CATEST Documentation (12/1/2008, ver.1.06d)

developed by Clemens Lode
clemens@lode.de
University Karlsruhe
for Thomas Worsch, Hidenosuke Nishio

CATEST is a program to determine injectivity and surjectivity of 1D-Cellular Automata of any configuration (2, 3, 4, 5 and 7 cell states, neighborhood size is limited by CPU speed, time and memory size), to simulate them and output the results into an image file.

Outline of this documentation

1. Code and Algorithms

The only code I used from elsewhere was the code for the TableSorter (TableSorter.java, by Philip Milne, Brendon McLean, Dan van Enckevort and Parwinder Sekhon, version 2.0 02/27/04). I'm using DOT (which is part of the graphviz package, see http://www.graphviz.org/) to draw the De Bruijn graphs, currentl mainly for debug issues of the core algorithm.

The basic idea, i.e. using De Bruijn automata / graphs as a representation of a CA rule in order to determine the injectivity and surjectivity of a cellular automata, is based on the work of Klaus Sutner (*Linear Cellular Automata and Finite Automata*).
According to Sutner the following three conditions are equivalent:

- Cellular automata p is surjective
- de Bruijn automata B(p) is balanced
- de Bruijn automata B(p) is unambigious

To test a de Bruijn automata for unambiguousness we need to construct a product automata and test it for strongly connected components. Iff the diagonal is the only strongly connected component then the CA is also injective.

2. Deployed algorithm for testing a graph for strongly connected components

In CATEST I'm using an optimized version of Tarjan's algorithm based on own ideas and some ideas from the paper "*On Finding the Strongly Connected Components in a Directed Graph*" by Esko Nuutila and Eljas Soisalon-Soininen.

The idea is to iterate through all nodes (in the worst case no nodes are connected) starting with the diagonal elements. If the diagonal elements are not strongly connected then we can cancel the calculation early.

Visiting a node is done by calling a recursive function ("visit") that will visit other, connected nodes. Because of this we ignore already visited nodes in the main loop.

In order to save time and memory we initialize the nodes and edges of the graph online during the run when they are visited. The speed-up occurs because in many cases we can cancel the calculation early.

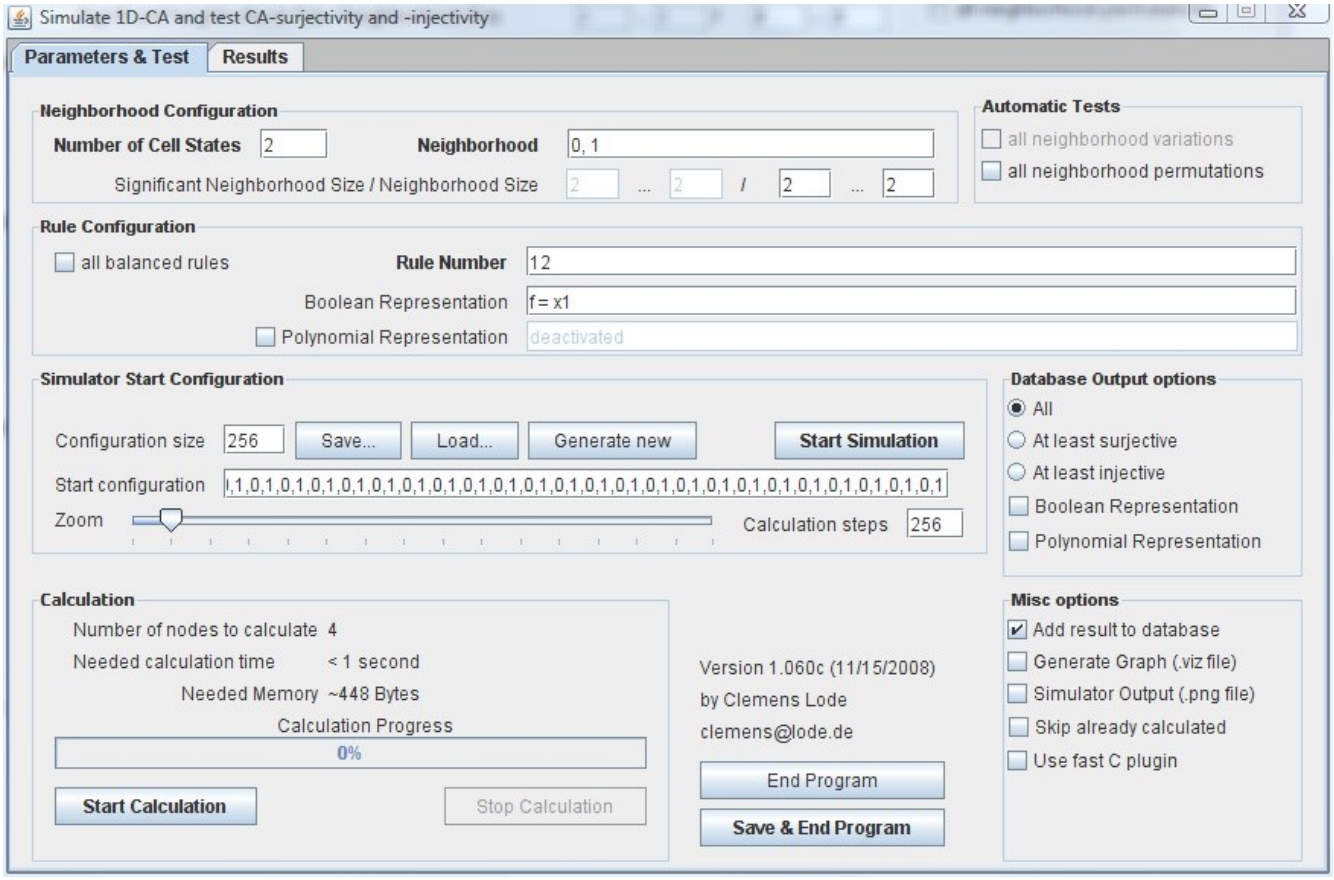
Each node we visit gets a unique ID, gets a secondary ID that marks its component number (all members of a component will have the same secondary ID at the end of the algorithm), is marked and is pushed on a stack (in order to trace back the connected graph later).

Then the up to <number of cell states>^2 connections are calculated and yet unvisited nodes are visited. If we hit a node that we already have visited we check if it is still marked (more about that later) and if the secondary ID is lower than ours. If that's the case we change the secondary ID to that lower value.
The interesting part comes when no more unvisited nodes are available, then we test if the nodes on the stack form a strongly connected component and return the result to our calling node. If it isn't strongly connected we can cancel the calculation for the whole set of nodes on the stack and move on to the next unvisited node.
If it is strongly connected we check again for a lower secondary ID of the node we just called and continue visiting unvisited nodes.

The final part is the test for strongly connected components on the stack. If the node in question isn't the root node (i.e. the secondary id differs from the id) we cancel and return true so that the calling node can continue to visit other unvisited connected nodes.
But if we reached the root node (secondary id == id) we go through the stack and count the number of elements on the diagonal.
If all elements of the diagonal were on the stack we know that the rule is surjective.
If only some elements of the diagonal were on the stack we know that the rule is neither surjective nor injective and we can cancel the calculation.
If we have established that the rule is surjective and we now find another strongly connected component we can cancel, too, because we know that the rule is surjective but not injective.
Only in the case where no other strongly connected component is found we can complete the calculation with the result that the rule is surjective and injective (worst case concerning time and memory consumption).

3. Description of the GUI Elements

3.1. Parameters & Test



3.1.1. Automatic Tests

All neighborhood variations
Selecting this option will let the program test all neighborhood variations. A neighborhood variation of the given neighborhood is any neighborhood that has the same smallest and largest value of the neighborhood (e.g. '0' and '6' for the neighborhood '0, 1, 2, 6') and has the same order of the values in between ('1' and '2' for the example above). For example, given a neighborhood size 6 and a significant neighborhood size 4, the neighborhoods (0, 1, 2, 5), (0, 1, 3, 5), (0, 1, 4, 5), (0, 2, 3, 5), (0, 2, 4, 5) and (0, 3, 4, 5) would be tested.
This option is disabled if the significant neighborhood size is only 2 or if no variation is possible (e.g. neighborhood '0,1,2,3').

All neighborhood permutations
Selecting this option will also test all permutations of all neighborhoods given by other options. E.g. if the neighborhood (0, 1, 2) was specified then the neighborhoods (0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1) and (2, 1, 0) would be tested.

3.1.2. Neighborhood Configuration

Numer of Cell States
The number of states of a cell.

Neighborhood
Neighborhoods can be entered using points (“.”), commas (“,”) or blanks as delimiter. The values have to be integers, they can be negative and can be entered in any order.
Internally the neighborhood will be standardized so that the first entry of the neighborhood is always 0, e.g. if you enter the neighborhood string “-2, -1, 4” it will be transformed to “0, 1, 6”. As we are testing only for surjectivity and injectivity adding an index to the neighborhood does not affect the result.
The neighborhood doesn't need to be in order, e.g. (0, -1, 3) will be processed properly.

Significant Neighborhood Size / Neighborhood Size
The maximum number *m* of cells relevant for the rule, i.e. a neighborhood “0, 1, 4” woud have neighborhood size 5, a neighborhood “0,1” would have neighborhood size 2. These fields control which neighborhood size ranges will be tested. If multiple neighborhood sizes should be tested then the **Neighborhood** will be filled automatically during the calculation. Specifying different values in each pair of fields will cause the program to test multiple neighborhoods. For example '3 ... 3 / 3 ... 5' will test neighborhoods (0,1,2), (0,1,3) and (0,1,4). If “**All neighborhood variations**” is activated then it would also test (0,2,3), (0,2,4) and (0,3,4).

3.1.3. Rule Configuration

All balanced rules
For each neighborhood configuration test all balanced rule numbers. For example, given the neighborhood (0,1), the rule numbers 3, 5, 6, 9, 10, 12 would be tested.

Wolfram / Boolean / Polynomial expression
All three fields are representations of the same rule, when one field is updated the other two will be updated appropriety. The boolean expression will only be shown for cases with two cell states. Entering a function will not affect the neighborhood size, e.g. with neighborhood size 2 and 2 cell states there will be an error if you enter “100” as rule number or “x0 + x1 + x2” as boolean representation. The first index of the variables is 0, i.e. x_0 denotes the first index. The index numbers are in ascending order and do not represent the neighborhood, i.e. “x0 + x2 + x3” with the neighborhood (0, 4, 5, 7) refers to positions 0, 5 and 7.

Boolean representation
Infix “ , ” : logical OR
Nothing or Infix “ * ” : logical AND
Postfix “ ' ” : negation of a single variable
Constants 1 and 0 can be used freely, spaces are ignored, the “f = “ expression at the beginning is optional, braces ( ) and other logical operators are not yet implemented.

Polynomial representation
“p” being the number of cell states
Infix “ + “ : addition with modulo p, in the case of multiple summands the modulo operator will be applied after all summands were added
Nothing or Infix “ * “ : multiplication with modulo p, in the case of multiple factors the modulo operator will be applied after all factors were multiplicated
Integral constants (positive and negative ones) can be used anywhere
Note that the transformation from Wolfram numbers / Boolean representations to Polynomial representation can take a long time, the algorithm still needs to be optimized.

3.1.4. Database Output options

All
All results will be put into the table in the 'Results' tab.

At least surjective
Only output results that are either surjective or surjective and injective.

Only injective
Only output results that are both surjective and injective

Boolean Representation
Calculates the boolean representation of each calculated configuration (needs additional time)

Polynomial Representation
Calculates the polynomial representation of each calculated configuration (needs additional time)

3.1.5. Misc options

Add result to database
All results fulfilling above conditions will be added to the database.

Generate Graph
A .viz file will be created for each result that is put into the database. The .viz file can be either converted manually to a .png file or automatically converted using the results tab, see below.

Simulator Output (.png file)
Generate a simulator output image together with each calculation.

Skip already calculated
Do not re-calculate those configurations that are already in the database.

Use fast C plugin
Calls the external C implementation of the core algorithm. It is about 100% faster and uses 50% less memory usage but has a calling overhead, maybe useful for few larger calculations (large neighborhood and/or large number of cell states)

3.1.6. Simulator Start Configuration

Configuration Size
Width of the cell space / output graph in cells

Start configuration
First line of the simulator

Zoom
Zoom factor of the output graph (width of each cell in pixels)

Calculation steps
Number of calculation steps (height of the output graph in cells)

Save... / Load...
Saves the current configuration to a file / Loads a configuration from a file (simple text file).

Generate new
Generates a new random start configuration

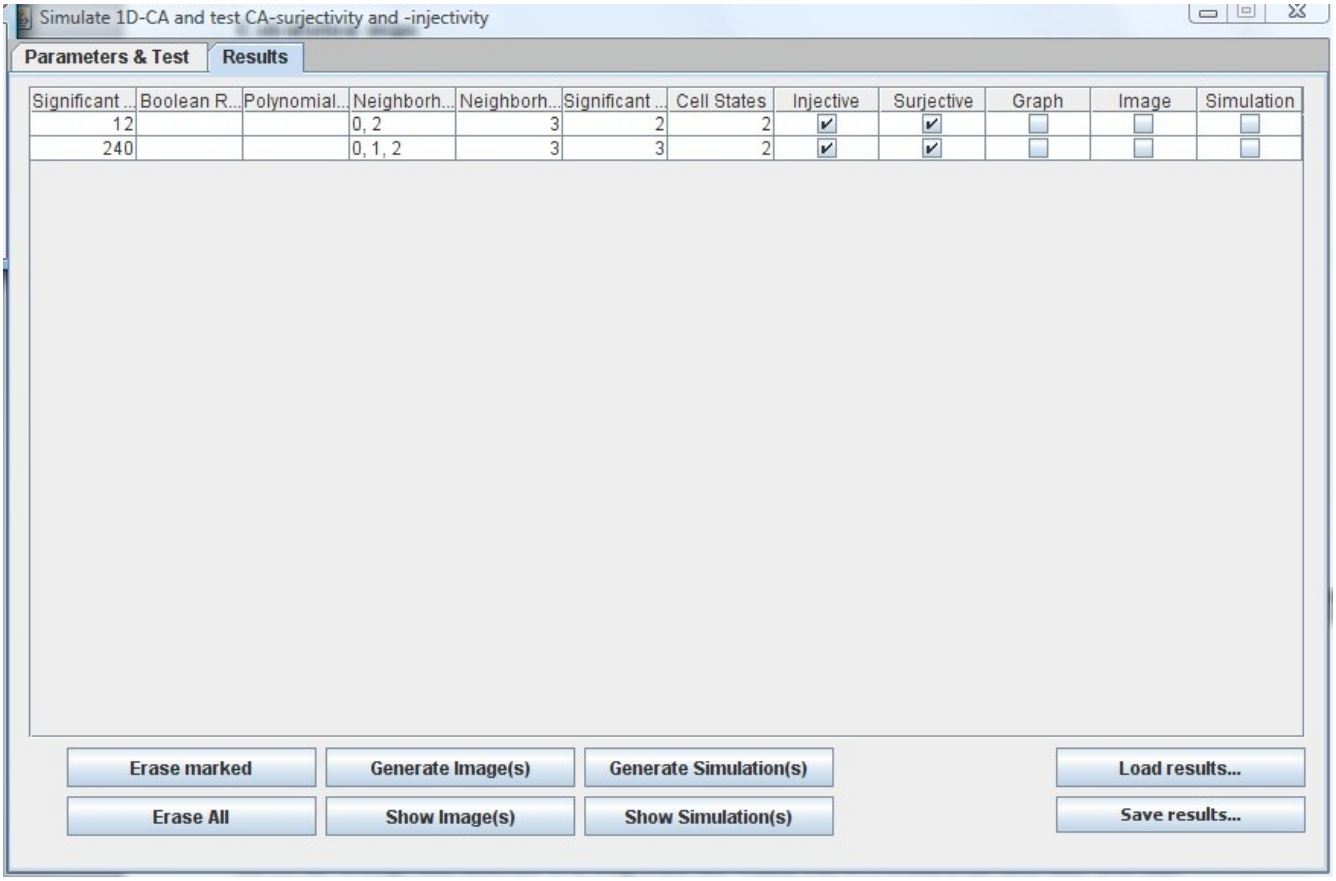
Start simulation
Starts a simulation with the given parameters.

3.1.7 Other

End Program
Quit the program.

Save&End Program
Saves the settings and database to a temporary file which will be reloaded the next time you start CATEST so you can continue where you left the program. This does not work during a calculation.

3.2. Results



Erase marked / Erase All
Remove items from the database, either all you have selected or every single on in the database.

Generate Image(s)
If the test was run with the option “Generate Graph (.viz file)” you can create a .png file automatically with the DOT program (which is part of the archive). If you use linux you have to create a soft-link from the dot directory to your dot binary.

Show Image(s)
Show the png files (if available) of all marked entries.

Generate Simulation(s)
Generates the simulation with the current simulation configuration and save it as an image.

Show Simulation(s)
Shows the generated image of the simulation.

Load/Save result(s)...
Load or save the results from or to a separate raw text file.

Note: You can sort each column by clicking on the column name.

4. Permutations and variations

The main calculation consists of a number of nested loops. In order to understand what is going on I outline the loop:

```
do {
  do {
    do {
      do {
        do {

          *** Calculate surjectivity and injectivity with the current settings ***

        } while nextFunctionPermutation
      } while nextNeighborhoodPermutation
    } while nextNeighborhoodVariation
  } while nextSignificantNeighborhoodSize
} while nextNeighborhoodSize
```

nextFunctionPermutation: (Rule Configuration: “all balanced rules”)
Permutates the table of the rule creating a new (balanced) function,
e.g. rule 12 → 10 → 6 → 9 → 5 → 3 (1100 → 1010 → 0110 → 1001 → 0101 → 0011)

nextNeighborhoodPermutation: (Automatic Tests: “all neighborhood permutations”)
Permutates the neighborhood,
e.g. (0,1,2) → (0,2,1) → (1,0,2) → ...

nextNeighborhoodVariation (Automatic Tests: “all neighborhood variations”)
Cycles through all neighborhood variations,
e.g. (0,3,4,5) → (0,2,4,5) → (0,1,4,5) → (0,1,3,5) → (0, 1, 2, 5)

nextSignificantNeighborhoodSize (Neighborhood Configuration: “Significant Neighborhood Size / Neighborhood Size”)
Cycles through all maximum neighborhood sizes from the significant neighborhood size up to the neighborhood size,
e.g. 4 → 5 → 6 with the setting “significant neighborhood size = 4” and “neighborhood size = 6”

nextNeighborhoodSizePermutation (Neighborhood Configuration: “Significant Neighborhood Size / Neighborhood Size”)
Cycles through all significant neighborhood sizes up to the neighborhood size,
e.g. 2/2 → 2/3 → 3/3 → 2/4 → 3/4 → 4/4 with the setting “neighborhood size = 4”