Technical Report
RAL-TR-2012-017

FLAME tutorial examples: a simple SIR infection model

DJ Worth, LS Chin, C Greenough

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: A Simple SIR Infection Model

DJ Worth, LS Chin, C Greenough

November 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 sheffield University and the Software Engineering Group of the Scientific Computing Department, STFC Rutherford Appleton Laboratory. This report documents the FLAME implementation of a simple SIR infection model which includes agent birth, movement and death. The model is a re-implementation of a NetLogo model and results can be compared with those of the NetLogo version.

Keywords: FLAME, agent-based modelling, SIR, infection modelling, tutorial example

---

Email: {`david.worth, shawn.chin, christopher.greenough`}`@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 **2**

2 A brief description of FLAME **2**

3 Problem description **3**

4 FLAME implementation **4**

5 The FLAME model **5**

5.1 FLAME environment 5

5.2 Agent memory 6

5.3 Agent functions 7

5.4 Agent messages 9

6 FLAME provided functions and macros **9**

7 Agent C functions **10**

8 Parsing the FLAME model **11**

9 The FLAME task dependency graph **11**

10 Input data generation **11**

11 Testing **11**

12 Comments on implementation **12**

A FLAME XMML Model **15**

B FLAME C Functions **19**

1 Introduction

This report describes the FLAME implementation of a simple SIR infection model. The standard SIR model proposed by Kermack and McKendrick in 1927 [1] includes populations of **S**usceptible, **I**nfected and **R**emoved agents and can be written as a set of differential equations

$$\begin{aligned}\frac{dS}{dt} &= -\beta SI \\ \frac{dI}{dt} &= \beta SI - \lambda I \\ \frac{dR}{dt} &= \lambda I\end{aligned}$$

where β is the rate of infection and λ is the rate of recovery.

In this formulation the total population is constant and there is no spatial spread of the agent population. This rather limits the "real world" application of this model but allows for analytical study of the population dynamics as the parameters change.

In the agent based approach described in this report we include agent birth and death (whether because of the infection or old age) and also movement which, though random in this initial study could be replaced by rule-based behaviour. The specifics of the model are taken from the NetLogo [6] biological virus model because it is straight forward to implement and also to give a comparison set of results with which to test the FLAME model.

2 A brief description of FLAME

FLAME (The Flexible Large-scale Agent Modelling Environment) is what it says - it is an environment for developing agent-based applications. FLAME is an agent-based applications generator. 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 (contain 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 [2, 3, 4].

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.

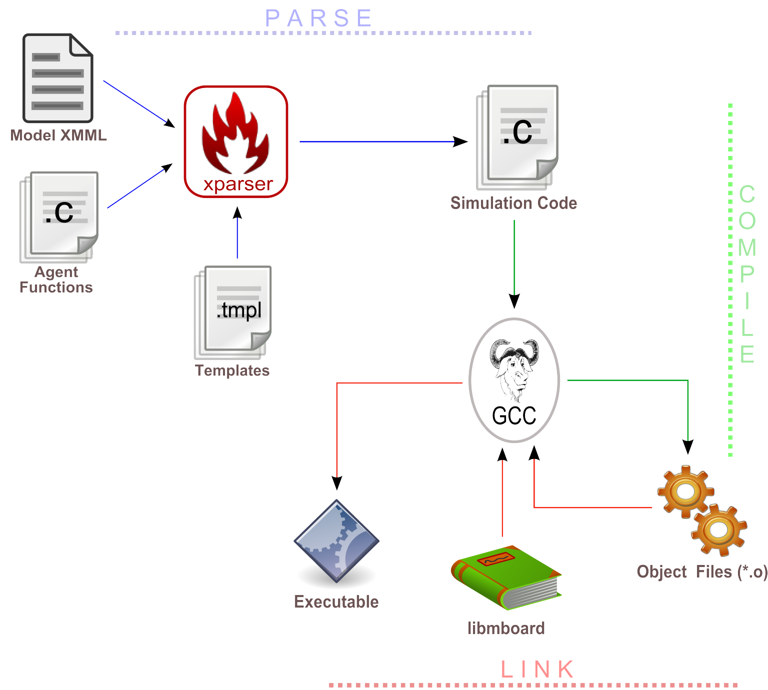

Figure 1: The structure of the FLAME Environment

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 naive approach might achieve 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.

3 Problem description

First we will give a very general description of the SIR infection model in the NetLogo library without reference to its implementation. Then we will give more details on the implementation before describing the FLAME implementation in the next section.

The SIR infection model has agents moving randomly on a domain encountering each other with a fixed chance of picking up an infection from an infected agent. Healthy agents may reproduce a limited number of times if they are healthy and give birth to agents susceptible to infection. The total population in the domain is limited by a predefined carrying capacity. Agents who are infected have a fixed chance of recovery but if they are infected for too long they will die. In addition agents who live beyond the allotted lifespan will also die.

The implementation within NetLogo imposes the following details on this general framework

- Agents move and act in a sequence of iterations.
- The domain is toroidal.
- Agents have an (x, y) position and heading.

- Heading is modified $\pm 100^{\circ}$ each iteration and agents move one unit in the new direction.
- Agents can only be infected by agents in the same "patch"[1]
- Infected agents *push* the infection to susceptible agents in their patch.
- Agent's age is incremented each iteration.

The details of the algorithms at the decision points are given below.

Reproduction

Check first that the agent is not sick and there is carrying capacity in the environment. If there is then draw a random number from a uniform distribution from 0 the agent lifespan. If this number is less than the average number of offspring an agent can have then create a new agent.

Infection

Draw a random number from a uniform distribution from 0 to 100. If this number is less than the percentage of people becoming infected when in contact with infection then the agent will become sick.

When an agent is infected

Draw a random number from a uniform distribution from 0 to the number of iterations for which the agent has been infected. If this number is less than or equal to the duration of the infection scaled by the agent lifespan/100 then the agent is still infected. If it is greater draw a random number from a uniform distribution from 0 to 100. If this number is less than the percentage chance of recovery then the agent becomes immune. If not the agent dies.

Death from old age

Agent dies if its age becomes greater then the lifespan of agents.

4 FLAME implementation

Our stated aim of comparing the NetLogo infection model with a FLAME version provides contraints on the way we design the model and implement the agent functions. Agents will have a position (x, y) in the domain and a heading (initially randomly distributed). There are no predefined domain types in FLAME so the agent movement function must take into account the toroidal domain imposed by NetLogo. In addition the agents will have boolean flags which indicate whether they are sick or immune. Susceptible agents will have these flags both set to 0, each of the other states will have the appropriate flag set to 1 and the other to 0.

There is no concept of a patch in FLAME and so this will have to be explicity included in the agent function that finds the infected agents from which an agent can itself become infected.

One difference in the FLAME implementation is that susceptible agents will *pull* infection when they are close to an infected agent. All communication between agents in FLAME is via *message boards* and the mechanism for contracting infection is as follows:

- Sick agents post their location to a message board.
- Susceptible agents read the locations of infected agents.

[1] The domain in NetLogo is divided into a number of 2D patches from which an agent can collect environment data or determine neighbouring agents.

- Susceptible agents pick out the messages from their patch and calculate whether they become infected.

This approach means that there are fewer messages to pass around and read for each agent as only the sick agents post their location. The alternative, infection *pushing*, approach would mean that all susceptible agents had to post their location which would be inefficient when the the infected agents were in a minority. On a more conceptual level the idea that one agent decides the state of another goes against the autonomous approach inherent in agent based modelling.

5 The FLAME model

We firstly define the various elements of the FLAME model:

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 `infection.xml` model is given in Appendix A. Within each of these sections there are various tags to define the FLAME model. Only the elements relating to the infection model will be highlighted below and the user should refer to the 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 infection model there are parameters that define the domain and the characterise the infection

```
<environment>
    <functionFiles>
        <file>functions.c</file>
    </functionFiles>
    <constants>
        <variable>
          <type>int</type>
          <name>lifespan</name>
          <description>Lifespan of agent in weeks</description>
        </variable>
        <variable>
          <type>int</type>
          <name>average_offspring</name>
          <description>Average number of offspring an agent could have</description>
        </variable>
        <variable>
          <type>int</type>
          <name>carrying_capacity</name>
          <description>Maximum number of agents that can be in the world at one
                       time</description>
        </variable>
        <variable>
          <type>float</type>
          <name>infectiousness</name>
          <description>Percentage of people becoming infected when in contact with
                       infection</description>
```

```
        </variable>
        <variable>
          <type>float</type>
          <name>chance_recovery</name>
          <description>Percentage change of recovery from infection</description>
        </variable>
        <variable>
          <type>float</type>
          <name>duration</name>
          <description>Virus duration in weeks</description>
        </variable>
        <variable>
          <type>float</type>
          <name>width</name>
          <description>Width of domain</description>
        </variable>
        <variable>
          <type>float</type>
          <name>height</name>
          <description>Height of domain</description>
        </variable>
    </constants>
</environment>
```

The names of constants can be used within the user's functions but they must be in upper case, e.g. `CHANCE_RECOVERY` The `file` tag is used by the FLAME parser to locate the function source code.

5.2 Agent memory

The start of an agent definition is tagged by `<xagent>` and the agent memory variables are then defined. Within FLAME there are no pre-defined global data elements apart from some limiting global constants so all *permanent* data must be held in an agent's memory. Each agent behaves in the same way and so there is only one agent type with the following internal memory:

```
<memory>
  <variable><type>int</type><name>id</name></variable>
  <variable><type>double</type><name>x</name></variable>
  <variable><type>double</type><name>y</name></variable>
  <variable><type>double</type><name>heading</name>
            <description>Angle in degrees clockwise from N</description>
  </variable>
  <variable><type>int</type><name>is_sick</name>
            <description>If 1 agent is infectious</description>
  </variable>
  <variable><type>int</type><name>is_immune</name>
            <description>If 1 agent can't be infected</description>
  </variable>
  <variable><type>int</type><name>sick_count</name>
            <description>How long agent has been infectious in weeks</description>
  </variable>
  <variable><type>int</type><name>age</name>
            <description>How many weeks old agent is</description>
  </variable>
</memory>
```

We have give each agent an *id* so that it would be possible, in post-processing of the results, to follow the health of a particular agent through its life.

5.3 Agent functions

The number of functions or states the agent has is determined by the approach taken. We have decided, again following the NetLogo model, that the agent will have states that perform only one action. So we define the following functions or state transitions which are illustrated in Figure 2:

get_older : Updating the agent's age, length of infection and causing agent to die if it is too old.

move : Updating the agent's heading and moving the agent.

post_position : Only called for infected agents - posting the agent's position.

infect : Only called for susceptible agents - reading the position messages and deciding if the agent becomes infected.

recover : Only called for infected agents - calculation of recovery or death from infection.

reproduce : Only called for uninfecteed agents - a function to create a new agent if appropriate.

To restrict the agents for which certain functions are called we use state branching. This reduces the number of function calls compared to an implementation in which the restrictions are computed within the functions. Furthermore, it increases the scope for parallel execution, allowing separate branches to be followed in different 'threads' as no agent can be in more than one state at a time. Details on how the state branching is introduced into the more are given below, see for example the function `post_position`.

When agents in a particular state can do nothing while agents in other states call their functions we use an *idle* function in the model definition. FLAME recognises the name as a marker, hence there is no need to use distinct names and the framework can construct the implementation of the functions for itself.

The XMML function definitions follow.

get_older : This function updates the agent's age, length of infection and causes the agent to die if it is too old. It returns 1 if the agent is to die indicating that the framework should perform garbage collection on the agent memory.

```
<function><name>get_older</name>
  <currentState>start</currentState>
  <nextState>1</nextState>
</function>
```

move : This function updates the agent's heading and moves the agent one unit in the new direction taking into account the toroidal nature of the domain.

```
<function><name>move</name>
  <currentState>1</currentState>
  <nextState>2</nextState>
</function>
```

post_position : This function simply posts the position of the agent to the *infected* message board. The `<condition>` tag specifies that this function is only called when the `is_sick` memory variable of an agent is 1, i.e. the agent is infected.

```
<function><name>post_position</name>
  <currentState>2</currentState>
  <nextState>3</nextState>
  <outputs>
```

```
    <output><messageName>infected</messageName></output>
  </outputs>
  <condition>
    <lhs><value>a.is_sick</value></lhs>
    <op>EQ</op>
    <rhs><value>1</value></rhs>
  </condition>
<function>
```

infect : This function reads position of infected agents from the *inefcted* message board and decides if the agent becomes infected. This time the `<condition>` tag ensures this function is only called for susceptible agents.

```
<function><name>infect</name>
  <currentState>2</currentState>
  <nextState>4</nextState>
  <inputs>
    <input><messageName>infected</messageName></input>
  </inputs>
  <condition>
    <lhs>
      <lhs><value>a.is_sick</value></lhs>
      <op>EQ</op>
      <rhs><value>0</value></rhs>
    </lhs>
    <op>AND</op>
    <rhs>
      <lhs><value>a.is_immune</value></lhs>
      <op>EQ</op>
      <rhs><value>0</value></rhs>
    </rhs>
  </condition>
</function>
```

idle : Indicates that agents that are immune to the infection should do nothing during this state transition.

```
<function><name>idle</name>
  <description></description>
  <currentState>2</currentState>
  <nextState>4</nextState>
  <condition>
    <lhs><value>a.is_immune</value></lhs>
    <op>EQ</op>
    <rhs><value>1</value></rhs>
  </condition>
</function>
```

recover : This function decides whether an infected agent recovers from the infection, continues as infected or dies from the infection. It returns 1 if the agent is to die indicating that the framework should perform garbage collection on the agent memory.

```
<function><name>recover</name>
  <currentState>3</currentState>
  <nextState>4</nextState>
</function>
```

reproduce : This function creates a new agent if appropriate. The `<condition>` tag means this function is only called for healthy agents.

```
<function><name>reproduce</name>
  <currentState>4</currentState>
  <nextState>end</nextState>
  <condition>
    <lhs><value>a.is_sick</value></lhs>
    <op>EQ</op>
    <rhs><value>0</value></rhs>
  </condition>
</function>
```

idle : Indicates that agents that are sick should do nothing during this state transition.

```
<function><name>idle</name>
  <description></description>
  <currentState>4</currentState>
  <nextState>end</nextState>
  <condition>
    <lhs><value>a.is_sick</value></lhs>
    <op>EQ</op>
    <rhs><value>1</value></rhs>
  </condition>
</function>
```

It would be possible to combine some functions into single functions but the current division serves the purpose. Furthermore it lays down a framework in which more complex functions with greater dependencies can replace the simple functions in this initial implementation. 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.

5.4 Agent messages

As mentioned above all agent communication occurs via message boards. In this model we define one message type - *infected*. This provides all the information required by agents to implement the interactions.

```
<message>
  <name>infected</name>
  <description>Position and id of sick agent</description>
  <variables>
    <variable><type>int</type><name>id</name><description></description></variable>
    <variable><type>double</type><name>x</name><description></description></variable>
    <variable><type>double</type><name>y</name><description></description></variable>
  </variables>
</message>
```

The complete XMML model is given in Appendix A

6 FLAME provided functions and macros

Before considering in detail the transition functions 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. See Section 5.1 for the environment definition in the infection model. Once parsed these constants are defined in the file `header.h` as C macros (uppercase of the variable name) 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 model under discussion here as there is only one agent type.

Accessing message boards : For each message type defined in the model FLAME generates two important access mechanisms: one to write these messages to the associated message board and another to read the messages from the 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 *infected* message board in the infection model FLAME provides the function:

```
void add_infected_message(int id, double x, double y);
```

Generically this will be:

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

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

Accessing message data : Accessing message data is a little 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 `infected` and it holds the following data `id`, `x` and `y`. 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 pointers to access the message data elements. The macro initialises a pointer - *message_board_name*`_message` - to the message structure so that

message_board_name`_message ->` *data_element*

can be used to access elements of a message.

END_message_boards_name_LOOP : Iterates the pointer to the next message in the loop and terminates the message board loop when complete.

7 Agent C functions

Associated with each agent state change is a C function that performs the change. These functions are named according to the names given in the `<function>` tags of the agent definition. They return an integer value which the FLAME framework expects to be either 0 - the agent is OK do nothing; or 1 - the agent should be destroyed. When the infection model determines that an agent should die, either because of infection or old age then that particular function returns 1. Before each function call the framework organises its memory so that the memory variables of the agent on which the function should operate are available with the `get_` and `set_` functions described in Section 6.

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 defined in XMML (model.xml) and the agent functions written (functions.c) the complete model can now be parsed with the FLAME parser.

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

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

9 The FLAME task dependency graph

FLAME uses a task dependency graph to schedule the execution of agent functions and communications. One of the files 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 graph for the SIR model.

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

10 Input data generation

The initial data for the model is generated using a simple python program. The program generates the specified number of agents in random positions with random headings in a given domain. The first 10 agents are set to be infected. The envinronment variables are hard-coded into the file but can be easily changed.

```
Usage: init_start_state.py <width> <height> <agent_count>
More info:
    <width> and <height> will determine the size of simulation space in which
    agents will be randomly placed in.

    <agent_count> specifies the number of agents to initialise.
```

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

11 Testing

The method of testing this implementation of a simple SIR model is to test it against the results from the NetLogo model with the same parameters. Further tests against the differential equation formulation would be possible provided parameters for the the agent based model

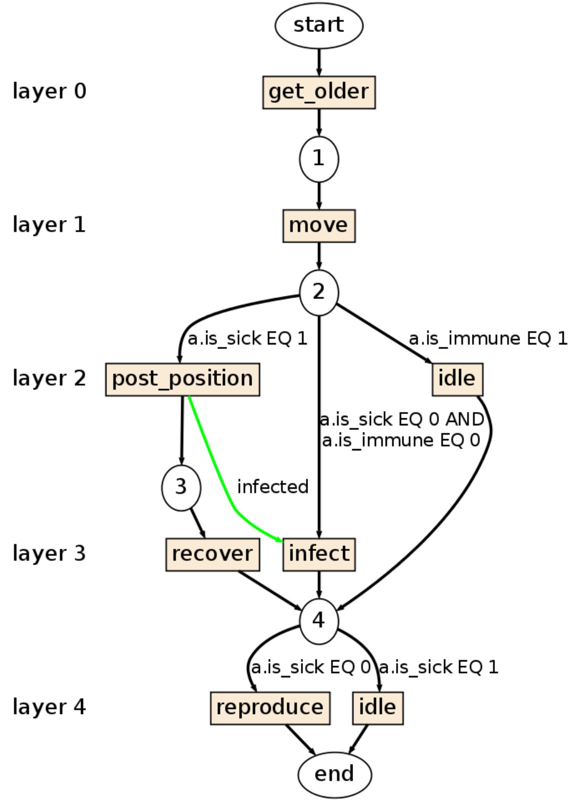

Figure 2: The *task dependency graph* of the SIR infection model

Parameter	**Value**	**Parameter**	**Value**
Initial number of agents	150	lifespan	100
Domain width	34	Domain height	34
Carrying capacity	750	Average offspring	4
Chance of recovery	50%	Infectiousness	65%
Duration of infection	20		

Table 1: The model parameters used in comparison of NetLogo and FLAME implementations

could be chosen to remove birth and death and the parameters of the infection β and λ can be identified.

The parameters for the NetLogo and FLAME model are given in Table 1. Figures 3 and 4 show graphs of the sesceptible, infected and removed populations and it is clear that the results are similar between the NetLogo and the FLAME implementations.

12 Comments on implementation

Performance of this model strongly depends on the product of the number of susceptible agents and the number of infected agents since each susceptible agent must read all the messages from the infected agents. We have chosen that the infected agents post messages rather than those who are susceptible because initially there are fewer of these messages. However, in the case of a virrulent disease the majority of the population will become infected and so a large number of infected messages are produced. If filters were available to restrict the messages that each susceptible agent sees to only those from its patch then the time taken to search through the messages would be significantly reduced.

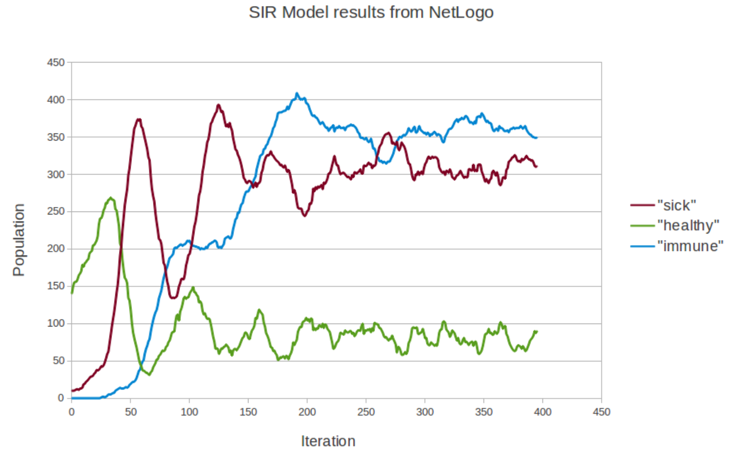

Figure 3: Results from NetLogo for simple SIR model

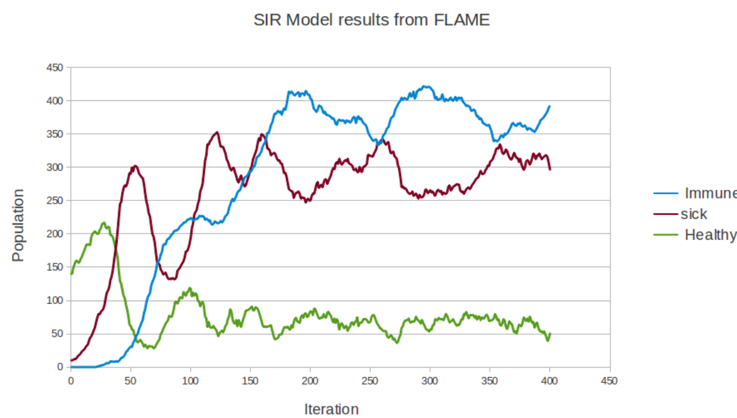

Figure 4: Results from FLAME for simple SIR model

It may also be possible to dynamically swap from susceptible agents pulling the infection to infected agents pushing the infection. If there were a large number of infected messages that had to be sent then having susceptible agents (presumably a small number) posting their locations would reduce the amount of messages to be read.

We have one final point concerning the model itself. Netlogo uses patches to determine the agents with which an agent can possibly interact. This is fine for a model where patches arise naturally but in this infection model the setting is a continuous space. Agents move freely and should be able to interact with those who are close enough, no matter what artificially imposed patch they belong to. In this model there can occur situations in which a possible infection pathway is missed because the infected agent is not in the same patch as the susceptible one.

References

[1] V. Capasso (1993), Mathematical Structures of Epidemic Systems, Springer Verlag.

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

[3] C. Greenough, D.J. Worth, L.S. 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.

[4] M. Holcombe, S. Coakley, R. Smallwood (2006), A General Framework for agent-based modelling of complex systems, Proceedings of the 2006 European Conference on Complex Systems.

[5] P. Kefalas, G. Eleftherakis, E. Kehris (2003), Communicating X-machines: A practical approach for formal and modular specification of large systems, Journal of Information and Software Technology, **45**, pp 269–280, Elsevier.

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

A FLAME XMML Model

```
<xmodel version="2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation='http://flame.ac.uk/schema/xmml_v2.xsd'>

<name>Simple SIR infection model</name>
<version>01</version>
<description></description>

<environment>
    <functionFiles>
        <file>functions.c</file>
    </functionFiles>
    <constants>
        <variable>
          <type>int</type>
          <name>lifespan</name>
          <description>Lifespan of agent in weeks</description>
        </variable>
        <variable>
          <type>int</type>
          <name>average_offspring</name>
          <description>Average number of offspring an agent could have</description>
        </variable>
        <variable>
          <type>int</type>
          <name>carrying_capacity</name>
          <description>Maximum number of agents that can be in the world at one time</description>
        </variable>
        <variable>
          <type>float</type>
          <name>infectiousness</name>
          <description>Percentage of people becoming infected when in contact with
                      infection</description>
        </variable>
        <variable>
          <type>float</type>
          <name>chance_recovery</name>
          <description>Percentage change of recovery from infection</description>
        </variable>
        <variable>
          <type>float</type>
          <name>duration</name>
          <description>Virus duration in weeks</description>
        </variable>
        <variable>
          <type>float</type>
          <name>width</name>
          <description>Width of domain</description>
        </variable>
        <variable>
          <type>float</type>
          <name>height</name>
          <description>Height of domain</description>
        </variable>
    </constants>
</environment>

<agents>
```

```
<xagent>

    <name>Person</name>
    <description></description>

    <memory>
        <variable><type>int</type><name>id</name><description></description></variable>
        <variable><type>double</type><name>x</name><description></description></variable>
        <variable><type>double</type><name>y</name><description></description></variable>
        <variable><type>double</type><name>heading</name>
                  <description>Angle in degrees clockwise from N</description>
        </variable>
        <variable><type>int</type><name>is_sick</name>
                  <description>If 1 agent is infectious</description>
        </variable>
        <variable><type>int</type><name>is_immune</name>
                  <description>If 1 agent can't be infected</description>
        </variable>
        <variable><type>int</type><name>sick_count</name>
                  <description>How long agent has been infectious in weeks</description>
        </variable>
        <variable><type>int</type><name>age</name>
                  <description>How many weeks old agent is</description>
        </variable>
    </memory>

    <functions>

        <function><name>get_older</name>
            <description></description>
            <currentState>start</currentState>
            <nextState>1</nextState>
        </function>

        <function><name>move</name>
            <description></description>
            <currentState>1</currentState>
            <nextState>2</nextState>
        </function>

        <function><name>post_position</name>
            <description></description>
            <currentState>2</currentState>
            <nextState>3</nextState>
            <outputs>
                <output><messageName>infected</messageName></output>
            </outputs>
            <condition>
                <lhs><value>a.is_sick</value></lhs>
                <op>EQ</op>
                <rhs><value>1</value></rhs>
            </condition>
        </function>

        <function><name>infect</name>
            <description></description>
            <currentState>2</currentState>
            <nextState>4</nextState>
```

```
    <inputs>
        <input>
            <messageName>infected</messageName>
        </input>
    </inputs>
    <condition>
        <lhs>
            <lhs><value>a.is_sick</value></lhs>
            <op>EQ</op>
            <rhs><value>0</value></rhs>
        </lhs>
        <op>AND</op>
        <rhs>
            <lhs><value>a.is_immune</value></lhs>
            <op>EQ</op>
            <rhs><value>0</value></rhs>
        </rhs>
    </condition>
</function>

<function><name>idle</name>
    <description></description>
    <currentState>2</currentState>
    <nextState>4</nextState>
    <condition>
        <lhs><value>a.is_immune</value></lhs>
        <op>EQ</op>
        <rhs><value>1</value></rhs>
    </condition>
</function>

<function><name>recover</name>
    <description></description>
    <currentState>3</currentState>
    <nextState>4</nextState>
</function>

<function><name>reproduce</name>
    <description></description>
    <currentState>4</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs><value>a.is_sick</value></lhs>
        <op>EQ</op>
        <rhs><value>0</value></rhs>
    </condition>
</function>

<function><name>idle</name>
    <description></description>
    <currentState>4</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs><value>a.is_sick</value></lhs>
        <op>EQ</op>
        <rhs><value>1</value></rhs>
    </condition>
</function>
```

```
        </functions>
    </xagent>

</agents>

<messages>
    <message>
        <name>infected</name>
        <description>Position and id of sick agent</description>
        <variables>
            <variable><type>int</type><name>id</name><description></description></variable>
            <variable><type>double</type><name>x</name><description></description></variable>
            <variable><type>double</type><name>y</name><description></description></variable>
        </variables>
    </message>
</messages>

</xmodel>
```

B FLAME C Functions

```
#include <math.h>
#include "header.h"

/* Need count of the number of agents */
int GLOBAL_num_agents = 0;

double PI=3.14159267;

/* Set variables when agent becomes infected */
void become_sick()
{
    set_is_sick(1);
    set_is_immune(0);
}

/* Set variables for healthy agent */
void become_healthy()
{
    set_is_sick(0);
    set_is_immune(0);
    set_sick_count(0);
}

/* Set variables when agent becomes immune */
void become_immune()
{
    set_is_sick(0);
    set_is_immune(1);
    set_sick_count(0);
}

/* Increment age and length of time agent has been sick. If too old then die */
int get_older()
{
    /* Read agent memory */
    int age = get_age(), is_sick = get_is_sick(), sick_count = get_sick_count();

    /* Do some accounting in the first iteration */
    if (iteration_loop == 1) GLOBAL_num_agents++;

    /* Increment age */
    set_age(age+1);

    /* If I'm sick increment the time I've been sick */
    if (is_sick == 1)
    {
        set_sick_count(sick_count+1);
    }

    /* Die of old age if age exceeds lifespan */
    if (age > LIFESPAN)
    {
        GLOBAL_num_agents--;
        return 1; /* Shuffling off now ... Bye ... */
    }

    /* Otherwise all is OK */
    return 0;
}

/* Move about at random */
int move()
```

```
{
    /* Read agent memory */
    double x = get_x(), y = get_y();
    double heading = get_heading();

    /* Change heading up to 100 degrees in either direction */
    heading += -100.0 + 200.0 * rand()/(RAND_MAX+1.0);

    /* Calculate new position. Note heading is in degrees relative to North, i.e. +ve y axis so must  */
    /* convert to radians */
    x += sin(heading/180.0*PI);
    /* Hard coded domain wrapping in x direction */
    if (x > WIDTH/2.0) x -= WIDTH;
    if (x < -WIDTH/2.0) x += WIDTH;
    y += cos(heading/180.0*PI);
    /* Hard coded domain wrapping in y direction */
    if (y > HEIGHT/2.0) y -= HEIGHT;
    if (y < -HEIGHT/2.0) y += HEIGHT;

    /* Set new position and heading*/
    set_x(x);
    set_y(y);
    set_heading(heading);

    return 0;  /* remain alive. 1 = death */
}

/* Post the position of the agent. Should only be called for infected agents via condition in XMML. */
int post_position()
{
    /* Read agent memory */
    int id = get_id();
    double x = get_x(), y = get_y();

    /* Add message to "infected" board  (id, x, y) */
    add_infected_message(id, x, y);

    return 0;  /* remain alive. 1 = death */
}

/*
If agent is neither immune nor already infected (condition defined in XMML) then there is a chance of
becomming sick if agent within 1x1 patch is sick. Position of sick agents posted on messageboard.
How kind!!
*/
int infect()
{
    /* Read agent memory */
    double x1 = get_x(), y1 = get_y();

    /* Local variables */
    double x2 = 0.0, y2 = 0.0;

    /* Loop through all messages */
    START_INFECTED_MESSAGE_LOOP
        x2 = infected_message->x;
        y2 = infected_message->y;

        /* Check whether message came from within a 1x1 patch around me */
        if ( (trunc(x1) == trunc(x2)) && (trunc(y1) == trunc(y2)) )
        {
        /* Do I get sick? */
        if (100.0 * rand()/(RAND_MAX+1.0) < INFECTIOUSNESS ) become_sick(); /* Bleurgh */
        }
```

```
    FINISH_INFECTED_MESSAGE_LOOP

    return 0;
}

/* Once agent has been sick long enough it either recovers and becomes immune or dies. */
int recover()
{
    if ( rand()/(RAND_MAX+1.0)*get_sick_count() > LIFESPAN * DURATION / 100.0)
    {
        if ( rand()/(RAND_MAX+1.0) * 100.0 < CHANCE_RECOVERY )
        {
            become_immune(); /* I survived. Yay!! */
        }
        else
        {
            GLOBAL_num_agents--;
            return 1;  /* Oh no!! */
        }
    }
    return 0;
}

/* If there are less agents than the carrying capacity then healthy agents can reproduce */
int reproduce()
{
    if ( (GLOBAL_num_agents < CARRYING_CAPACITY) && (rand()/(RAND_MAX+1.0)*LIFESPAN < AVERAGE_OFFSPRING) )
    {
        add_Person_agent(GLOBAL_num_agents*iteration_loop+get_id(),get_x(),get_y(),
                         rand()/(RAND_MAX+1.0)*360.0,0,0,0,1);
        GLOBAL_num_agents++;
    }
    return 0;
}
```