Science & Technology Facilities Council

Technical Report
RAL-TR-2012-018

FLAME tutorial examples: Cellular Automata - the Game of Life

LS Chin, C Greenough, DJ Worth

November 2012

©2012 Science and Technology Facilities Council

Enquiries about copyright, reproduction and requests for additional copies of this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358- 6254

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

FLAME Tutorial Examples:
Cellular Automata - the Game of Life

LS Chin, C Greenough, DJ Worth

October 2012

Abstract

FLAME - the Flexible Large-scale Agent Modelling Environment - is a framework for developing agent-based models. FLAME has been developed in a collaboration between the Computer Science Department of the University of Sheffield and the Software Engineering Group of the STFC Rutherford Appleton Laboratory.

This report documents the FLAME implementation of the Cellular Automata program known as the the Game of Life. The Game of Life was invented by John Horton Conway and it has been used extensively in demonstrating and testing complex systems software.

Keywords: FLAME, agent-based modelling, cellular automata, game of life, tutorial example

Email: {`shawn.chin, christopher.greenough, david.worth`}`@stfc.ac.uk`

Reports can be obtained from: `http://epubs.stfc.ac.uk`

Software Engineering Group
Scientific Computing Department
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

This report describes the FLAME (The Flexible Large-scale Agent Modelling Environment) [1] implementation of the cellular automata problem known as the *Game of Life*. The Game of Life was invented by John Horton Conway [2, 3]. Most of the work of John Horton Conway, a mathematician at Gonville and Caius College of the University of Cambridge, has been in pure mathematics but he also enjoys what we will call recreational mathematics. The *Game of Life* is one of these outings into recreational mathematics.

2 A brief description of FLAME

FLAME is what it says - it is an environment for developing agent-based applications. FLAME is an agent-based applications generator. FLAME is one of the main outputs of a collaboration between the Computer Science Department at the University of Sheffield and the Software Engineering Group of the STFC Rutherford Appleton Laboratory. Although FLAME does not have many of the interactive features of systems such NetLogo [4] it has been design to provide easy access to the utility of agent-based modelling and to accommodation the simulation of very large agent populations by using parallel high performance computing.

FLAME develops the ideas of Kefalas *et al.* [5] which describes a formal basis for the development of an agent-based simulation framework using the concept of a communicating X-machine.

FLAME has an agent specification language, XMML (based on the XML standard), a set of tools to compile the specified agent-based systems into code using a set of standard templates and the potential to produce optimised code for efficient parallel processing. FLAME allows modellers to define their agent based systems and automatically generate efficient C code which can be compiled and executed on both serial and parallel systems. So the main elements of FLAME are: the XMML model definition, the functions files (containing C code) and the FLAME `xparser` with associated templates.

The modeller provides a description of his model and the functions that define to operations, communications and changes of state of the agent population and FLAME generates the applications program. Figure 1 shows the structure of the FLAME environment. The modeller provides two input files: the Model XMML and the agents functions. These are parsed by the `xparser` and the results combined with the `xparser`'s template library to generate the application. Full details of the theoretical background to FLAME and the X-Machine approach to agent-based modelling is given in other various reports and papers [6, 7, 1, 11].

So the basic characteristic of FLAME and its agents are those of activation (state changes) and communication (agent to agent). This communication between agents is implemented within FLAME as a set of *message boards* on which agents post messages (information) and from which agents can read the messages. There is one message board per message type and FLAME manages all the interactions with the message boards through a Message Board API. The use of simple read/write, single-type message boards allows FLAME to divide the agent population and their associated communications areas. This approach has allowed the implementation of both serial and parallel versions within the same program generator.

Using this approach the modeller can design a model that can be realised as a serial or a parallel program. Although for many models this niave approach might achive reasonable parallel performance there are many pitfalls. To gain reasonable parallel performance in a very complex model the modeller will need to be aware of the impact of his choices on the performance of the model.

As mentioned above FLAME takes two forms of modeller input: the XMML description of the model and the C code implementation of the state change functions. These both have a straightforward structures. The XMML has a set of predefined tags and the C code has access to a number of predefined and model specific macros and functions. We will describe these in the section on the implementation.

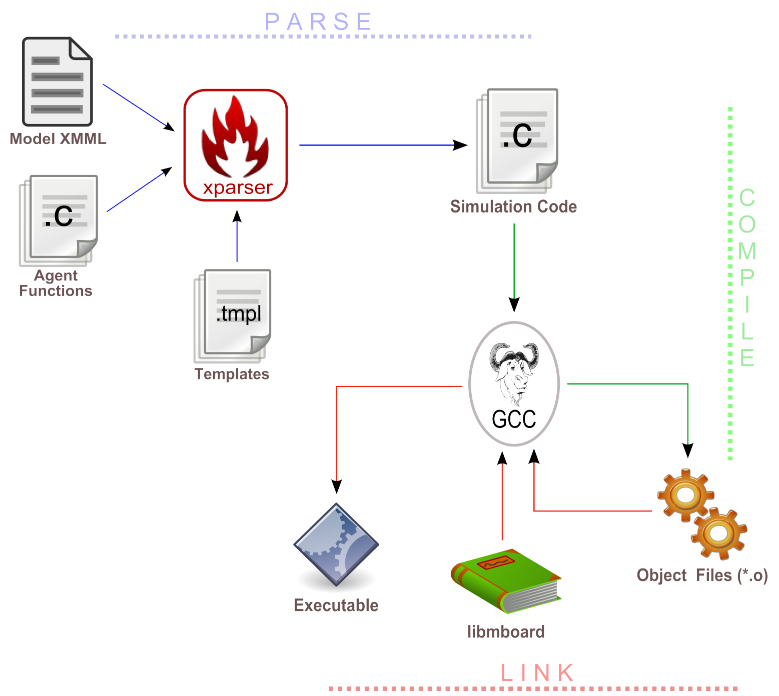


Figure 1: The structure of the FLAME Environment

3 Problem description

The *Game of Life* is an example of a two-dimensional cellular automaton. A cellular automaton is a computational machine that performs actions based on certain rules. The automaton domain can be thought of as a board which is divided into cells (such as square cells of a chessboard).

Each cell can be either *alive* or *dead*. This is called the "state" of the cell. According to specified rules, each cell will be alive or dead at the next time step. The rules of the game are as follows. Each cell checks the state of itself and its eight surrounding neighbours and then sets itself to either alive or dead. If there are less than two alive neighbours, then the cell dies. If there are more than three alive neighbours, the cell dies. If there are 2 alive neighbours, the cell remains in the state it is in. If there are exactly three alive neighbours, the cell becomes alive. These rules are applied in parallel and the simulation continues forever.

There are certain recurring shapes in Life, for example, the *stable* forms, the *gliders* and the *blinkers*. The glider will wiggle across the world, retaining its shape. A blinker is a block of three cells (either up and down or left and right) that rotates between horizontal and vertical orientations. These are example of the many shape classifications found in the Game of Life: *Still Lifes*, *Oscillators* and *Spaceships* are a few.

4 FLAME implementation

There are numerous different ways of implementing the Game of Life and we will explored one approach using FLAME. Initially one might think that the *agents* in the simulations should be the live *Cells* and the simulations should replicate the process of giving birth and dying according to Conways' rules.

However as the Cells do not move and their neighbours do not change, so a more straightforward approach is to make each Cell in the two-dimensional domain an *agent* with appropriate properties. It seems reasonable that each Cell knows its *state*, its position - (i, j) co-ordinates or the cell identifier and its 8 neighbours, Within FLAME all communications between agents is through *message boards* and not through some share background data structure. So for Cells to know the state of their neighbours information on cell states will need to be posted and read from a message board.

5 The FLAME model

We firstly define the various elements of the FLAME model. The model is divided into thee major groups of information:

environment - defines of global variables and C code file names

agents - all things related to the agents

messages - this things related to the model messages

A full listing the `CA.xml` model is given in Appendix A. Within each of these sections there are various tags to define the FLAME model. The overall structure is:

```
<xmodel>
   <environment/>
   <agents/>
   <messages/>
</xmodel>
```

Only the elements relate to the CA model will be highlighted below and the user should refer to the full FLAME User Manual for full details.

5.1 FLAME environment

The environment section contains various user defined constants and other essential system information. For the CA model only the domain size and the source code file name is required.

```
<!--FLAME Environment -->
<environment>
   <constants>
      <variable><type>int</type><name>DOMAINX</name></variable>
      <variable><type>int</type><name>DOMAINY</name></variable>
   </constants>
   <functionFiles>
      <file>functions.c</file>
   </functionFiles>
</environment>
```

The names `DOMAINX` and `DOMAINY` can be used within the user's C functions and $file$ is used by the FLAME parser to locate the function source code.

5.2 FLAME Agents

The start of an agent group is tagged by `agents` and each agent type definition is tagged by `<xagent>`. Within each agent definition are the specifications of its memory and its transition functions. So the overall structure is:

```
<agents>
   <xagent>
       <name>Cell</name>
       <memory/>
       <functions/>
   </xagent>
<agents>
```

5.2.1 Agent memory

As there are no pre-defined global data elements, apart from some limit global constants, within FLAME all *permanent* data must be held in an agent's memory.

Assuming each Cell is an agent they will need the following internal memory:

```
<memory>
    <variable><type>int</type><name>id</name></variable>
    <variable><type>int</type><name>i</name></variable>
    <variable><type>int</type><name>j</name></variable>
    <variable><type>int</type><name>state</name></variable>
    <variable><type>int</type><name>neighbours_state[8]</variable>
</memory>
```

We have give each agent an *id* so we can build a table of neighbours and defined a static array to hold the states of the cell's neighbours. Its cell position and state are held in i, j and *state*.

5.2.2 Agent functions

The number of functions or states the cell has is determined by the approach taken. We have decided that the agent will have states that perform only one action. So we define three functions or state transitions: *write_state* - writing the current cell's state to message board; *read_state* - reading state information from a message board and *react* - a function to applied the rules of life. The XMML function definitions follow.

write_state : this function write the current state of a cells to the *state* message board.

```
<|--Function write_state-->
<function><name>write_state</name>
    <currentState>start</currentState>
    <nextState>read_states</nextState>
    <outputs>
         <output><messageName>state</messageName></output>
    </outputs>
</function>
```

read_states : this function reads the information on the *state* message board and locates the states of the cell's neighbours.

```
<!-- Function read_states-->
<function><name>read_states</name>
     <currentState>read_states</currentState>
     <nextState>react</nextState>
     <inputs>
          <input><messageName>state</messageName></input>
     </inputs>
</function>
```

react : this functions determines the reaction of the Cell to the neighbour information - to live or die. The information is stored in the agent local memory so no messages need considering.

```
<!-- Function react-->
<function><name>react</name>
     <currentState>react</currentState>
     <nextState>end</nextState>
</function>
```

It would be possible to combine the *read_state* and *react* functions into a single function but the current division serves the purpose. The states of an agent are linked through the *currentState* and *nextState* tags in the XML. These tags, plus the *messageNames*, define the dependencies between and the ordering of the transformation functions.

5.3 FLAME messages

As mentioned above all agents communication via message boards. In this problem we define one message board - *state*. This provides all the information required by agent during the Game of Life. All message definitions are contained within the `<messages>` group.

```
<message>
    <name>state</name>
    <variables>
        <variable><type>int</type><name>id</name></variable>
        <variable><type>int</type><name>i</name></variable>
        <variable><type>int</type><name>j</name></variable>
        <variable><type>int</type><name>state</name></variable>
    </variables>
</message>
```

The complete XMML model is given in Appendix A

6 FLAME provided functions and macros

Before considering in detail the transitions functions of the CA model we will describe some of the basic facilities provided by FLAME. Transition functions can perform any operation and it is down to the modeller what they actually do. It the context of FLAME most agent functions will either read or write to agent memory or read or write to the model's message boards. FLAME provides a number of basic interfaces to help the modeller in these tasks.

Accessing environment data : FLAME provides an `environment` section in the model definition. This section can be used to define constants that can be used throughout the simulation code. In our example we have defined `DOMAINX` and `DOMAINY`. Once parsed these constants are defined in the file `header.h` as C macros and can be use in the same way.

Accessing agent memory : FLAME generates automatically a pair of memory access routines for all memory variables defined in the model. These are `get_` and `set_` in both cases they are postfixed by the memory variable name. For example for the memory variable `id` there will be the two functions `get_id` and `set_id`. These functions are context dependent as it is possible for more than one agent type to have a memory variable `id`. FLAME manages the use of these inbuilt functions so that the required memory variable is accessed. This is not an issue in the CA model as there is only one agent type.

Accessing message boards : For each message board type define in the model FLAME generates two important access mechanisms: one to write to message boards and another to read a message board. The elements of a message are defined in the model description and FLAME generates a simple function to write messages from this description. To write information to the *state* message board in the CA model FLAME provides the function:

```
add_state_message(int id, int i, int j, int state);
```

Generically this will be:

`add_`*message_board_name*`_message` (*variable list*)

All message boards defined in the model will have similar access functions.

Accesses message data : Accessing message data is a like more complex. In general an agent will wish to scan a message board looking for information of interest. FLAME provides a set of C macros that define and control a loop construct that will allow an agent to search a message board. In our example the only message board is `state` and it hold the following data *id*, *i*, *j* and *state*. For each message board FLAME provide two macros:

START_message_board_name_LOOP : Starts a loop structure to scan over message board `message_board_name` and sets up points to access the message data elements. The macro initialise a point - `message_board_name` - the message structure so that

```
message_board_name -> data_element
```

can be used to access elements of a message.

END_message_boards_name_LOOP : Terminates the message board loop.

7 Agent C functions

Associated with each agent state is C function that performs the change of state.

write_state : the *write_state* function writes the current state of a cell on the *state* message board. As we only interested in live agents only live agents write messages to the message board.

```
int write_state()
{
    int my_state, my_id, my_i, my_j;

    my_id = get_id();
    my_i = get_i();
    my_j = get_j();
    my_state = get_state();

    if (my_state == 1)
        add_state_message(my_id, my_i, my_j, my_state);

    return 0;
}
```

The function accesses agent memory and posts a message on the *state* message board. (The use of `my_` is a personal convention. Any local variable names can be used.)

read_states : the *read_states* function reads information posted by its neighbours on the message board.

```
int read_states()
{
    int my_id, my_i, my_j;
    int *my_neighbours;

    int mes_i, mes_j, mes_state;

    int count;

    my_id = get_id();
    my_i = get_i();
    my_j = get_j();
    my_neighbours = get_neighbours();

    for (count = 0; count <= 7; count++)
        my_neighbours[count] = 0;

    count = 0;
    START_STATE_MESSAGE_LOOP;
    if (state_message->id != my_id) {
        mes_i = state_message->i;
        mes_j = state_message->j;
        if (((mes_i == my_i - 1) && (mes_j == my_j - 1)) ||
            ((mes_i == my_i - 1) && (mes_j == my_j)) ||
            ((mes_i == my_i - 1) && (mes_j == my_j + 1)) ||
            ((mes_i == my_i) && (mes_j == my_j - 1)) ||
            ((mes_i == my_i) && (mes_j == my_j + 1)) ||
            ((mes_i == my_i + 1) && (mes_j == my_j - 1)) ||
            ((mes_i == my_i + 1) && (mes_j == my_j)) ||
            ((mes_i == my_i + 1) && (mes_j == my_j + 1))) {
            mes_state = state_message->state;
            my_neighbours[count] = mes_state;
            count++;
        }
    }
    FINISH_STATE_MESSAGE_LOOP;

    return 0;
}
```

The *read_states* function must scan all the messages on the *state* message board to select the agents neighbour state information. In this implement the whole message must be scanned and the appropriate information selected. The FLAME framework provides a set of simple macros. *read_states* need to scan the message boards to collect data from its neighbours. The macros `START_STATE_MESSAGE_LOOP` and `FINISH_STATE_MESSAGE_LOOP` provide loop over the message board *state*. Any information in the message are referenced through the `state_message-->` construct.

react : *react* function applies the current rule set to the agent's state.

```
int react()
{
    int my_id, my_i, my_j, my_state;
    int *my_neighbours;
    int i, count;

    my_id = get_id();
    my_i = get_i();
    my_j = get_j();
    my_state = get_state();
    my_neighbours = get_neighbours();

    count = 0;
    for (i = 0; i < 8; i++) {
        if (my_neighbours[i] == 1) count++;
    }

    if (count < 2)  my_state = 0;
    if (count == 2) my_state = my_state;
    if (count == 3) my_state = 1;
    if (count > 3)  my_state = 0;

    set_state(my_state);

    return 0;
}
```

8 Parsing the FLAME model

In the above section we have only given a description of the essential parts of the FLAME model. Appendix A and Appendix B give the complete model files. With the model define in XMML (CA.xml) and the agent functions written (functions.c) the complete model can now be parsed with the FLAME parser. The actual command will depend on the implementation and operating system. On a Linux system the following would be common:

```
beersheba% xparser
xparser (Version 0.16.2)
Usage: xparser [XMML file] [-s | -p] [-f]
        -s      Serial mode
        -p      Parallel mode
        -f      Final production mode
beersheba%
```

and the full output of the generation would be:

```
beersheba% xparser CA.xml
xparser (Version 0.16.2)
Environment variable FLAME_DIR not set - looking in current directory for Templates

Code type       : Serial (DEBUG)
Input XMML file : CA.xml
Model root dir  :
Template dir    : /home/cg/SANDBOX/FLAME/xparser/

Reading XMML file (CA.xml)
- Model name     : Game of Life
- Functions file : functions.c
- xagent   : Cell
- Message  : state
End of XMML file

Creating dependency graph
Finished dependency loop check
Total communication sync lengths = 1
Ordering functions in process layers
New communication sync lengths = 1

Writing file : stategraph.dot
Writing file : stategraph_colour.dot
Writing file : process_order_graph.dot
Writing file : latex.tex

Generating Makefile using /home/cg/SANDBOX/FLAME/xparser/Makefile.tmpl
Generating xml.c using /home/cg/SANDBOX/FLAME/xparser/xml.tmpl
Generating main.c using /home/cg/SANDBOX/FLAME/xparser/main.tmpl
Generating header.h using /home/cg/SANDBOX/FLAME/xparser/header.tmpl
Generating memory.c using /home/cg/SANDBOX/FLAME/xparser/memory.tmpl
Generating low_primes.h using /home/cg/SANDBOX/FLAME/xparser/low_primes.tmpl
Generating messageboards.c using /home/cg/SANDBOX/FLAME/xparser/messageboards.tmpl
Generating partitioning.c using /home/cg/SANDBOX/FLAME/xparser/partitioning.tmpl
Generating timing.c using /home/cg/SANDBOX/FLAME/xparser/timing.tmpl
Generating Doxyfile using /home/cg/SANDBOX/FLAME/xparser/Doxyfile.tmpl
Generating rules.c using /home/cg/SANDBOX/FLAME/xparser/rules.tmpl

Writing header file : Cell_agent_header.h

--- xparser finished ---
```

```
To compile and run the generated code, you will need:
 * libmboard (version 0.2.1 or newer)
beersheba
```

This provides some error diagnostics should the parser find errors in the model. The FLAME parser will generate the complete application and various additional files. These include:

Makefile - the Unix make

header.h, low_primes.h - system header files

Cell_agent_header.h - model specific header file

main.c, memory.c messageboards.c partitioning.c, rules.c, timing.c, xml.c - system C files.

process_order_graph.dot, stategraph_colour.dot, stategraph.dot - various graphical state diagrams

latex.tex, Doxyfile - documentation templates

The develop should not modify these files as they will be automatically overwritten next time the FLAME parser is run.

FLAME uses a task dependency graph to schedule the execution of agent functions and communications. One of the file generated by the FLAME parser is the task dependency graph. The graph shows all the agent and their functions in the model together with all the defined message board accesses. Figure 2 shows the dependency graph (`stategraph_colour.dot`) for the CA model using the *Graphviz* utility `dotty` [8].

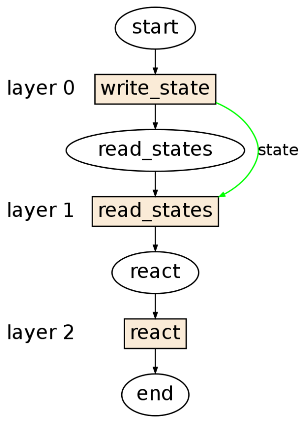


Figure 2: The *task dependency graph* of the CA model

The dependency graph provides a very good visual check on the structure of the model.

9 Input data generation

The initial data for the CA model is generated using a simple python program. The program can either generate a random distribution of life agents on a specified domain or generate an input based on a given live agent pattern.

```
Usage: init_start_state.py <width> <height>
                           <agent_count>|{<file_name> [<x-offset> <y-offset>]}
More info:
```

```
<width> (int) and <height> (int) will determine the size of cell space in which
life will be randomly placed in.
<agent_count> (int) specifies the number of live cells to initialise.
<file_name> (char) specifies a file of initial data in csv format.
<x-offset> <y-offset> (int - optional) allows the csv embedding to be offset.
```

The program will generate an `0.xml` output file containing the initial data.

10 Testing

There are various ways in which the *Game of Life* can be tested. It has been the subject of considerable interest. We will use a number, of the many, *Still lifes*, *Oscillators* and *Spaceships* to test our implementation.

The *pgplot* graphics library [10] has been used to generate the graphical outputs.

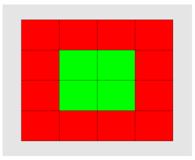
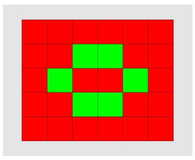
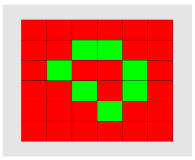
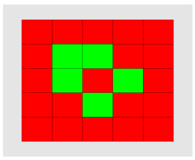
Block		Beehive	
Loaf		Boat	

Table 1: A selection of *Still lifes*

Table 1 show some basic *Still lifes*. These are unchanging configurations to which the simulation can move to. They clearly check many of the basic model operations.

The next set of test configurations are some *Oscillators*. These configurations oscillator two and from there base state via other configurations in a define period. The *Oscillators* show in Tabel 2 have periods of only one, two or three - there being only one, two or three intermediate states. Figure 3 show the life-cycle of the *Pulsar* configuration.

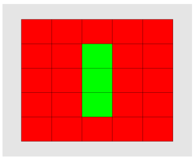
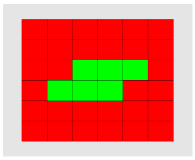
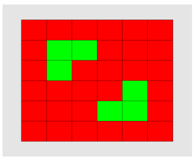
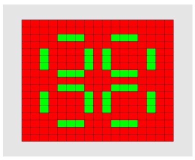
Blinker (period 2)		Toad (period 2)	
Beacon (period 2)		Pulsar (period 3)	

Table 2: A selection of *Oscillators*

The final group of test examples are two *Spaceships*. These configurations travel across the simulation space through a regular cycle of transformations. Table 3 shows the configuration of the *Glider* and *LWSS* (Lightwieght spaceship). Each of these *Spaceships* move through the computational domain through a characteristic set of transformations. Figures 4 and 5 show the life cycles of the *Glider* and *LWSS* spaceships.

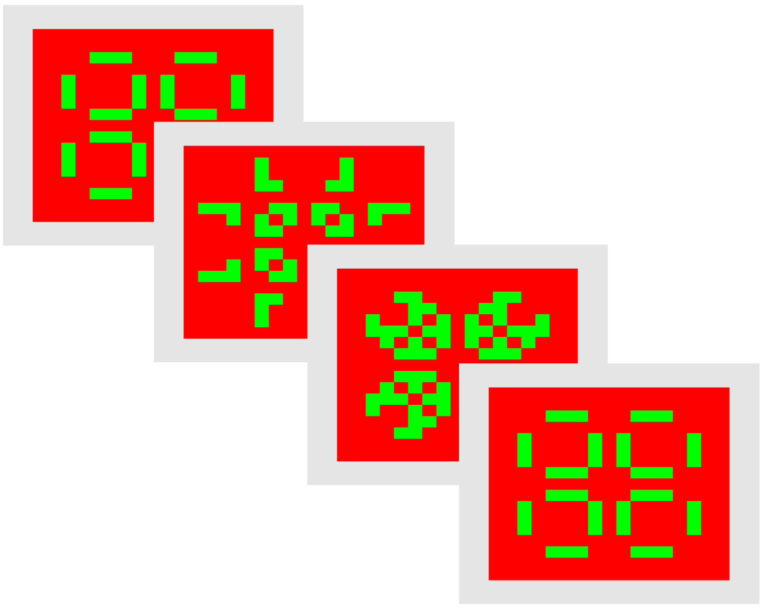


Figure 3: The *Pulsar* life-cycle

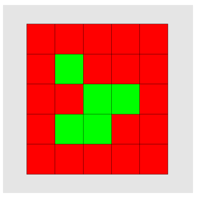
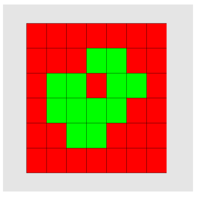
Glider (period 3)		LWSS (period 6)	

Table 3: A selection of *Spaceships*

11 Example results

In this final section we show the results from two larger simulations. The first is one of the so called *Glider Guns* - *Gosper's Glider Gun* [14]. *Guns* are configuration that repeatedly shoot out moving objects such as *Gliders*. The *Gosper's Glider Gun* was the first and is the smallest glider gun so far discovered.

The second example is a larger random life population of 900 live cells on a 50x50 cell grid.

11.1 The Gosper Glider Gun

In a cellular automaton, a gun is a pattern with a main part that repeats periodically, like an oscillator, and that also periodically emits spaceships. There are then two periods that may be considered: the period of the spaceship output, and the period of the gun itself, which is necessarily a multiple of the spaceship output's period. A gun whose period is larger than the period of the output is a pseudoperiod gun.

Since guns continually emit spaceships, the existence of guns in Life means that initial patterns with finite numbers of cells can eventually lead to configurations with limitless numbers of cells, something that John Conway himself originally did not believe was possible. Bill Gosper discovered the first glider gun (and, so far, the smallest one found) in 1970, earning $50 from Conway. Figure6 shows the initial configuration of the gun. The *Gosper glider gun* shown above produces its first glider on the 15th generation, and another glider every 30th generation from then on. Figure7 show the configuration after many iterations. The multiple *gliders* can be

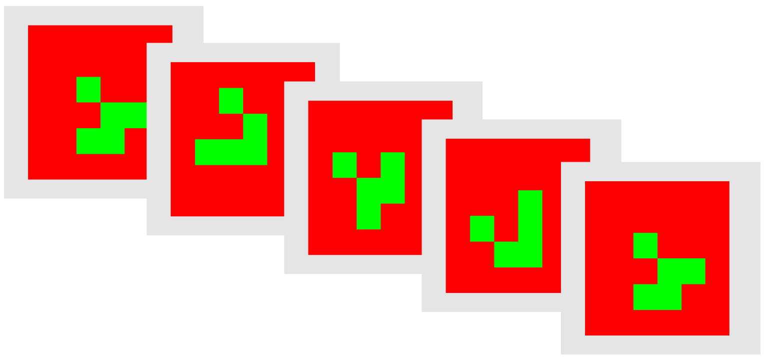


Figure 4: The *Glider* life-cycle

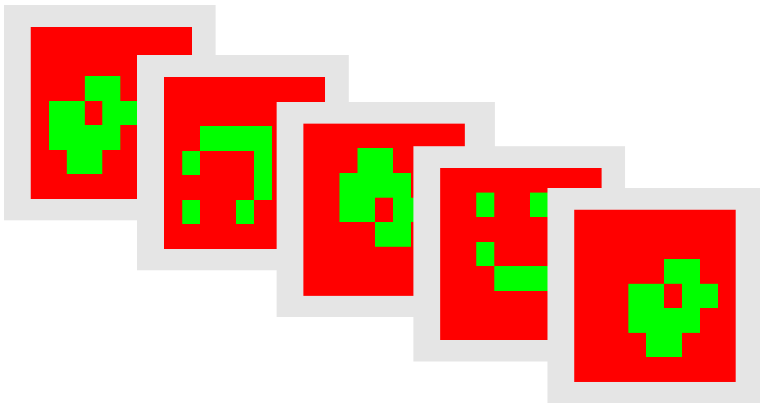


Figure 5: The *LWSS* life-cycle

clearly seen with the main pattern in its initial state.

11.2 Random initial distributions

The final example is a Life simulation started from a random distribution. We have taken a domain size 100x100 (10000) cells with an population of 3300 live cells. Figure 8 show the initial population of cells and Figure 9 show the population after 300 iterations after which the population of live cells has reduced to 857. By this time, as can bee seen in Figure 9 a large number of still and oscillator patterns have developed.

12 Comments on implementation

There are quite a few ways of implementing cellular automata using FLAME the approach taken here is one of the simplest. Although only *live* agents post messages on the state message board this could lead to quite large numbers in a large simulation. In the current implementation of FLAME there are no intrinsic functions to return information on the neighbours of an agent. Consequently all such searching and selection must be performed by the agent functions and therefore each agent must read all the contents of the message board.

When neighbour based filters are implemented in FLAME in this examples the number of messages scan could be reduced from many hundreds to a maximum of eight thus significant improving the performance of the simulation.

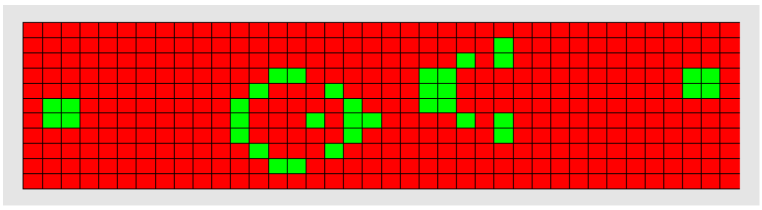


Figure 6: The initial state of the Gosper Glider Gun

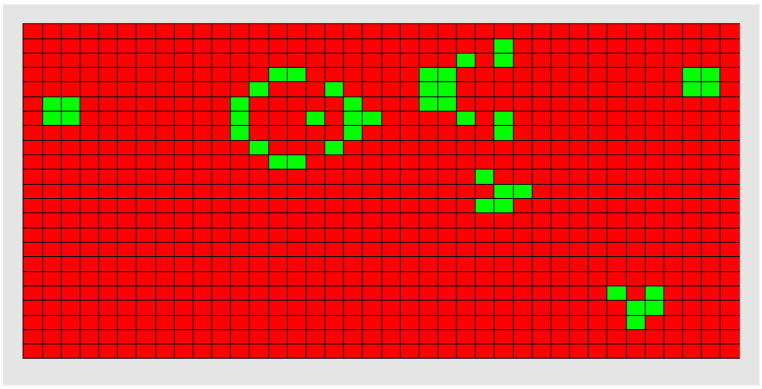


Figure 7: The Gosper Glider Gun after 61iterations

References

[1] FLAME web site - http://www.flame.ac.uk

[2] M. Gardner (1970) Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life". 223. pp. 120123. ISBN 0894540017.

[3] Conways' Game of Life, Wikipedia web page: http://en.wikipedia.org/wiki/Conway's_Game_of_Life

[4] U. Wilensky (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[5] P Kefalas *etal* (2003) "Communicating X-Machines: From Theory to Practice", chapter in Advances in Informatics, 8th Panhellenic Conference on Informatics, PCI 2001 Nicosia, Lecture Notes in Computer Science, Vol 2563, pp 21-33

[6] M Holcombe (1988) "X-Machines as a basis for dynamic system specification", Software Engineering Journal, Vol 3, Issue 2

[7] S Coakley (2005) "Formal Software Architecture for Agent-Based Modelling in Biology", PhD Thesis, University of Sheffield

[8] The home of `dot` and `dotty` http://www.graphviz.org/

[9] Stephen Silver's Life Lexicon, an explanation of over seven hundred terms used in Conway's Life: http://www.argentum.freeserve.co.uk/lex_home.htm

[10] PGPLOT web site - http://www.astro.caltech.edu/∼tjp/pgplot/

[11] C. Greenough, D.J. Worth, LS Chin, M. Holcome and S Coakley (2009), Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework, Rutherford Appleton Laboratory Technical Report RAL-TR-2009-022, Jul 2009

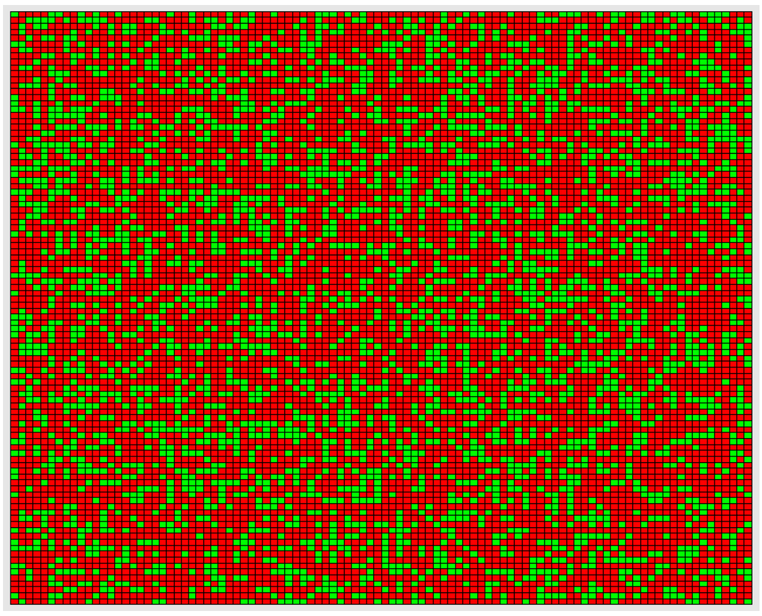


Figure 8: The initial random distribution of live cells

[12] R.J. Allen (2009) Survey of Agent Based Modelling and Simulation Tools, STFC Daresbury Laboratory Technical Report

[13] S. Coakley (2005) ”Formal Software Architecture for Agent-Based Modelling in Biology”, PhD Thesis, University of Sheffield

[14] S.A. Silver (2009) ”Gosper glider gun” - The Life Lexicon - http://www.argentum.freeserve.co.uk

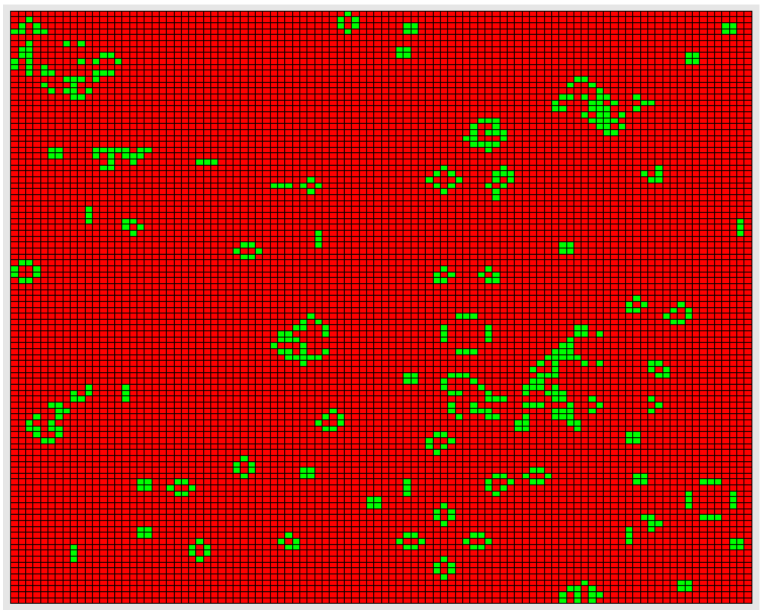


Figure 9: The distribution of live cells after 300 iterations

A FLAME XMML Model

Below is the full XMML definition of the Game of Life.

```
<xmodel version="2">

<name>Game of Life</name>
<version>01</version>

<!-- FLAME Environment -->
<environment>
   <constants>
      <variable><type>int</type><name>DOMAINX</name></variable>
      <variable><type>int</type><name>DOMAINY</name></variable>
   </constants>
   <functionFiles>
      <file>functions.c</file>
   </functionFiles>
</environment>

<!-- FLAME agents -->
<agents>
   <xagent>
      <name>Cell</name>
      <memory>
        <variable><type>int</type><name>id</name></variable>
        <variable><type>int</type><name>i</name></variable>
        <variable><type>int</type><name>j</name></variable>
        <variable><type>int</type><name>state</name></variable>
        <variable><type>int</type><name>neighbours[8]</name></variable>
      </memory>
      <functions>
         <function>
            <name>write_state</name>
            <currentState>start</currentState>
            <nextState>read_states</nextState>
            <outputs>
               <output><messageName>state</messageName></output>
            </outputs>
         </function>
         <function>
            <name>read_states</name>
            <currentState>read_states</currentState>
            <nextState>react</nextState>
            <inputs>
               <input>
                 <messageName>state</messageName>
               </input>
            </inputs>
         </function>
         <function>
            <name>react</name>
            <currentState>react</currentState>
            <nextState>end</nextState>
         </function>
      </functions>
   </xagent>
</agents>

<!-- FLAME messages -->
```

```
<messages>
   <message>
      <name>state</name>
      <variables>
         <variable><type>int</type><name>id</name></variable>
         <variable><type>int</type><name>i</name></variable>
         <variable><type>int</type><name>j</name></variable>
         <variable><type>int</type><name>state</name></variable>
      </variables>
   </message>
</messages>
</xmodel>
```

B FLAME C Functions

```
int write_state()
{
    int my_state, my_id, my_i, my_j;

    my_state = get_state();
    my_id = get_id();
    my_i = get_i();
    my_j = get_j();

    if (my_state == 1) {
        add_state_message(my_id, my_i, my_j, my_state);
    }

    return 0;
}

int read_states()
{
    int my_id, my_i, my_j;
    int *my_neighbours;

    int mes_id, mes_i, mes_j, mes_state;

    int count;

    my_id = get_id();
    my_i = get_i();
    my_j = get_j();
    my_neighbours = get_neighbours();

    for (count = 0; count <= 7; count++)
        my_neighbours[count] = 0;

    count = 0;
    START_STATE_MESSAGE_LOOP;
    if (state_message->id != my_id) {
        mes_id = state_message->id;
        mes_i = state_message->i;
        mes_j = state_message->j;
        if (((mes_i == my_i - 1) && (mes_j == my_j - 1)) ||
            ((mes_i == my_i - 1) && (mes_j == my_j)) ||
            ((mes_i == my_i - 1) && (mes_j == my_j + 1)) ||
            ((mes_i == my_i) && (mes_j == my_j - 1)) ||
            ((mes_i == my_i) && (mes_j == my_j + 1)) ||
            ((mes_i == my_i + 1) && (mes_j == my_j - 1)) ||
            ((mes_i == my_i + 1) && (mes_j == my_j)) ||
            ((mes_i == my_i + 1) && (mes_j == my_j + 1))) {
            mes_state = state_message->state;
            my_neighbours[count] = mes_state;
            count++;
        }
    }
    FINISH_STATE_MESSAGE_LOOP;
    return 0;
}

int react()
```

```
{

    int my_id, my_i, my_j, my_state;
    int *my_neighbours;
    int i, count;

    my_id = get_id();
    my_i = get_i();
    my_j = get_j();
    my_state = get_state();
    my_neighbours = get_neighbours();

    count = 0;
    for (i = 0; i < 8; i++) {
        if (my_neighbours[i] == 1) count++;
    }
    if (count < 2)  my_state = 0;
    if (count == 2) my_state = my_state;
    if (count == 3) my_state = 1;
    if (count > 3)  my_state = 0;

    set_state(my_state);

    return 0;
}
```