Science & Technology Facilities Council

Technical Report
RAL-TR-2014-008

FLAME Tutorial Examples: Predation - a simple predator-prey model

GL Poulter, C Greenough

August 2014

Enquiries concerning this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
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

FLAME Tutorial Examples: Predation - a simple predator-prey model

GL Poulter, C Greenough

June 2014

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 a simple predator-prey model. Within a fixed domain sheep and wolves move randomly, and this costs a wolf 1 unit of energy. If, after a particular move a wolf and sheep are close enough, the wolf is able to eat the sheep and increase the amount of energy it has.

Keywords: FLAME, agent-based modelling, tutorial example

Email: {gemma.poulter, 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 **1**

2 A brief description of FLAME **1**

3 Problem description **2**

4 FLAME implementation **3**

5 The FLAME model **4**
5.1 FLAME environment 5
5.2 Agent memory 6
5.3 Agent functions 7
5.4 Agent messages 13

6 FLAME provided functions and macros **14**

7 Agent C functions **15**

8 Parsing the FLAME model **15**

9 The FLAME task dependency graph **15**

10 Input data generation **18**

11 Testing **18**
11.1 Setup 18
11.2 Model without grass 18
11.3 Model with grass 20

12 Parallel implementation **20**
12.1 Design 20
12.2 Initial data partition 21
12.3 Results 23

13 Comments on implementation **26**

A FLAME XMML Model **28**
A.1 FLAME XMML Model without grass 28
A.2 FLAME XMML Model with grass 32

B FLAME C Functions **39**
B.1 Wolf functions 39
B.2 Sheep functions 41
B.3 Grass functions 43

1 Introduction

This report describes the FLAME implementation of a simple predator-prey model. The specifics of the model are taken from the NetLogo [5], Wolf Sheep Predation model, because it is straightforward to implement and because running the NetLogo simulation provides a set of results with which to test the FLAME model.

We begin by giving an introduction to the FLAME software in Section 2 and follow that with details of the predation model in Section 3. The various components of the FLAME implementation are then given in Sections 4 - 9. We describe, in Sections 10 and 11, how we can test the model and compare it to the NetLogo version, as a form of validation. The majority of this report is focused on the implementation and validation of the predator model in series, however parallel execution is available in FLAME and we discuss this further in Section 12. Finally, the full model and codes are available in the Appendices.

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.* [4] 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 their model and the functions that define the 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 [1, 2, 3].

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 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 their 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 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 describe these in the

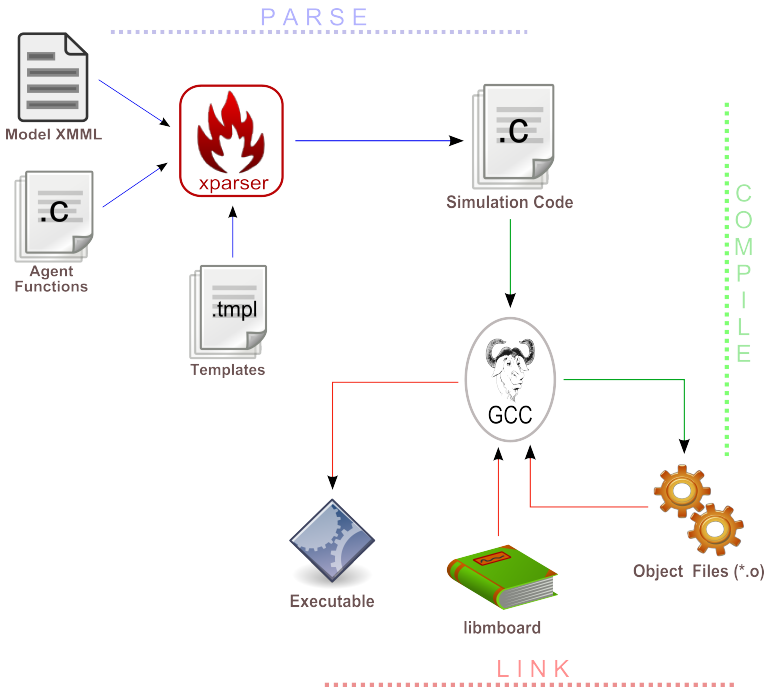


Figure 1: The structure of the FLAME Environment

section on the implementation (4).

3 Problem description

Firstly, we give a very general description of the Wolf Sheep Predation model in the NetLogo library without reference to its implementation. Then we will give more details on the implementation before describing the FLAME version in the next section.

The Wolf Sheep Predation model has 2 variants. In the more basic form, there are sheep and wolves which move randomly within a fixed domain, and this costs a wolf some energy. If, after a particular move, a wolf and sheep are close enough, the wolf is able to eat the sheep and increase the amount of energy it has. Wolves die when they run out of energy.

The second version of the model includes grass as an additional component. If sheep move to a location which has grass, they are able to eat. This increases the amount of energy they have, but they can no longer move for free, as similarly to wolves, with each move, some energy is lost. Sheep will now die either when they are eaten by a wolf, or if they run out of energy. When a sheep eats grass, the grass in that area is set to re-grow afer a set length of time. Both wolves and sheep are able to reproduce with set probabilites.

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

- Wolf and sheep agents move and act in a sequence of iterations.
- The domain is toroidal.
- Wolf and sheep agents have an (x, y) position and heading.
- Heading is modified $\pm 100^\circ$ each iteration and the agents move one unit in the new direction.
- A wolf's energy is reduced by 1 unit each time it moves.
- Wolf agents can only eat sheep agents in the same "patch"[1]

[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.

- A wolf's energy increases each time it catches a sheep
- A wolf dies if it has run out of energy.

Additonally, when grass is included in the model:

- Each "patch" is either "green" or "brown", indicating whether grass is currently there.
- A sheep's energy is reduced by 1 unit each time it moves.
- A sheep's energy increases each time it eats grass.
- A sheep dies if it has run out of energy.
- Once eaten, grass regrows after a set number of iterations.

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

Sheep catching

A wolf can only eat a maximum of one sheep per iteration; it chooses at random which sheep to eat if there are multiple sheep on its patch. Also, a sheep cannot be shared between multiple wolves. If there are 2 wolves and 1 sheep on the same patch, the first wolf to attempt to catch the sheep will do so.

Reproduction

Draw a random number from a uniform distribution from 0 to 100. If this number is less than the set value for reproduction then create a new agent. The parent's energy is shared with the new agent, so both will have half of the parent's energy for the next iteration. Note that the set value for reproduction can be different for wolves and sheep.

4 FLAME implementation

Our stated aim of comparing the NetLogo predation model with a FLAME version provides constraints on the way we design the model and implement the agent functions. All agents will have a position (x, y) in the domain while wolf and sheep agents also have 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 wolf and sheep agents will have variables which contain the amount of energy they have, while grass agents have a boolean flag indicating whether there is currently grass at their location. Unique identifiers are required for the wolves and sheep and the reasons for this will be explained later in this section.

There is no concept of a patch in FLAME and so this will have to be explicity included in the agent function that hunts sheep or grass.

All communication between agents in FLAME is via *message boards* and in this model we have 3 message boards, or 4 when grass is included. The mechanism for "hunting" is as follows:

1. Sheep post their (x, y) coordinates to a *location* message board.
2. Wolves read the positions of the sheep from the *location* message board.
3. Wolves pick out the messages from their patch and then post their ID along with the ID of 1 sheep on its patch, to an *order* message board. Note if there is more than 1 sheep on its patch then the wolf will choose one of the sheep at random.

4. Sheep read the *order* message board.

5. Sheep pick out the messages on the *order* board which contain their ID. They then post the corresponding wolf ID to a *sheep_food* message board before deleting themselves. If they appear more once on the *order* board they choose at random which wolf will eat them.

6. Wolves read the *sheep_food* message board.

7. Wolves pick out the messages on the *sheep_food* board which contain their ID. They then increase their energy accordingly.

Note that the model could have been implemented in FLAME differently, for example both sheep and wolves could post their locations to message boards. This way both wolf and sheep can independently calculate if they are in close enough proximity for predation to occur. The inclusion of 3 message boards in total, however, is still necessary regardless of the approach. This may seem excessive and at first glance 2 boards would appear to be sufficient. The complexity of an additional board is evident though when we remember that a sheep cannot be eaten by more than one wolf. Consider the situation where there are 2 wolves and 1 sheep all on the same patch. Even if sheep and wolves both post and subsequently read each others' locations, there is still a conflict which needs to be resolved: it must be decided which wolf will eat the sheep and this must be communicated to both wolves. The only way of solving this conflict and communicating the results is via a third message board.

When grass is included in the model a fourth message board is required as a sheep can only eat of course, if it has moved to a patch containing grass. The mechanism for "grazing" is as follows:

1. Grass agents which currently have grass read the positions of the sheep from the *location* message board.

2. Grass agents pick out the messages from their patch and then post the ID of a sheep on their patch, to a *grass_food* message board. Note that if there is more than 1 sheep on its patch then the grass agent will choose one of the sheep at random.

3. Grass agents then modify the boolean flag to indicate they no longer have grass at this time.

4. Sheep read the *grass_food* message board.

5. Sheep pick out the messages on the *grass_food* board which contain their ID. They then increase their energy accordingly.

5 The FLAME model

We firstly define the various elements of the FLAME model:

environment - definitions of global variables and C code file names

agents - all things related to the agents

messages - the things related to the model messages

A full listing of the `predation.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 predation 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 predation model there are parameters that define the domain and characterise the behaviour of the agents. The environment section shown below includes the addition of grass in the model.

```
<environment>
    <constants>
        <variable>
            <type>double</type>
            <name>reproduce_sheep_prob</name>
            <description>Probability that a sheep will reproduce</description>
        </variable>
        <variable>
            <type>double</type>
            <name>reproduce_wolf_prob</name>
            <description>Probability that a wolf will reproduce</description>
        </variable>
        <variable>
            <type>int</type>
            <name>gain_from_food_wolf</name>
            <description>Amount of energy a wolf gains having eaten a sheep</description>
        </variable>
        <variable>
            <type>int</type>
            <name>gain_from_food_sheep</name>
            <description>Amount of energy a sheep gains having eaten grass</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>
        <variable>
            <type>int</type>
            <name>grass_regrowth</name>
            <description>Number of iterations before grass can re-grow</description>
        </variable>
    </constants>
    <functionFiles>
        <file>sheep_functions.c</file>
        <file>wolf_functions.c</file>
        <file>grass_functions.c</file>
    </functionFiles>
</environment>
```

The names of constants can be used within the user's functions but they must be in upper case, e.g. `GAIN_FROM_FOOD_WOLF`. 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. We have 3 types of agent in the full model with following internal memories:
Wolf:

```
<memory>
    <variable>
        <type>int</type>
        <name>wolf_id</name>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_x</name>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_y</name>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_heading</name>
        <description>Angle in degrees clockwise from N</description>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_energy</name>
        <description>Amount of energy the wolf has</description>
    </variable>
</memory>
```

Sheep:

```
<memory>
    <variable>
        <type>int</type>
        <name>sheep_id</name>
    </variable>
    <variable>
        <type>double</type>
        <name>sheep_x</name>
    </variable>
    <variable>
        <type>double</type>
        <name>sheep_y</name>
    </variable>
    <variable>
        <type>double</type>
        <name>sheep_heading</name>
        <description>Angle in degrees clockwise from N</description>
    </variable>
    <variable>
        <type>double</type>
        <name>sheep_energy</name>
        <description>Amount of energy the sheep has</description>
    </variable>
</memory>
```

Grass:

```
<memory>
    <variable>
        <type>int</type>
        <name>grass_x</name>
    </variable>
    <variable>
        <type>int</type>
        <name>grass_y</name>
    </variable>
    <variable>
        <type>int</type>
        <name>grass_colour</name>
        <description>1 means there is currently grass</description>
    </variable>
    <variable>
        <type>int</type>
        <name>grass_countdown</name>
        <description>Number of iterations till grass can re-grow</description>
    </variable>
</memory>
```

5.3 Agent functions

The number of functions or states the agents have 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 below in Table 1, they are illustrated in Section 9.

Agent(s)	**Function**	**Description**
Wolves & Sheep	move	Update the agent's heading, move the agent and reduce the amount of energy by 1 unit.
Wolves	choose_dinner	Wolves look to see if there is a sheep near them, if so they order sheep for dinner.
Wolves	eat_dinner	Wolves check whether they have a sheep to eat.
Wolves & Sheep	die	Agent removed from simulation - only called for agents with no energy.
Wolves & Sheep	reproduce	The chance to create a new agent.
Sheep	post_position	Post the position of the sheep.
Sheep	eat_grass	Check to see if able to eat grass and if so, increase energy accordingly.
Sheep	am_dinner	Check to see if will be eaten and if so, remove self from simulation.
Grass	grass_eaten	Read sheep locations to find out if will be eaten - only called for grass agents which are green.
Grass	grow_grass	Check to see if can re-grow - only called for grass agents which are brown.

Table 1: Summary of the agent functions

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 model are given below, see for example the function `reproduce`.

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 for a wolf agent type are:

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>
    <description></description>
    <currentState>start</currentState>
    <nextState>1</nextState>
</function>
```

choose_dinner: This function reads the *location* message board to find out if there is a sheep nearby. If there is they order the sheep for dinner posting the IDs of both the sheep and wolf to the *order* message board.

```
<function>
    <name>choose_dinner</name>
    <description></description>
    <currentState>1</currentState>
    <nextState>2</nextState>
    <inputs>
        <input>
            <messageName>location</messageName>
        </input>
    </inputs>
    <outputs>
        <output>
            <messageName>order</messageName>
        </output>
    </outputs>
</function>
```

eat_dinner: This function reads the *sheep_food* message board to find out if the sheep they ordered in `choose_dinner` is available for eating. If it is, the wolf increases its energy appropriately.

```
<function>
    <name>eat_dinner</name>
    <description></description>
    <currentState>2</currentState>
    <nextState>3</nextState>
    <inputs>
        <input>
            <messageName>sheep_food</messageName>
        </input>
    </inputs>
</function>
```

reproduce: This function decides whether a new agent will be created. The `<condition>` tag specifies that this function is only called when the `wolf_energy` memory variable of an agent is greater or equal to 0, i.e. the agent will not die this iteration.

```
<function>
    <name>reproduce</name>
    <description></description>
    <currentState>3</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.wolf_energy</value>
        </lhs>
        <op>GEQ</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
```

die: The agent is to die and so this function returns 1. This indicates that the framework should perform garbage collection on the agent memory. This time the `<condition>` tag ensures this function is only called for agents with an amount of energy which is less than 0.

```
<function>
    <name>die</name>
    <description></description>
    <currentState>3</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.wolf_energy</value>
        </lhs>
        <op>LT</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
```

The XMML function definitions for a sheep agent type are:

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

```
<function>
<name>move</name>
    <description></description>
    <currentState>start</currentState>
    <nextState>1</nextState>
</function>
```

post_position: This function simply posts the position of the sheep agent to the *location* message board.

```
<function>
    <name>post_position</name>
    <description></description>
    <currentState>1</currentState>
    <nextState>2</nextState>
    <outputs>
        <output>
            <messageName>location</messageName>
```

```
        </output>
    </outputs>
</function>
```

eat_grass: This function reads the `grass_food` message board to find out if there is grass on their patch. If there is, the sheep increases its energy accordingly.

```
<function>
    <name>eat_grass</name>
    <description></description>
    <currentState>2</currentState>
    <nextState>3</nextState>
    <inputs>
        <input>
            <messageName>grass_food</messageName>
        </input>
    </inputs>
</function>
```

am_dinner: This function reads the *order* message board to find out if their ID is present. If it is, they look for potential conflict and ensure that only 1 wolf can eat them. They post the ID of the winning wolf to the *sheep_food* message board.

```
<function>
    <name>am_dinner</name>
    <description></description>
    <currentState>3</currentState>
    <nextState>4</nextState>
    <inputs>
        <input>
            <messageName>order</messageName>
        </input>
    </inputs>
    <outputs>
        <output>
            <messageName>sheep_food</messageName>
        </output>
    </outputs>
</function>
```

reproduce: This function decides whether a new agent will be created. The `<condition>` tag specifies that this function is only called when the `sheep_energy` memory variable of an agent is greater or equal to 0, i.e. the agent will not die this iteration.

```
<function>
    <name>reproduce</name>
    <description></description>
    <currentState>4</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.sheep_energy</value>
        </lhs>
        <op>GEQ</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
```

die: The agent is to die and so this function returns 1. This indicates that the framework should perform garbage collection on the agent memory. This time the `<condition>` tag ensures this function is only called for agents with an amount of energy which is less than 0.

```
<function>
    <name>die</name>
    <description></description>
    <currentState>4</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.sheep_energy</value>
        </lhs>
        <op>LT</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
```

The XMML function definitions for a grass agent type are:

grass_eaten: This function reads the *location* message board to find out if there is a sheep nearby. If there is, they post the ID of the sheep to the `grass_food` message board. This time the `<condition>` tag ensures this function is only called for grass agents which are green, i.e. `grass_colour` = 1.

```
<function>
    <name>grass_eaten</name>
    <description></description>
    <currentState>start</currentState>
    <nextState>1</nextState>
    <condition>
        <lhs>
            <value>a.grass_colour</value>
        </lhs>
        <op>EQ</op>
        <rhs>
            <value>1</value>
        </rhs>
    </condition>
    <inputs>
        <input>
            <messageName>location</messageName>
        </input>
    </inputs>
    <outputs>
        <output>
            <messageName>grass_food</messageName>
        </output>
    </outputs>
</function>
```

idle: Indicates that grass agents which are brown, i.e. `grass_colour` = 0, should do nothing during this state transition.

```
<function>
    <name>idle</name>
    <description></description>
```

```
    <currentState>start</currentState>
    <nextState>1</nextState>
     <condition>
        <lhs>
            <value>a.grass_colour</value>
        </lhs>
        <op>EQ</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
```

grow_grass: This function checks to see if the grass agent has been waiting long enough to re-grow. If it has, agent colour will turn green, if not, the time waiting will be incremented by 1. This time the `<condition>` tag ensures this function is only called for grass agents which are brown, i.e. `grass_colour` = 0.

```
<function>
    <name>grow_grass</name>
    <description></description>
    <currentState>1</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.grass_colour</value>
        </lhs>
        <op>EQ</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
```

idle: Indicates that grass agents which are green, i.e. `grass_colour` = 1, should do nothing during this state transition.

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

It would be possible to combine some functions into single functions, for example we have brown grass agents idle, whilst green ones call `grass_eaten`. Then after state 1, we have green grass agents idle whilst brown ones call `grow_grass`. It would seem sensible to simply have only 1 transition from `start` state to `end` state, between which brown agents call `grow_grass` and green agents call `grass_eaten`. The reason we have not done this though is because with the current

implementation it more closely matches the NetLogo version and hence model validation will be more reliable. 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 the full model with grass included we define 4 message types: *location*, *order*, *sheep_food*, *grass_food*. These provide all the information required by agents to implement the interactions. The XMML message board definitions are:

```
<messages>
    <message>
        <name>location</name>
        <description></description>
        <variables>
            <variable>
                <type>int</type>
                <name>sheep_id</name>
            </variable>
            <variable>
                <type>double</type>
                <name>sheep_x</name>
            </variable>
            <variable>
                <type>double</type>
                <name>sheep_y</name>
            </variable>
        </variables>
    </message>
    <message>
        <name>order</name>
        <description>Wolf ID and the ID of the sheep they'd like to eat</description>
        <variables>
            <variable>
                <type>int</type>
                <name>wolf_id</name>
            </variable>
            <variable>
                <type>int</type>
                <name>sheep_id</name>
            </variable>
        </variables>
    </message>
    <message>
        <name>sheep_food</name>
        <description>ID of wolf who has eaten dinner</description>
        <variables>
            <variable>
                <type>int</type>
                <name>wolf_id</name>
            </variable>
        </variables>
    </message>
    <message>
        <name>grass_food</name>
        <description>ID of sheep who has eaten grass</description>
        <variables>
```

```
            <variable>
                <type>int</type>
                <name>sheep_id</name>
            </variable>
        </variables>
    </message>
</messages>
```

The complete XMML model for both versions of the 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. In 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 referenced throughout the simulation code. See Section 5.1 for the environment definition in the predation 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 : Similarly to accessing environment data, variables defined in the agent memory section in the model definition, once parsed, are available to the modeller as C macros. They are defined in header file for each agent, for example this would be `wolf_agent_header.h` for wolves.

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. For example, to write information to the *location* message board in the predation model, FLAME provides the function:

```
void add_location_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. For example, one of the message boards in the predation model is `location` 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 predation model determines that an agent should die, either due to lack of energy or having been eaten, 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 C macros, as described in Section 6.

In this, and the preceding sections we have only given a description of the essential components of the FLAME model. Appendices A and B give the complete model files for versions both with and without grass.

8 Parsing the FLAME model

With the model defined in XMML and the agent functions written, the fully specified 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

wolf_agent_header.h, sheep_agent_header.h, grass_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 modeller 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. This graph shows all the agents and their functions in the model, together with all the defined message boards. Figures 2 and 3 show the graph for the predation model without and with grass respectively.

The dependency graphs provide a very good visual check on the structure of the model.

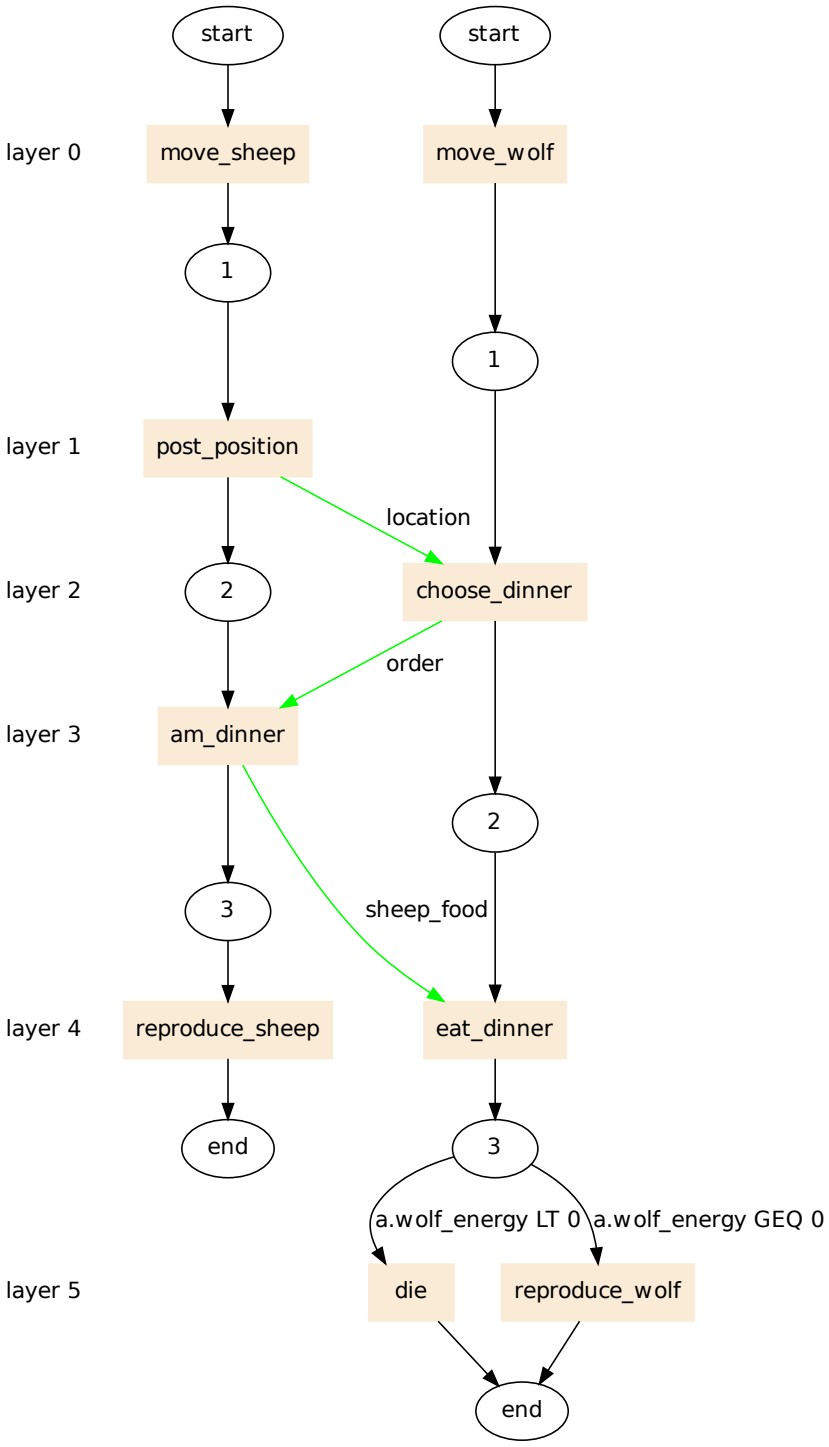


Figure 2: The *task dependency graph* of the predation model without grass.

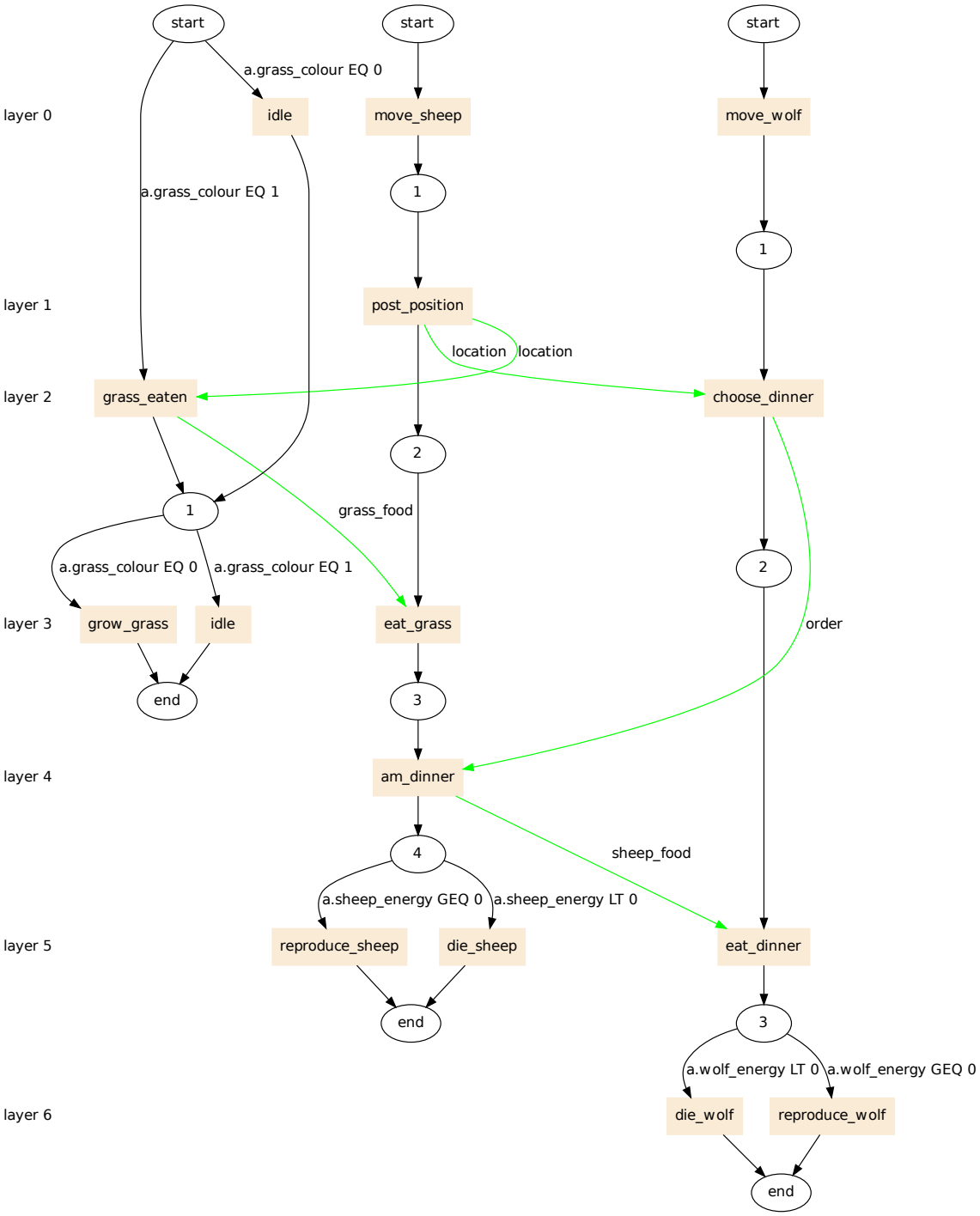


Figure 3: The *task dependency graph* of the predation model with grass.

10 Input data generation

The initial data for the model can be generated using various tools but we choose to write a simple python program for the task. The program generates the specified number of wolf and sheep agents in random positions with random headings in a given domain. In addition each wolf is given an amount of energy which is randomly selected from the interval $[0, 39]$. This interval is chosen to align the initial data as closely to the NetLogo implementation as possible.

In the case where grass is included, sheep agents require an energy variable also and this is computed similarly to wolves but from the interval $[0, 7]$. The grass agents are now generated, one for each grid point within the domain. Whether a grass is "green" or "brown" is chosen at random for each grass agent. The countdown variable, which is a variable in a grass agent's memory, represents how many iterations a "brown" grass agent must wait, before being able to "regrow" and turn "green". In the intitial data, therefore, grass agents which are "green" have their corresponding countdown variable set to 30, which is the maximum value chosen for our simulation. A random value in the interval $[0, 29]$ is selected for all grass agents which are "brown".

All environment variables are hard-coded into the file but can be easily changed. The program generates an `0.xml` output file containing the initial data.

11 Testing

11.1 Setup

The method for testing this implementation of a simple predation model is to compare it against the results from the NetLogo model with the same parameters. The parameters for the NetLogo and FLAME model are given in Table 2.

Parameter	**Value**
Domain width	50
Domain height	50
Initial number of wolf agents	100
Initial number of sheep agents	50
Amount of energy gained from eating sheep	20
Amount of energy gained from eating grass	4
Probability of wolves reproducing	0.05
Probability of sheep reproducing	0.04
Number of iterations before grass can re-grow	30

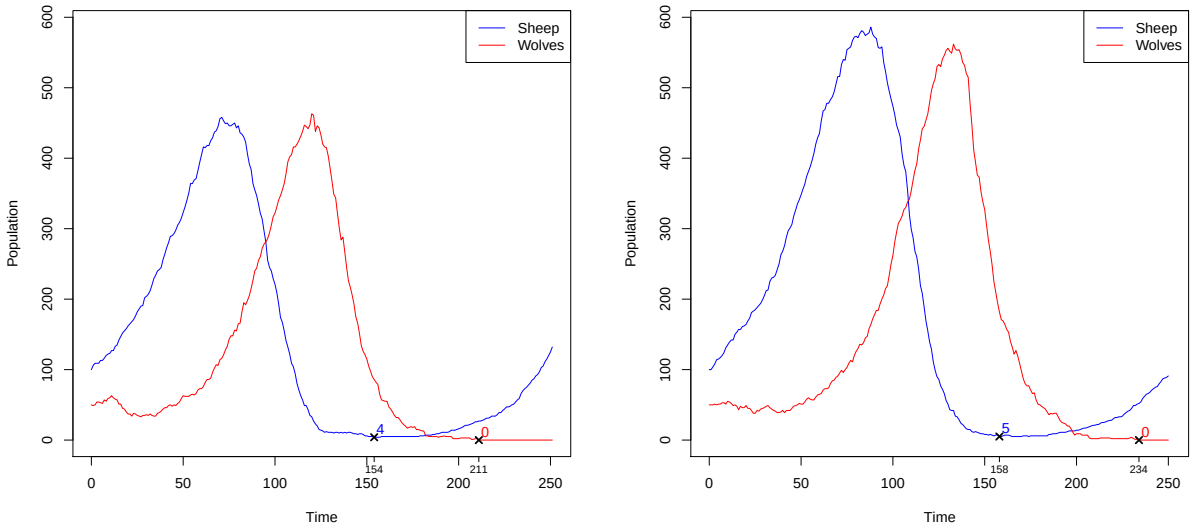
Table 2: The model parameters used in comparison of NetLogo and FLAME implementations.

11.2 Model without grass

When grass is not included in the model, the environment is generally not sustainable. Simulations show that the pattern is always similar up to approximately 200 iterations when the numbers of both wolves and sheep become very low. There are then two possible outcomes:

1. The wolves die out due to the number of sheep being too low, but a small number of sheep just manage to stay alive and hence continue then to multiply exponentially for the rest of the simulation. An example of this is illustrated in Figure 4.

2. Sheep become extinct and therefore very soon after the wolves follow accordingly. See Figure 5 for an example of this.



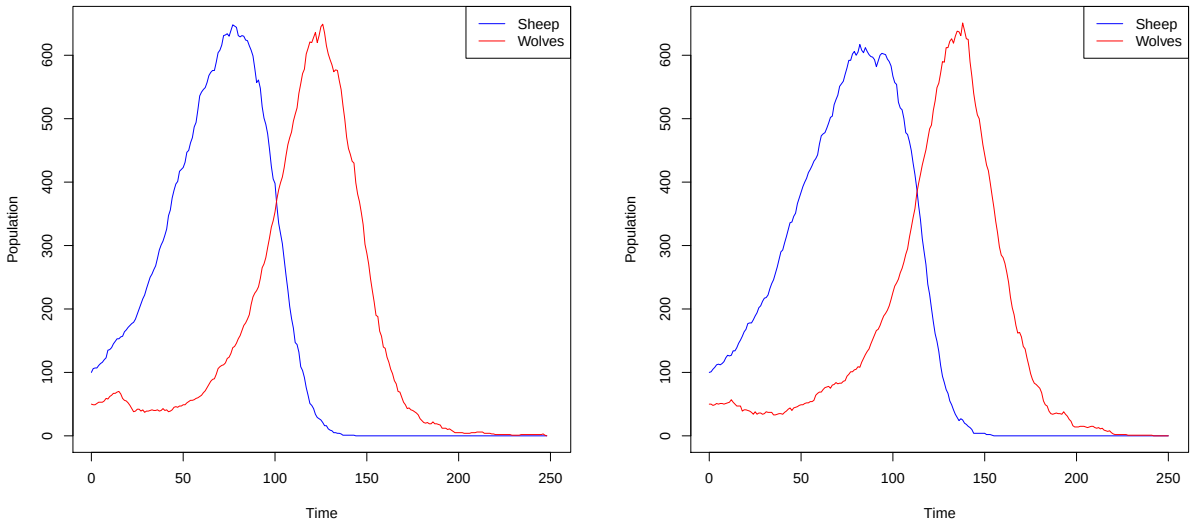
(a) Results from NetLogo implementation

(b) Results from FLAME implementation

Figure 4: Examples of simulations where sheep survive but wolves become extinct. There is no grass included in this model.

Figures 4a and 4b show comparative simulations for the case where the wolves die out, but the sheep continue to grow exponentially. We can see that the 2 plots show a very similar pattern with lowest sheep populations of 4 and 5 at iterations 154 and 158 from NetLogo and FLAME respectively. Similarly the wolves become extinct at iterations 211 and 234 from NetLogo and FLAME respectively. The only noteable difference between the implementations appears to be the maximum values for the populations: values from FLAME being higher than the corresponding NetLogo ones. Repeated simulations in NetLogo though show that the maximum population size differs due to the random effects in the model and therefore the difference observed in Figure 4 is not thought to be significant.

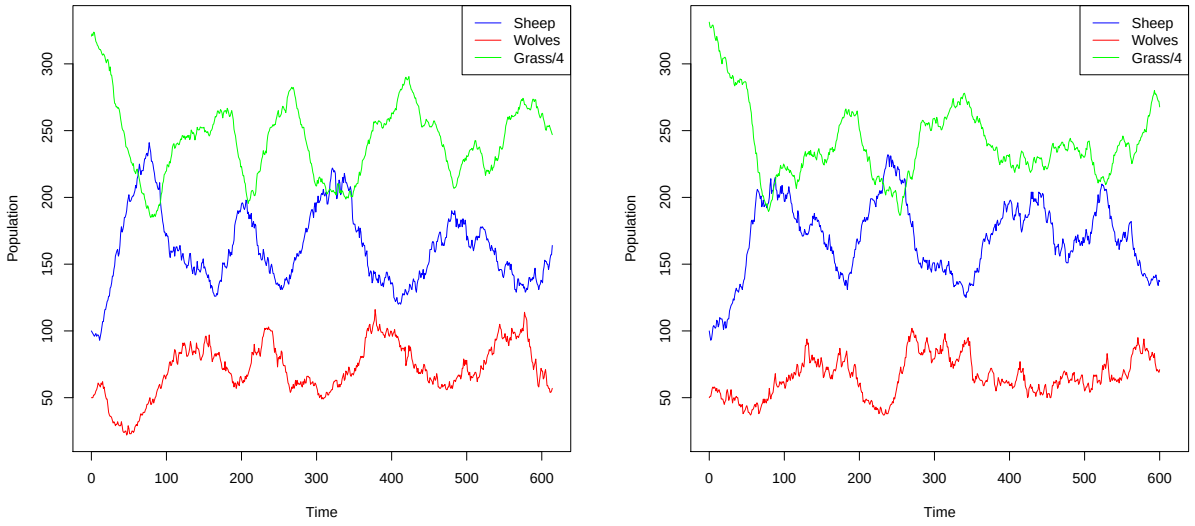
Figures 5a and 5b show comparative simulations in NetLogo and FLAME respectively for the case where both species become extinct. Again the 2 plots show a very similar pattern with maximum values close and the times at which the populations become extinct also similar.



(a) Results from NetLogo implementation

(b) Results from FLAME implementation

Figure 5: Examples of simulations where both sheep and wolves become extinct. There is no grass included in this model.



(a) Results from NetLogo implementation

(b) Results from FLAME implementation

Figure 6: Examples of simulations with grass included in the model.

11.3 Model with grass

The version of the model when grass is included is generally sustainable with all 3 species able to live in harmony as the iterations continue. Figures 6a and 6b show comparative simulations in NetLogo and FLAME respectively. It is clear that the results are similar between the NetLogo and the FLAME implementations.

All simulations, as illustrated in Figures 4, 5 and 6, were produced using the same values for the model parameters, as given in Table 2. Clearly there is a random element in the model though and this must be taken into consideration when comparing results from different implementations.

12 Parallel implementation

12.1 Design

Sections 1 to 11 focus on describing the predator model and its implementation in series, in order to accurately compare the FLAME implementation to that of NetLogo. With FLAME though, the modeller has the option to run a simulation in parallel (using MPI) and in this section we investigate how the predator model can be successfully and efficiently partioned and run on high performance computers.

Before we describe how to parse and run the model in parallel we need to consider whether the design of the model is still appropriate. Of course, normally we would consider the implications of running in parallel when initially designing the model. As this model, however, has been written primarily to compare FLAME with NetLogo, the main purpose was to implement the model in series.

When new sheep and wolf agents are created, in the `reproduce` functions, a unique `ID` is required for the new agent. When running in series the most straightforward method of doing this is to keep track of the total number of each type of agent, and add one to these integers each time an additional wolf or sheep agent is created. We implemented this in the series version with the variables `GLOBAL_wolf_ids` and `GLOBAL_sheep_ids`, see Appendices B.1 and B.2 for details. In parallel, however, this is not as straightforward as each processor will have its own value for `GLOBAL_wolf_ids` and `GLOBAL_sheep_ids` and therefore additional wolf and sheep agents will not have unique identifiers. We tackle this issue by writing a `create_agent_id` function which

returns an integer trusted to be unique for this application. The `reproduce` functions then call `create_agent_id` when required. How `create_agent_id` is implemented can vary of course, but we choose to use a combination of the process ID and the current time. The current time is represented by 2 integers: the number of seconds and microseconds since the epoch. Care is taken to ensure that each subsequent time `create_agent_id` is called, at least 1 microsecond has passed, thus ensuring unique times. Utilising the process ID in addition, allows for the possiblity that more than one process could execute `create_agent_id` at the same microsecond. The process ID and number of seconds and mircoseconds now need to be combined and converted into one unique integer. We do this by concatonating the integers into a string and then applying a hash function to the string, which returns an integer. We can now trust to a reasonable degree of certainity that this resulting integer will be unique and can therefore use it as the ID for the new wolf or sheep.

12.2 Initial data partition

The FLAME parser, as described in Section 8, takes a parameter which controls whether the model will be parsed in series or parallel mode. The default is serial and modellers wishing to take advantage of the parallel features should specify `-p` when running the parser. See the FLAME user manual for more detail.

As with the parser, to run the FLAME model in parallel, some additional parameters are required. These include the number of processors to use and how to partition the initial data so that each processor has a roughly equivalent workload. Figure 7 shows the initial data used in Section 11, in the version where grass is included.

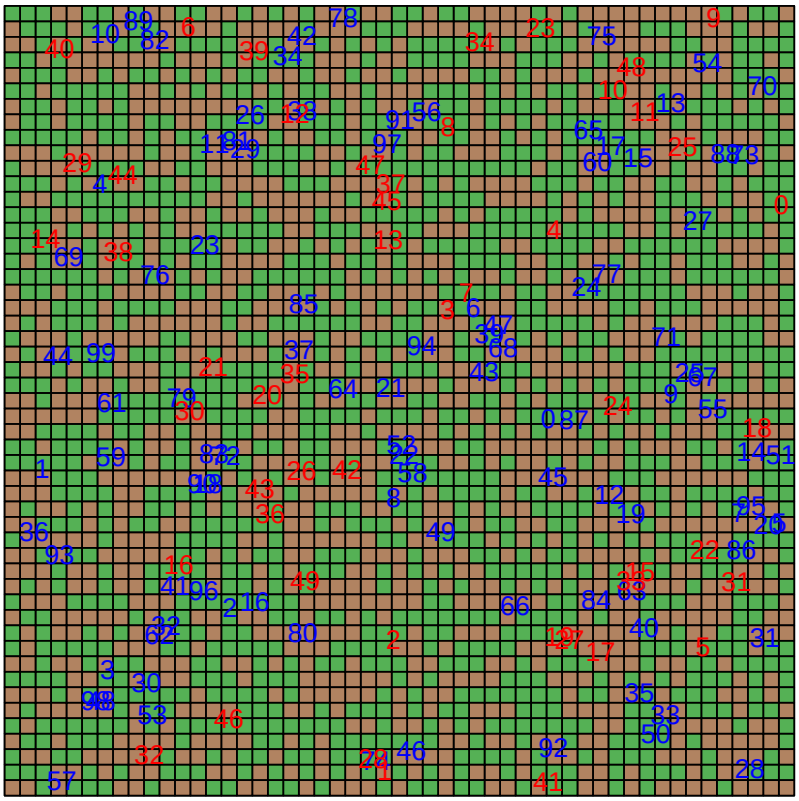


Figure 7: Initial data. Blue text shows the ID of the sheep and red text shows the ID of the wolves. Green and brown cells show whether grass is present or not respectively. Note that although the domain is toroidal it is depicted here as a square.

The default method for partioning the data is to split up the agents geometrically between all the processors. This is also specified at run time by the parameter `-g`. For the initial data shown in Figure 7 and with 4 processors specified using the option `-np 4`, the resulting data partioned using the geometric approach is illustrated in Figure 8.

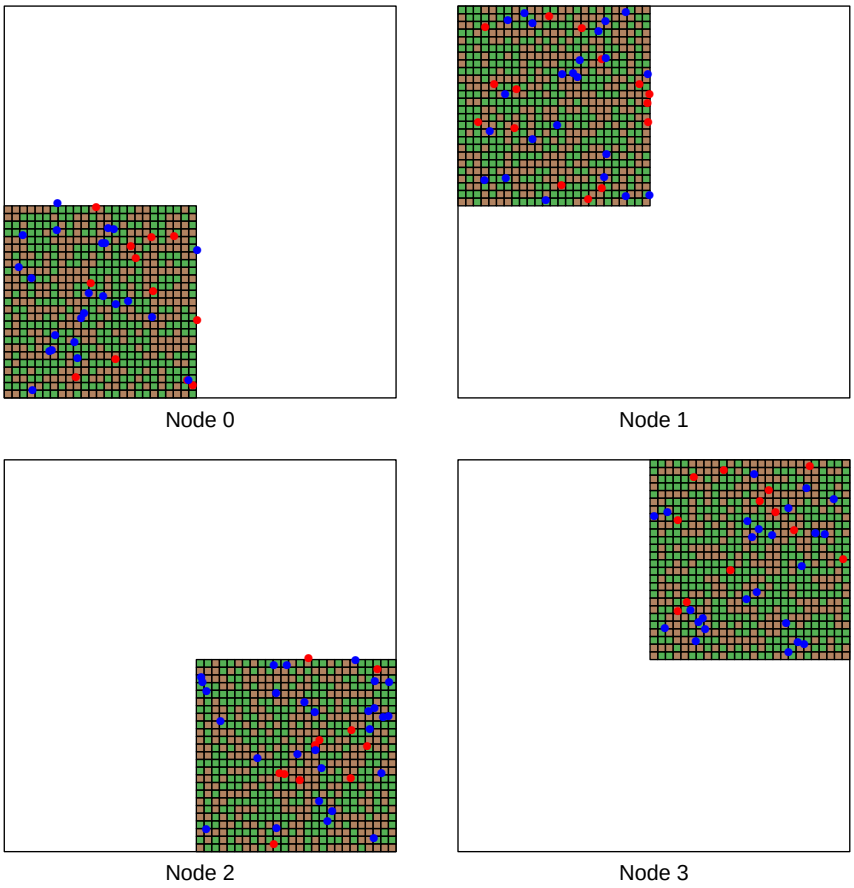


Figure 8: Geometric partition with wolves shown as red circles, and sheep shown as blue. If a grass agent is present for a particular node this is shown as green or brown depending on whether the grass agent currently has grass available.

If preferred, an alternative method to geometric partioning is the round-robin approach. This simply allocates the first of each agent to node 0, the second of each agent to node 1 and so on. The first agent is defined as the first agent specified in the input file, `0.xml`. To employ this method specify parameter `-r` at run time. For the initial data shown in Figure 7 and with 4 processors specified using the option `-np 4`, the resulting data partioned using the round-robin method is illustrated in Figure 9. Note that in both Figures 8 and 9 the IDs of the individual agents have been replaced with circles for ease of viewing.

In summary, to run the model in parallel, on 4 processors, using geometric partioning the following command should be used:

```
mpirun -np 4 ./main its/0.xml -g
```

depending on the location of the input file and assuming the MPI implementation, OpenMPI, is installed. Similarly for the same setup but with round-robin pationing this would be:

```
mpirun -np 4 ./main its/0.xml -r
```

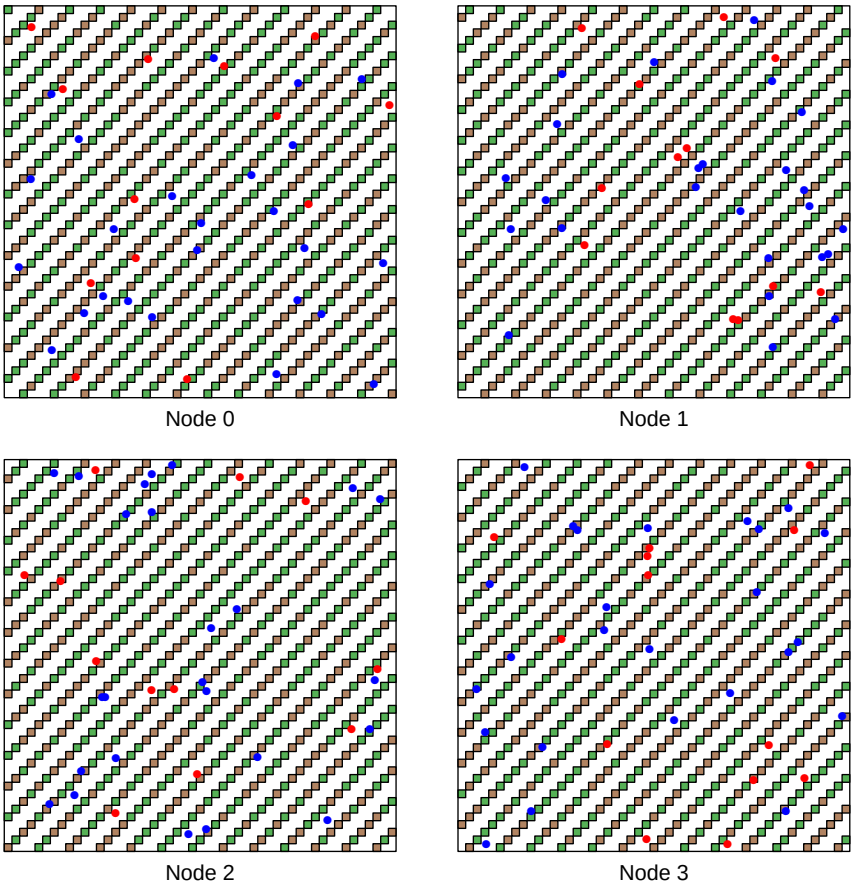


Figure 9: Round robin partition with wolves shown as red circles, and sheep shown as blue. If a grass agent is present for a particular node this is shown as green or brown depending on whether the grass agent currently has grass available.

12.3 Results

To test that the parallel implementation produces comparable results to the series version, the model with grass is run across 4 processors using both partioning methods. Figure 10 shows the populations of all 3 agents types as the number of iterations increase in a parallel simulation. Firstly, it should be noted that the general trends are the same as the execution in series, with all 3 species exisiting in harmony. In addition there are similar population sizes at each iteration and similar variation between agents. This is to be expected as running the model in parallel should not, of course, produce a different outcome. It is clear however, that the model when run in parallel does not produce identical results to that of the model when run in series. In addition, results are not identical between methods of partioning. Although there are random elements in the model, for example the direction a wolf agent chooses to move in each iteration, the seed has been set to the same value for both parallel simulations as was used when running the model in series. This was specified to increase consistency across experiments, however as each processor has the same value for the seed, each will be working along the same set of random numbers for the various decisions. Consider the set of random numbers $\{r_1, r_2, r_3, \ldots\}$. The first decision to be made is from the function `move_wolf`. When the model is run in series, Wolf_1 moves in the direction indicated by r_1. The second decision then results in Wolf_2 moving in the direction indicated by r_2 and decision making continues in this way. When run in parallel with round-robin partioning, Wolf_1 is assigned to Node_0 and Wolf_1 therefore moves in the direction indicated by r_1, the same as when run in series. Wolf_2 is assigned to Node_1 and as this is

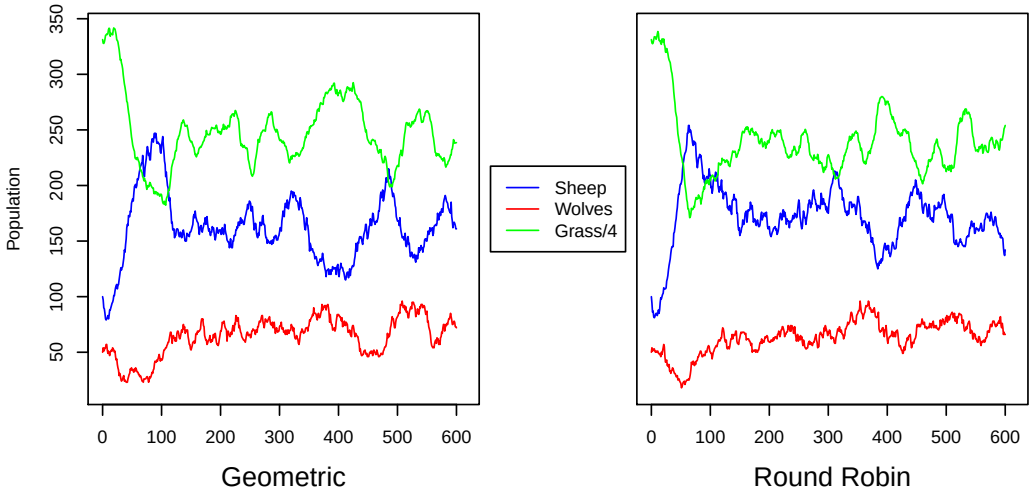


Figure 10: Simulation results when model is run in parallel across 4 processors.

the first decision for this node, the first random value will be chosen and Wolf_2 moves in the direction indicated by r_1. Geometric partioning will of course result in a different permutation to round-robin as a different set of agents will be assigned to each node. It is clear, therefore, that with the seed set to the same value for each processor we should expect non-identical results between series and different methods of partioning the initial data. Equally, running on different numbers of processors should result in non-identical results. The model, with the parameters as set in Table 2, should always produce a sustainable environment though, regardless of the number of processors or the method of partioning the initial data.

We now repeat the parallel simulations with the same initial data, seed and number of processors as previously used. Figure 11 shows the results of this experiment repeated twice for each method of partioning the data. When running in series, if the initial data and seed are kept the same the model is determinstic, that is if the model is run twice, identical results will be produced. We see from Figure 11 that results are not deterministic for parallel simulations. This is because although the same random numbers are selected for each decision, there is some varation in the order that messages are posted to the message boards. This is now discussed in more detail.

There are potential conflicts in the model, for example it is possible that 1 wolf and 2 sheep may all be on the same patch. In this case only one sheep can be eaten by the wolf, as explained in Section 3. There are different ways of resolving this conflict but we took the following approach. The wolf in question reads the location message board and upon finding the second sheep on its patch, it chooses randomly whether to remain with the first sheep it found, or change to the new sheep. This is implemented in the function `choose_dinner`, see Appendix B.1 for details. The random element in this function is of course consistent as we have set the seed to the same value and used the same method of partioning the initial data across the 4 processors. We need to now consider the order that sheep appear on the location message board. Sheep post their x and y values to the location board in the function `post_position`, see Appendix B.2. When run in parallel, each processor has its own version of the location board whilst executing the `post_position` function for whichever sheep agents assigned to that particular processor. When the model reaches a synchronisation point, the data on each of the 4 location message boards are shared between the processors in the order in which the processors reach that point. In this way the order which the sheep appear on each location board will firstly, be different for each processor, and secondly, be in one of 6 (3!) orders for a particular processor. For example, for

Node_0, the order of the location board would be the sheep assigned to Node_0 first, and then either those assigned to Node_1, Node_2, Node_3, or Node_1, Node_3, Node_2, etc. The order within nodes will remain consistent. If it happens that the 2 sheep which are on the same patch as the single wolf are assigned to different processors, then which sheep becomes the "first" sheep will of course differ, depending on which processor returns first. In this way it is possible for the sheep chosen by the wolf to differ, even with the random number remaining the same. Only 1 instance of this is required to then slightly alter the rest of the popultation sizes resulting in non-deterministic results for parallel simulations.

There is a feature in the FLAME framework that we have not discussed which eliminates this source of variability. It is possible to add a `sort` tag in the xml when a function has a message board input. This will require FLAME to sort the messages on a board into a particular order as specified by the message variable in the sort tag. This is useful when testing and applying the sort tag to each message board input does result in deterministic parallel simulations. Obviously though we would not adopt this feature in this case generally, as we would require truely random results to the decisions.

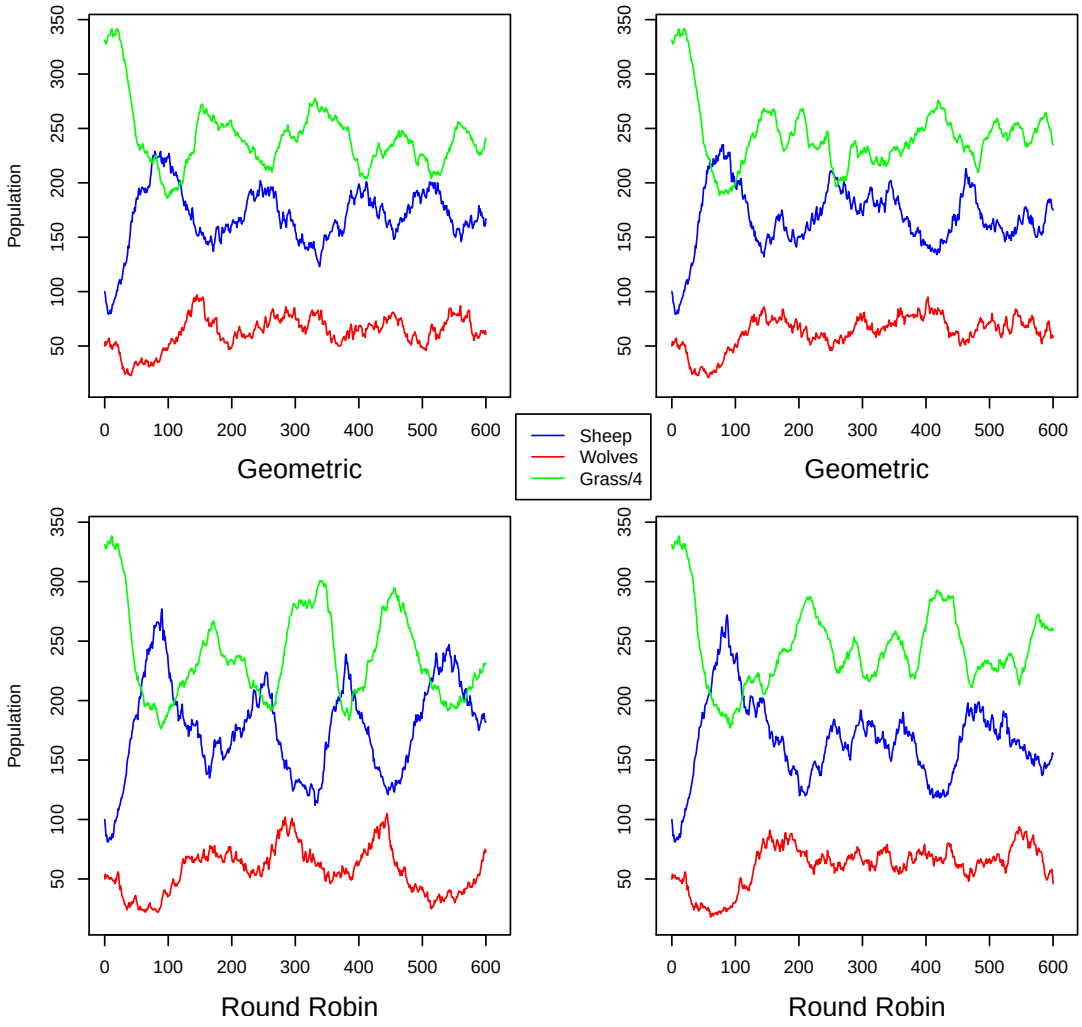


Figure 11: Simulations presented in Figure 10 repeated twice more here.

13 Comments on implementation

The FLAME implemention of the predation model has been designed to mimic that of NetLogo and to attempt to be as straightforward as possible. There is an additonal feature available in FLAME called "message filters" though which we could have chosen to make use of. For each message board input defined in the xml, a `filter` tag could be applied which consists of a message variable and an agent variable. The filter then only allows the function to read messages where the relevant message variable is equal to the agent variable specified. In this way, performing a loop to go through the whole of the location message board to check which sheep are on a wolf's partiuclar patch can be avoided. Thus a more efficient implementation is achieved. See the FLAME manual for more detail on message filters.

We have one final point concerning the model itself. Netlogo uses patches to determine the agents with which another agent can possibly interact. This is fine for a model where patches arise naturally but in this predation 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 predation is missed because the wolf is not on the same patch as the sheep despite being very close in euclidean distance.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] U. Wilensky (1997). NetLogo Wolf Sheep Predation model. `http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation/`. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[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

A.1 FLAME XMML Model without grass

```
<?xml version="1.0" encoding="UTF-8"?>
<xmodel version="2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://flame.ac.uk/schema/xmml_v2.xsd">

    <name>predator</name>
    <version>0.1</version>
    <author>GLP</author>
    <description>Implementation of NetLogo's Wolf-Sheep Predation model</description>
    <environment>
        <constants>
            <variable>
                <type>double</type>
                <name>reproduce_sheep_prob</name>
                <description></description>
            </variable>
            <variable>
                <type>double</type>
                <name>reproduce_wolf_prob</name>
                <description></description>
            </variable>
            <variable>
                <type>int</type>
                <name>gain_from_food_wolf</name>
                <description></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>
        <functionFiles>
            <file>sheep_functions.c</file>
            <file>wolf_functions.c</file>
        </functionFiles>
    </environment>
    <agents>
        <xagent>
            <name>wolf</name>
            <description></description>
            <memory>
                <variable>
                    <type>int</type>
                    <name>wolf_id</name>
                    <description></description>
                </variable>
                <variable>
                    <type>double</type>
                    <name>wolf_x</name>
                    <description></description>
                </variable>
                <variable>
                    <type>double</type>
                    <name>wolf_y</name>
                    <description></description>
```

```
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_heading</name>
        <description></description>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_energy</name>
        <description></description>
    </variable>
</memory>
<functions>
    <function>
        <name>move_wolf</name>
        <description></description>
        <currentState>start</currentState>
        <nextState>1</nextState>
    </function>
    <function>
        <name>choose_dinner</name>
        <description></description>
        <currentState>1</currentState>
        <nextState>2</nextState>
        <inputs>
            <input>
                <messageName>location</messageName>
            </input>
        </inputs>
        <outputs>
            <output>
                <messageName>order</messageName>
            </output>
        </outputs>
    </function>
    <function>
        <name>eat_dinner</name>
        <description></description>
        <currentState>2</currentState>
        <nextState>3</nextState>
        <inputs>
            <input>
                <messageName>sheep_food</messageName>
            </input>
        </inputs>
    </function>
    <function>
        <name>die</name>
        <description></description>
        <currentState>3</currentState>
        <nextState>end</nextState>
        <condition>
            <lhs>
                <value>a.wolf_energy</value>
            </lhs>
            <op>LT</op>
            <rhs>
                <value>0</value>
            </rhs>
        </condition>
    </function>
    <function>
        <name>reproduce_wolf</name>
        <description></description>
```

```
            <currentState>3</currentState>
            <nextState>end</nextState>
            <condition>
                <lhs>
                    <value>a.wolf_energy</value>
                </lhs>
                <op>GEQ</op>
                <rhs>
                    <value>0</value>
                </rhs>
            </condition>
        </function>
    </functions>
</xagent>
<xagent>
    <name>sheep</name>
    <description></description>
    <memory>
        <variable>
            <type>int</type>
            <name>sheep_id</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_x</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_y</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_heading</name>
            <description></description>
        </variable>
    </memory>
    <functions>
        <function>
            <name>move_sheep</name>
            <description></description>
            <currentState>start</currentState>
            <nextState>1</nextState>
        </function>
        <function>
            <name>post_position</name>
            <description></description>
            <currentState>1</currentState>
            <nextState>2</nextState>
            <outputs>
                <output>
                    <messageName>location</messageName>
                </output>
            </outputs>
        </function>
        <function>
            <name>am_dinner</name>
            <description></description>
            <currentState>2</currentState>
            <nextState>3</nextState>
            <inputs>
                <input>
```

```
                        <messageName>order</messageName>
                    </input>
                </inputs>
                <outputs>
                    <output>
                        <messageName>sheep_food</messageName>
                    </output>
                </outputs>
            </function>
            <function>
                <name>reproduce_sheep</name>
                <description></description>
                <currentState>3</currentState>
                <nextState>end</nextState>
            </function>
        </functions>
    </xagent>
</agents>
<messages>
    <message>
        <name>location</name>
        <description></description>
        <variables>
            <variable>
                <type>int</type>
                <name>sheep_id</name>
                <description>ID of sheep</description>
            </variable>
            <variable>
                <type>double</type>
                <name>sheep_x</name>
                <description>Location in x direction of sheep</description>
            </variable>
            <variable>
                <type>double</type>
                <name>sheep_y</name>
                <description>Location in y direction of sheep</description>
            </variable>
        </variables>
    </message>
    <message>
        <name>order</name>
        <description>Wolf id and the id of the sheep they'd like to eat</description>
        <variables>
            <variable>
                <type>int</type>
                <name>wolf_id</name>
                <description></description>
            </variable>
            <variable>
                <type>int</type>
                <name>sheep_id</name>
                <description></description>
            </variable>
        </variables>
    </message>
    <message>
        <name>sheep_food</name>
        <description>ID of wolf who has eaten dinner</description>
        <variables>
            <variable>
                <type>int</type>
                <name>wolf_id</name>
                <description></description>
```

```
            </variable>
        </variables>
    </message>
</messages>
</xmodel>
```

A.2 FLAME XMML Model with grass

```
<?xml version="1.0" encoding="UTF-8"?>
<xmodel version="2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://flame.ac.uk/schema/xmml_v2.xsd">

    <name>predator</name>
    <version>0.2</version>
    <author>GLP</author>
    <description>Implementation of NetLogo's Wolf-Sheep Predation model</description>
    <environment>
        <constants>
            <variable>
                <type>double</type>
                <name>reproduce_sheep_prob</name>
                <description></description>
            </variable>
            <variable>
                <type>double</type>
                <name>reproduce_wolf_prob</name>
                <description></description>
            </variable>
            <variable>
                <type>int</type>
                <name>gain_from_food_wolf</name>
                <description></description>
            </variable>
            <variable>
                <type>int</type>
                <name>gain_from_food_sheep</name>
                <description></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>
            <variable>
                <type>int</type>
                <name>grass_regrowth</name>
                <description>Number of iterations before grass can re-grow</description>
            </variable>
        </constants>
        <functionFiles>
            <file>sheep_functions.c</file>
            <file>wolf_functions.c</file>
            <file>grass_functions.c</file>
        </functionFiles>
    </environment>
    <agents>
        <xagent>
            <name>wolf</name>
```

```
<description></description>
<memory>
    <variable>
        <type>int</type>
        <name>wolf_id</name>
        <description></description>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_x</name>
        <description></description>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_y</name>
        <description></description>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_heading</name>
        <description></description>
    </variable>
    <variable>
        <type>double</type>
        <name>wolf_energy</name>
        <description></description>
    </variable>
</memory>
<functions>
    <function>
        <name>move_wolf</name>
        <description></description>
        <currentState>start</currentState>
        <nextState>1</nextState>
    </function>
    <function>
        <name>choose_dinner</name>
        <description></description>
        <currentState>1</currentState>
        <nextState>2</nextState>
        <inputs>
            <input>
                <messageName>location</messageName>
            </input>
        </inputs>
        <outputs>
            <output>
                <messageName>order</messageName>
            </output>
        </outputs>
    </function>
    <function>
        <name>eat_dinner</name>
        <description></description>
        <currentState>2</currentState>
        <nextState>3</nextState>
        <inputs>
            <input>
                <messageName>sheep_food</messageName>
            </input>
        </inputs>
    </function>
    <function>
        <name>die_wolf</name>
```

```
            <description></description>
            <currentState>3</currentState>
            <nextState>end</nextState>
            <condition>
                <lhs>
                    <value>a.wolf_energy</value>
                </lhs>
                <op>LT</op>
                <rhs>
                    <value>0</value>
                </rhs>
            </condition>
        </function>
        <function>
            <name>reproduce_wolf</name>
            <description></description>
            <currentState>3</currentState>
            <nextState>end</nextState>
            <condition>
                <lhs>
                    <value>a.wolf_energy</value>
                </lhs>
                <op>GEQ</op>
                <rhs>
                    <value>0</value>
                </rhs>
            </condition>
        </function>
    </functions>
</xagent>
<xagent>
    <name>sheep</name>
    <description></description>
    <memory>
        <variable>
            <type>int</type>
            <name>sheep_id</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_x</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_y</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_heading</name>
            <description></description>
        </variable>
        <variable>
            <type>double</type>
            <name>sheep_energy</name>
            <description></description>
        </variable>
    </memory>
    <functions>
        <function>
            <name>move_sheep</name>
            <description></description>
```

```
    <currentState>start</currentState>
    <nextState>1</nextState>
</function>
<function>
    <name>post_position</name>
    <description></description>
    <currentState>1</currentState>
    <nextState>2</nextState>
    <outputs>
        <output>
            <messageName>location</messageName>
        </output>
    </outputs>
</function>
<function>
    <name>eat_grass</name>
    <description></description>
    <currentState>2</currentState>
    <nextState>3</nextState>
    <inputs>
        <input>
            <messageName>grass_food</messageName>
        </input>
    </inputs>
</function>
<function>
    <name>am_dinner</name>
    <description></description>
    <currentState>3</currentState>
    <nextState>4</nextState>
    <inputs>
        <input>
            <messageName>order</messageName>
        </input>
    </inputs>
    <outputs>
        <output>
            <messageName>sheep_food</messageName>
        </output>
    </outputs>
</function>
<function>
    <name>reproduce_sheep</name>
    <description></description>
    <currentState>4</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.sheep_energy</value>
        </lhs>
        <op>GEQ</op>
        <rhs>
            <value>0</value>
        </rhs>
    </condition>
</function>
<function>
    <name>die_sheep</name>
    <description></description>
    <currentState>4</currentState>
    <nextState>end</nextState>
    <condition>
        <lhs>
            <value>a.sheep_energy</value>
```

```
                <lhs>
                <op>LT</op>
                <rhs>
                    <value>0</value>
                </rhs>
            </condition>
        </function>
    </functions>
</xagent>
<xagent>
    <name>grass</name>
    <description></description>
    <memory>
        <variable>
            <type>int</type>
            <name>grass_x</name>
            <description></description>
        </variable>
        <variable>
            <type>int</type>
            <name>grass_y</name>
            <description></description>
        </variable>
        <variable>
            <type>int</type>
            <name>grass_colour</name>
            <description>1 means there is currently grass</description>
        </variable>
        <variable>
            <type>int</type>
            <name>grass_countdown</name>
            <description>Number of iterations till grass can re-grow</description>
        </variable>
    </memory>
    <functions>
        <function>
            <name>grass_eaten</name>
            <description></description>
            <currentState>start</currentState>
            <nextState>1</nextState>
            <condition>
                <lhs>
                    <value>a.grass_colour</value>
                </lhs>
                <op>EQ</op>
                <rhs>
                    <value>1</value>
                </rhs>
            </condition>
            <inputs>
                <input>
                    <messageName>location</messageName>
                </input>
            </inputs>
            <outputs>
                <output>
                    <messageName>grass_food</messageName>
                </output>
            </outputs>
        </function>
        <function>
            <name>idle</name>
            <description></description>
            <currentState>start</currentState>
```

```
                <nextState>1</nextState>
                 <condition>
                    <lhs>
                        <value>a.grass_colour</value>
                    </lhs>
                    <op>EQ</op>
                    <rhs>
                        <value>0</value>
                    </rhs>
                </condition>
            </function>
            <function>
                <name>grow_grass</name>
                <description></description>
                <currentState>1</currentState>
                <nextState>end</nextState>
                <condition>
                    <lhs>
                        <value>a.grass_colour</value>
                    </lhs>
                    <op>EQ</op>
                    <rhs>
                        <value>0</value>
                    </rhs>
                </condition>
            </function>
            <function>
                <name>idle</name>
                <description></description>
                <currentState>1</currentState>
                <nextState>end</nextState>
                 <condition>
                    <lhs>
                        <value>a.grass_colour</value>
                    </lhs>
                    <op>EQ</op>
                    <rhs>
                        <value>1</value>
                    </rhs>
                </condition>
            </function>
        </functions>
    </xagent>
</agents>
<messages>
    <message>
        <name>location</name>
        <description></description>
        <variables>
            <variable>
                <type>int</type>
                <name>sheep_id</name>
                <description>ID of sheep</description>
            </variable>
            <variable>
                <type>double</type>
                <name>sheep_x</name>
                <description>Location in x direction of sheep</description>
            </variable>
            <variable>
                <type>double</type>
                <name>sheep_y</name>
                <description>Location in y direction of sheep</description>
            </variable>
```

```
            </variables>
        </message>
        <message>
            <name>order</name>
            <description>Wolf id and the id of the sheep they'd like to eat</description>
            <variables>
                <variable>
                    <type>int</type>
                    <name>wolf_id</name>
                    <description></description>
                </variable>
                <variable>
                    <type>int</type>
                    <name>sheep_id</name>
                    <description></description>
                </variable>
            </variables>
        </message>
        <message>
            <name>sheep_food</name>
            <description>ID of wolf who has eaten dinner</description>
            <variables>
                <variable>
                    <type>int</type>
                    <name>wolf_id</name>
                    <description></description>
                </variable>
            </variables>
        </message>
        <message>
            <name>grass_food</name>
            <description>ID of sheep who has eaten grass</description>
            <variables>
                <variable>
                    <type>int</type>
                    <name>sheep_id</name>
                    <description></description>
                </variable>
            </variables>
        </message>
    </messages>
</xmodel>
```

B FLAME C Functions

B.1 Wolf functions

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

int GLOBAL_wolf_ids = 0;

/* Move about at random */
int move_wolf()
{
double PI=3.14159267;
    double x, y;
    double heading;

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

    /* Read agent memory */
    x = WOLF_X;
    y = WOLF_Y;
    heading = WOLF_HEADING;

    /* Change heading up to 50 degrees in either direction */
    heading += -50.0 + 100.0 * 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*/
    WOLF_X = x;
    WOLF_Y = y;
    WOLF_HEADING = heading;

    /* Use up 1 unit of energy from the move */
    WOLF_ENERGY--;

    return 0;
}

/*
Wolves look to see if there is a sheep near them, if so they order sheep for dinner
*/
int choose_dinner()
{
    /* Local variables */
    double x1, y1, which_sheep;
    int mboard_sheep_id, sheep_ordered, count;

    count = 0;
    START_LOCATION_MESSAGE_LOOP
```

```
        x1 = location_message->sheep_x;
        y1 = location_message->sheep_y;

        /* Check whether location of sheep came from within a 1x1 patch around me */
        if ( (round(WOLF_X) == round(x1)) && (round(WOLF_Y) == round(y1)))
        {
         mboard_sheep_id = location_message->sheep_id;

         /* Check if I've already found a sheep */
         if ( count > 0 )
         {
         /* Have already found a sheep so choose whether I will order the previous sheep or this one */
         which_sheep = rand()/(RAND_MAX+1.0);
         if ( which_sheep < 0.5 ) sheep_ordered = mboard_sheep_id;
         } else {
         /* I've not already found a sheep so prepare to order the current sheep */
         sheep_ordered = mboard_sheep_id;
         }
            count++;
        }
    FINISH_LOCATION_MESSAGE_LOOP

   /* If found a sheep, add message to "order" board */
   if ( count > 0 )
   {
   add_order_message(WOLF_ID, sheep_ordered);
   }

    return 0;
}

int eat_dinner()
{
int id1;

/* Check whether I've got a sheep to eat */
START_SHEEP_FOOD_MESSAGE_LOOP
id1 = sheep_food_message->wolf_id;

if (id1 == WOLF_ID)
{
WOLF_ENERGY+= (double) GAIN_FROM_FOOD_WOLF;
}

FINISH_SHEEP_FOOD_MESSAGE_LOOP

return 0;

}

/* When a wolf runs out of energy, it dies. */
 /* Should only be called for wolves with less than 1 unit of energy via condition in XMML. */
int die_wolf()
{

    return 1;

}

/* Wolves reproduce with a fixed probability. */
int reproduce_wolf()
{

    if ( rand()/(RAND_MAX+1.0) < REPRODUCE_WOLF_PROB )
```

```
    {
     GLOBAL_wolf_ids++;
     /* Make new wolf ids simple for now */
     add_wolf_agent(GLOBAL_wolf_ids, WOLF_X, WOLF_Y,
                      rand()/(RAND_MAX+1.0)*360.0, WOLF_ENERGY/2);
     WOLF_ENERGY /= 2;
    }
    return 0;
}
```

B.2 Sheep functions

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

int GLOBAL_sheep_ids = 0;

/* Move about at random */
int move_sheep()
{
double PI=3.14159267;
    double x, y;
    double heading;

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

    /* Read agent memory */
    x = SHEEP_X;
    y = SHEEP_Y;
    heading = SHEEP_HEADING;

    /* Change heading up to 50 degrees in either direction */
    heading += -50.0 + 100.0 * 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*/
    SHEEP_X = x;
    SHEEP_Y = y;
    SHEEP_HEADING = heading;

    /* Use up 1 unit of energy from the move */
    SHEEP_ENERGY--;

    return 0;
}

/* Post the position of the sheep. */
int post_position()
```

```
{
    /* Add message to "location" board (id, x, y) */
    add_location_message(SHEEP_ID, SHEEP_X, SHEEP_Y);

    return 0;
}

int eat_grass()
{
int mboard_sheep_id;

    START_GRASS_FOOD_MESSAGE_LOOP
    mboard_sheep_id = grass_food_message->sheep_id;

if ( mboard_sheep_id == SHEEP_ID )
{
SHEEP_ENERGY+= (double) GAIN_FROM_FOOD_SHEEP;
    }
    FINISH_GRASS_FOOD_MESSAGE_LOOP

return 0;
}

/* Sheep check to see if they'll be eaten */
int am_dinner()
{
    /* Local variables */
    int mboard_sheep_id, mboard_wolf_id, wolf_eats;
    double which_wolf;
    int count;

    count = 0;
    START_ORDER_MESSAGE_LOOP
    mboard_sheep_id = order_message->sheep_id;
    mboard_wolf_id = order_message->wolf_id;

/* Check if I'm on the order board */
if ( mboard_sheep_id == SHEEP_ID )
{
     /* Check if I've already been ordered by a previous wolf */
     if ( count > 0 )
     {
     /* I've already been ordered so choose whether the previous wolf or this wolf gets to eat me */
      which_wolf = rand()/(RAND_MAX+1.0);
      if ( which_wolf < 0.5 ) wolf_eats = mboard_wolf_id;
     } else {
     /* I've not already been ordered so current wolf is selected */
         wolf_eats = mboard_wolf_id;
     }
     count++;
}

    FINISH_ORDER_MESSAGE_LOOP

    /* If I was ordered then deliver the wolf his dinner and delete myself */
    if ( count > 0 )
    {
     add_sheep_food_message(wolf_eats);
     return 1;
    }

    return 0;
}
```

```
/* When a sheep runs out of energy, it dies. */
 /* Should only be called for sheep with less than 1 unit of energy via condition in XMML. */
int die_sheep()
{

    return 1;

}

/* Sheep reproduce with a fixed probability. */
int reproduce_sheep()
{

    if (rand()/(RAND_MAX+1.0) < REPRODUCE_SHEEP_PROB )
    {
     GLOBAL_sheep_ids++;
     /* Make new sheep ids simple for now */
     add_sheep_agent(GLOBAL_sheep_ids,SHEEP_X,SHEEP_Y,
                     rand()/(RAND_MAX+1.0)*360.0, SHEEP_ENERGY/2);
     SHEEP_ENERGY /= 2;
    }

    return 0;
}
```

B.3 Grass functions

```
#include "header.h"
#include "grass_agent_header.h"

/* Read sheep locations to find out if I (grass) will be eaten
 * Should only be called for grass agents which are green (colour = 1) via XMML*/
int grass_eaten()
{
int mboard_sheep_id, sheep_eats, count;
double x1, y1, which_sheep;

    count = 0;
    START_LOCATION_MESSAGE_LOOP
        x1 = location_message->sheep_x;
        y1 = location_message->sheep_y;

        /* Check whether location of sheep came from within my 1x1 patch */
        /* Note use of round, not trunc to ensure grass on boundary can be eaten */
        if ( (GRASS_X == (int) round(x1)) && (GRASS_Y == (int) round(y1)) )
        {
         mboard_sheep_id = location_message->sheep_id;

         /* Check if I've already found a sheep */
         if ( count > 0 )
         {
         /* Have already found a sheep so choose whether the previous sheep or this one gets to eat me*/
         which_sheep = rand()/(RAND_MAX+1.0);
         if ( which_sheep < 0.5 ) sheep_eats = mboard_sheep_id;
         } else {
         /* I've not already found a sheep so prepare to tell current sheep that it can eat me*/
         sheep_eats = mboard_sheep_id;
         }
            count++;
        }
    FINISH_LOCATION_MESSAGE_LOOP

    /* If found a sheep, add message to "grass_food" board
```

```
       and update my memory to brown i.e. no grass */
    if ( count > 0 )
    {
     add_grass_food_message(sheep_eats);
     GRASS_COLOUR = 0;
    }

return 0;
}

/* Should only be called for grass agents which are brown (colour = 0)
  via condition in XMML */
int grow_grass()
{

if (GRASS_COUNTDOWN <= 0)
{
GRASS_COLOUR = 1;
GRASS_COUNTDOWN = GRASS_REGROWTH;
} else {
GRASS_COUNTDOWN--;
}
return 0;
}
```