Eckart, "A Cellular Automata Simulation System," http://www.cs.runet.edu/~dana/ca/cellular.html, Radford University, 1995.

[7] C. Hochberger and R. Hoffmann, "CDL—A Language for Cellular Processing," in *Proceedings of the Second International Conference on Massively Parallel Computing Systems*, edited by Giacomo R. Sechi (IEEE, 1996).

[8] Michael Cohrs, "CAXL – eine XML basierte Beschreibungssprache für Zellularautomaten," Diplomarbeit, Institute of Scientific Computing, Technical University Braunschweig, 2001.

[9] C. Hochberger, R. Hoffmann, and S. Waldschmidt, "Compilation of CDL for Different Target Architecures," in *Parallel Computing Technologies*, edited by Viktor Malyshkin (Springer, Berlin, Heidelberg, 1995).

[10] Christian Hochberger, "CDL—Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen," Dissertation, TU Darmstadt, 1998.

[11] Edward McCluskey, *Introduction to the Theory of Switching Circuits* (McGraw-Hill, New York, 1965).

[12] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic Publishers, Boston, MA, 1984).

[13] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *Transactions on Computers*, **8**(35) (1986) 677–691.

[14] Rolf Drechsler and Bernd Becker, *Binary Decision Diagrams: Theory and Implementation* (Kluwer Academic Publishing, Dordrecht, New York, and London, 1998).

[15] Randal E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," in *IEEE/ACM International Conference on Computer Aided Design*, ICCAD, San Jose/CA (IEEE CS Press, Los Alamitos, 1995).

[16] R. Friedberg and J. E. Cameron, "Test of the Monte Carlo Method: Fast Simulation of a Small Ising Lattice," *The Journal of Chemical Physics*, **52**(12) (1970) 6049–6058.

[17] Laurence Jacobs and Claudio Rebbi, "Multi-spin Coding: A Very Efficient Technique for Monte-Carlo Simulations of Spin Systems," *Journal of Computational Physics*, **41** (1981) 203–210.

[18] R. Zorn, H. J. Herrmann, and C. Rebbi, "Tests of the Multi-spin-coding Technique in Monte Carlo Simulations of Statistical Systems," *Computer Physics Communications*, **23** (1981) 337–342.

[19] G. O. Williams and M. H. Kalos, "A New Multispin Coding Algorithm for Monte Carlo Simulation of the Ising Model," *Journal of Statistical Physics*, **37**(3/4) (1984) 283–299.

[20] Stephan Wansleben, John G. Zabolitzky, and Claus Kalle, "Monte Carlo Simulation of Ising Models by Multispin Coding on a Vector Computer," *Journal of Statistical Physics*, **37**(3/4) (1984) 271–282.

[21] S. Succi, "Cellular Automata Modeling on IBM 3090/VF," *Computer Physics Communications*, **47** (1987) 173–180.

[22] U. Brosa and D. Stauffer, "Vectorized Multisite Coding for Hydrodynamic Cellular Automata," *Journal of Statistical Physics*, **57**(1/2) (1989) 399–403.

[23] K. Nagel and A. Schleicher, "Microscopic Traffic Modeling on Parallel High Performance Computers," *Parallel Computing*, **20** (1994) 125–146.

[24] Rolf Hoffmann and Klaus-Peter Völkmann, "CEPRA-8: A Cellular Processing Machine," in *Massive Parallelism: Hardware, Software and Applications*, edited by M. Mango Furnari (World Scientific, Capri, 1994).

[25] Rolf Hoffmann, Klaus-Peter Völkmann, and Mark Sobolewski, "The Cellular Processing Machine CEPRA-8L," in *Proceedings of the Fourth International Workshop on Parallel Processing by Cellular Automata and Arrays*, edited by Jesshope, Jossifov, and Wilhelmi (Akademie Verlag, Berlin, 1994).

[26] "JNI—Java Native Interface," Sun Microsystems, `http://java.sun.com/`.

[27] Jörg R. Weimar, "Cellular Automata for Reaction-diffusion Systems," *Parallel Computing*, **23**(11) (1997) 1699–1715.

[28] Jörg R. Weimar, *Simulation with Cellular Automata* (Logos-Verlag, Berlin, 1998).

[29] Martyn Honeyford, "Weighing in on Java Native Compilation," *Developerworks*, IBM, 2002, http://www-106.ibm.com/developerworks/java/library/j-native.html,