FLAME

Flexible Large-scale Agent-based Modelling Environment

User Manual

Simon Coakley
Mariam Kiran

University of Sheffield
Unit - USFD

October 10, 2009

Abstract

FLAME (Flexible Large-scale Agent-based Modelling Environment) is a tool which allows modellers from all disciplines, economics, biology or social sciences to easily write their own agent-based models. The environment is a first of its kind which allows simulations of large concentrations of agents to be run on parallel computers without any hindrance to the modellers themselves.

This document present a comprehensive guide to the keywords and functions available in the FLAME environment for the modellers to write their own agent models to facilitate research. This user manual describes how to create model description and write implementation code for the agents.

Installation guides to the tools required with FLAME have been provided separately in the document titled '*Getting started with FLAME*'. this documentation also contains details on executing example FLAME code and running your own models.

Contents

1 Introduction

The FLAME framework is an enabling tool to create agent-based models that can be run on high performance computers (HPCs). The framework is based on the the logical communicating extended finite state machine theory (X-machine) which gives the agents more power to enable writing of complex models for large complex systems.

The agents are modelled as communicating X-machines allowing them to communicate through messages being sent to each other as per required by the modeller. This information is automatically read by the FLAME framework and generate a simulation program which enables these models to be parallelised efficiently over parallel computers.

The simulation program for FLAME is called the **Xparser**. The Xparser is a series of compilation files which can be compiled with the modeller's files to produce the simulation package for running the simulations. Various tools have to be installed with the Xparser to allow the simulation program o be produced. These have been explained in the accompanying document, '*Getting started with FLAME*'.

Various parallel platforms like SCARF, (add more) have been used in the development process to test the efficiency of the FLAME framework. This work was done in conjunction with STFC and more details of the results obtained can be found in '*Deliverable 1.4: Porting of agent models to parallel computers*'.

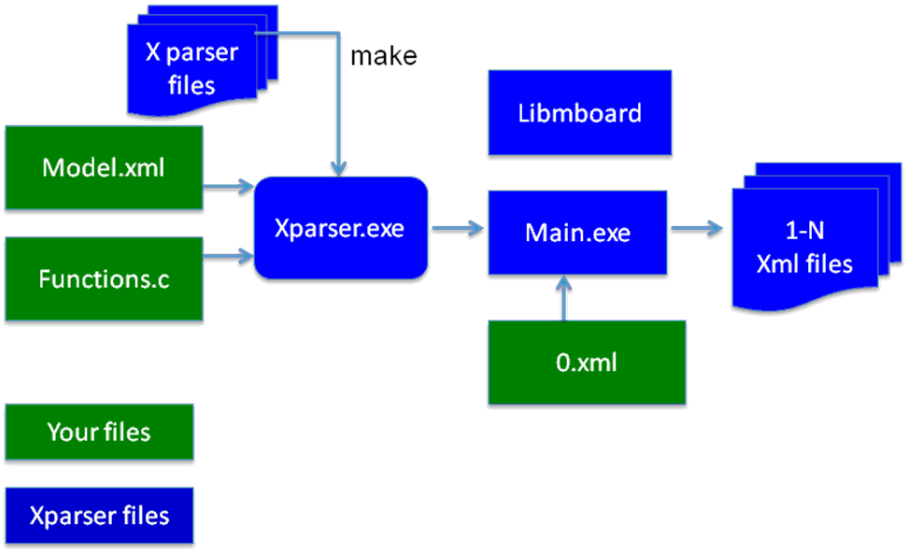

Figure 1: Block diagram of the Xparser, the FLAME simulation program. Blocks in blue are the files automatically generated. The green blocks are modeller files.

This document is a comprehensive guide to modellers of all disciplines to write their own agent models with ease. The key advantage of FLAME lies in the fact that modellers from

all disciplines, regardless of belonging to the computer science background can write their own models with ease.

This document has been divided in the following order, Section 2 describes the X-machine methodology to explain the structure of the agents. Modellers are required to write only two files for their model, model.xml and the functions of the agents. Section 3 describes how the model is written focussing on writing the 'model.xml' file which contains the complete description of the model, with the agents involved, their structures and the environment they exist in. Section 4 focuses on the second modeller file which is writing the agent functions and the routines provided by FLAME. Section 5 gives a brief description of how the model can be executed with a summary of the facilities provided by FLAME for data analysis for the output files generated. Appendices provides a listing of the tags and the data flow documentation used by FLAME.

2 Model Design

Traditionally specifying software behaviour has used finite state machines. Extended finite state machines (X-machines) are more powerful than the simple finite state machine and are used to represent the agents. Using these machines, the following characteristics of the agents are identified:

- A finite set of internal states
- Set of transitions functions that operate between the states.
- An internal memory set of the agent.
- A language for sending and receiving messages among agents.

Conventional state machines describe the state-dependent behaviour of a system in terms of its inputs, but this fails to include the effect of data. X-Machines are an extension to conventional state machines that include the manipulation of memory as part of the system behaviour, and thus are a suitable way to specify agents. Describing a system would thus include the following individual stages for creating a model:

- Identifying the agents and their functions.
- Identify the states which impose some order of function execution.
- Identify the input messages and output messages of each function (including possible filters on inputs).
- Identify the memory as the set of variables that are accessed by functions (including possible conditions on variables for the functions to occur).

Agent functions can accept an input stream of messages This is handled by the implementation of each function written in the source code.

2.1 Swarm Example

A swarm model is presented here as an example of how the agents behave. Swarm model describes the swarming behaviour in birds where they produce various patterns during flight. This simple flocking model would include agents which have to sense where other agents are and then respond accordingly. The functions they would possess would be:

- Signal. The agent would send information of its current position.

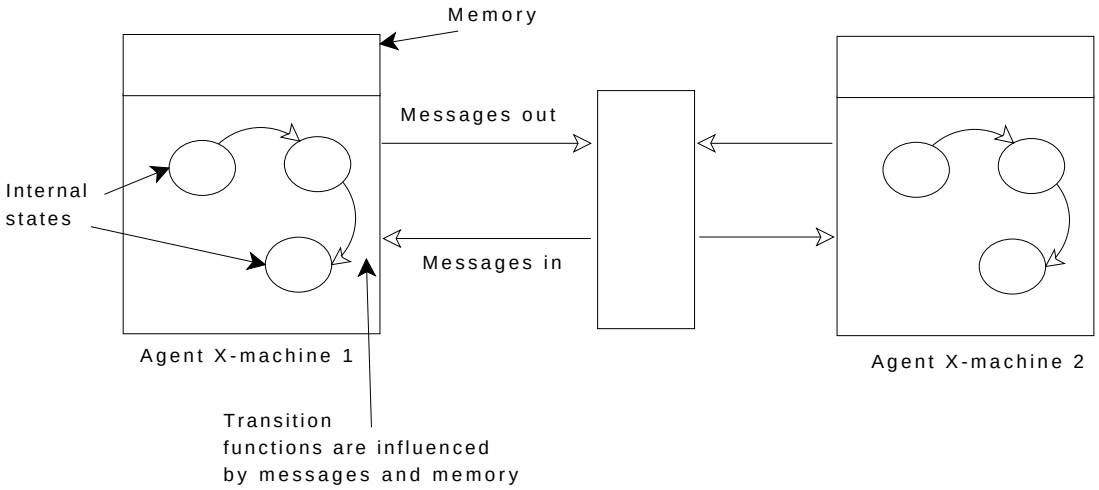

Figure 2: How two agent x-machines communicate. The agents send and read messages from the message board which maintains a database of all the messages sent by the agents.

- Observe. The agent would read in the positions from other agents and possibly change velocity.
- Respond. The agent would update position via the current velocity.

The functions would occur in an order as seen in Figure 3. The agents would traverse through the states during one iteration. FLAME prevents the agents to loop back due to parallelisation constraints.

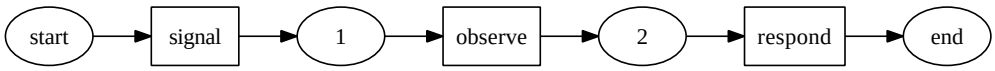

Figure 3: Swarm model including states

Functions can also have conditions in the model. For instance, in the swarm model, there can be a response function for flying and an alternate for resting on the ground. The condition on the flying response function would be that the z-axis position of the agent be greater than zero while the resting response function condition would be when the z-axis position was zero, see Figure 4.

2.2 Transition Function

The transition functions allow the agents to change the state in which they are in, modifying their behaviour accordingly. These would require as inputs their current state s_1, current

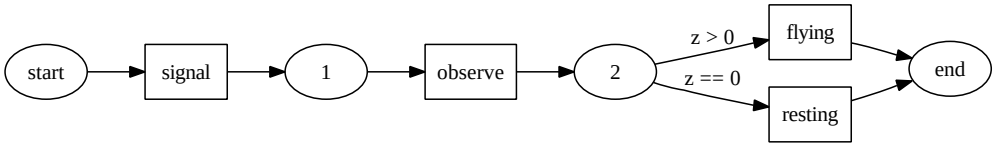

Figure 4: Swarm model including function conditions

memory value m_1, and the possible arrival of a message that the agent is able to read, t_1. Depending on these three values the agent can then change to another state s_2, updates the memory to m_2 and optionally sends a message, t_2. Figure **??** depicts how the transition function works within the agent.

The messages required for communication between agents are a signal message, which is output from 'signal' and input to 'observe', see Figure 5. This message would include the position of the agent that sent it, see Table 1. A feature of swarm models and most agent-based models is that there is generally a limit on incoming communication. In the swarm case this is the perceived distance of sight that an agent can view the location of other agents. This feature can be added to the model as a filter on inputs to a function, where the filter is a formula involving the position contained in the message (the position of the sending agent) and the receiving agent position.

Some of the transition functions may not depend on the incoming message. Thus the message would then be represented as:

$$Message = \{\emptyset, < data >\} \tag{1}$$

These agent transition functions may be expressed in terms of stochastic rules, thus allowing the multi-agent systems to be termed as stochastic systems.

2.3 Memory and States

The difference between the internal set of states and the internal memory set allows for added flexibility when modelling systems. There can be agents with one internal state and all the complexity defined in the memory or equivalently, there could be agents with a trivial memory with the complexity then bound up in a large state space. There are good examples of choosing an appropriate balance between these two as this enables the complexity of the models to be better managed.

Functions that take a message type as input are only executed once all functions that output the same message type have finished. One iteration is taken as a standalone run of a simulation, so once all the functions that have a message type as an input have been executed, the messages are deleted as they are no longer required. Messages cannot be sent between iterations.

Type	Name	Description
double	px	x-axis position
double	py	y-axis position
double	pz	z-axis position

Table 1: Signal Message

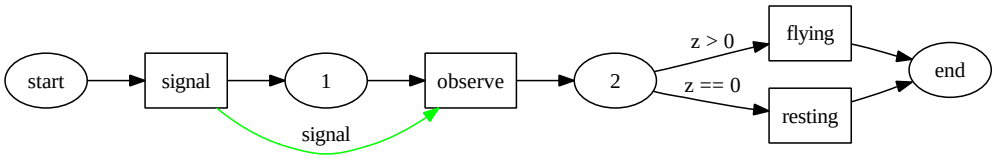

Figure 5: Swarm model including messages

Finally the memory required by the agent functions include the position of the agent, and its velocity, as shown in Table 2.

Type	Name	Description
double	px	position in x-axis
double	py	position in y-axis
double	pz	position in z-axis
double	vx	velocity in x-axis
double	vy	velocity in y-axis
double	vz	velocity in z-axis

Table 2: Swarm Agent Memory

The swarm model can also be represented as a transition table, see Table 3, where:

- Current State – is the state the agent is currently in.
- Input – is any inputs into the transition function.
- M_{pre} – are any preconditions of the memory on the transition.
- Function – is the function name.
- M_{post} – is any change in the agent memory.
- Output – is any outputs from the transition.
- Next State – is the next state that is entered by the agent.

Current State	Input	M_{pre}	Function	M_{post}	Output	Next State
start			signal		signal	1
1	signal		observe	(velocity updated)		2
2		$x > 0$	flying	(position updated)		end
2		$x == 0$	resting	(position updated)		end

Table 3: Swarm Agent Transition Table

Section 3 on model description describes how to write a model description into an XML file that FLAME can understand. Section 4 on model implementation describes how to implement the individual agent functions, i.e. M_{post} from the transition table. Section 5 on model execution describes how to use the tools in FLAME to generate a simulation program, compile it, and run it.

3 Model Description

Models descriptions are formatted in XML (Extensible Markup Language) tag structures to allow easy human and computer readability, and allow easier collaborations between developers writing applications that interact with model definitions.

The DTD (Document Type Definition) of the XML document is currently located at:

```
http://eurace.cs.bilgi.edu.tr/XMML.dtd
```

The start and end of a model file should be formatted as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE xmodel SYSTEM "http://eurace.cs.bilgi.edu.tr/XMML.dtd">
<xmodel version="2">
<name>Model_name</name>
<version>the version</version>
<description>a description</description>
...
</xmodel>
```

Where name is the name of the model, version is the version, and description allows the description of the model. Models can also contain:

- **Other models** (enabled or disabled)
- **Environment**
  - constant variables
  - location of function files
  - time units
  - data types
- **Agent types**
  - name
  - description
  - memory
  - functions
- **Message types**
  - name
  - description
  - variables

3.1 Model in Multiple Files

It is possible to define a model in multiple files. FLAME reads a model from multiple files as if the model was defined in one file. This capability allows different parts of a model to be enabled or disabled easily. For example if a model includes different versions of a sub-model that can be exchanged, or a subsystem of a model can be disabled to see how it affects the model. Alternatively this capability could be used as a hierarchy, for example a 'body' model could include a model of the 'cardiovascular system' that includes a model of the 'heart'. The following tags show the inclusion of two models, one enabled and one disabled:

```
<models>
  <model><file>sub_model_1.xml</file><enabled>true</enabled></model>
  <model><file>sub_model_2.xml</file><enabled>false</enabled></model>
</models>
```

3.2 Environment

The environment of a model holds information that maybe required by a model but is not part of an agent or a message. This includes:

- constant variables – for setting up different simulations easily
- location of function files – the path to the implementations of agent functions
- time units – for easily activating agent functions dependent on time periods
- data types – user defined data types used by agent memory or message variables

This notion of environment does not correspond to an environment that would be a part of a model where agents would interact with the environment. Anything that can change in a model must be represented by an agent, therefore if a model includes a changeable environment that agents can interact with, this in itself must be represented by an agent.

3.2.1 Constant Variables

These are constant variables that can be set as part of a simulation runs initial starting values, and can be defined as follows:

```
<constants>
  <variable>
   <type>int</type><name>my_constant</name>
```

```
    <description>value read in initial simulation settings</description>
   </variable>
 </constants>
```

3.2.2 Function Files

Function files hold the source code for the implementation of the agent functions. They are included in the compilation script (Makefile) of the produced model:

```
<functionFiles>
<file>function_source_code_1.c</file>
<file>function_source_code_2.c</file>
</functionFiles>
```

3.2.3 Time Units

Time units are used to define time periods that agent functions act within. For example a model that uses a calendar based time system could take a day to be the smallest time step, i.e. one iteration. Other time units can then use this definition to define other time units, for example weeks, months, and years.

A time unit contains:

- name – name of the time unit.
- unit – can contain 'iteration' or other defined time units.
- period – the length of the time unit using the above units.

An example of a calendar based time unit set up is given below:

```
<timeUnits>
  <timeUnit>
    <name>daily</name>
    <unit>iteration</unit>
    <period>1</period>
  </timeUnit>

  <timeUnit>
    <name>weekly</name>
    <unit>daily</unit>
    <period>5</period>
  </timeUnit>
```

```
  <timeUnit>
    <name>monthly</name>
    <unit>weekly</unit>
    <period>4</period>
  </timeUnit>

  <timeUnit>
    <name>quarterly</name>
    <unit>monthly</unit>
    <period>3</period>
  </timeUnit>

  <timeUnit>
    <name>yearly</name>
    <unit>monthly</unit>
    <period>12</period>
  </timeUnit>

</timeUnits>
```

3.2.4 Data Types

Data types are user defined data types that can be used in a model. They are a structure for holding variables. Variables can be:

- single C fundamental data types – int, float, double, char.
- static array – of size ten: variable_name[10].
- dynamic array – available by placing '_array' after the data type name: variable_name_array.
- user defined data type – defined before the current data type.

The example below the data type *line* contains a variable of data type *position* which is defined above it:

```
<dataTypes>

 <dataType>
  <name>position/name>
  <description>position in 3D using doubles</description>
  <variables>
```

```
    <variable><type>double</type><name>x</name>
     <description>position on x-axis</description>
    </variable>
    <variable><type>double</type><name>y</name>
     <description>position on y-axis</description>
    </variable>
    <variable><type>double</type><name>z</name>
     <description>position on z-axis</description>
    </variable>
   </variables>
  </dataType>

  <dataType>
   <name>line</name>
   <description>a line defined by two points</description>
   <variables>
    <variable><type>position</type><name>start</name>
     <description>start position of the line</description>
    </variable>
    <variable><type>position</type><name>end</name>
     <description>end position of the line</description>
    </variable>
   </variables>
  </dataType>

 </dataTypes>
```

3.3 Agents

An agent type contains a name, a description, memory, and functions:

```
 <agents>

   <xagent>
     <name>Agent_Name</name>
     <description></description>
     <memory>
      ...
     </memory>
     <functions>
       ...
     </functions>
   </xagent>
```

3.3.1 Agent Memory

Agent memory defines variables, where variables are defined by their type, C data types or user defined data types from the environment, a name, and a description:

```
<memory>
 <variable><type>int</type><name>id</name>
  <description>identity number</description>
 </variable>
 <variable><type>double</type><name>x</name>
  <description>position in x-axis</description>
 </variable>
</memory>
```

3.3.2 Agent Functions

An agent function contains:

- name - the function name which must correspond to an implemented function name
- description
- current state - the current state the agent has to be in.
- next state - the next state the agent will transition to.
- condition - a possible condition of the function transition.
- inputs - the possible input messages.
- outputs - the possible output messages.

And as tags:

```
<function>
 <name>function_name</name>
 <description>function description</description>
 <currentState>current_state</currentState>
 <nextState>next_state</nextState>
 <condition>
 ...
 </condition>
 <inputs>
 ...
```

```
  </inputs>
  <outputs>
  ...
  </outputs>
</function>
```

The current state and next state tags hold the names of states. This is the only place where states are defined. State names must coordinate with other functions states to produce a transitional graph from the start state to end states.

A function can have a condition on its transition. This condition can include conditions on the agent memory and also on any time units defined in the environment. At any state with outgoing transitions with conditions it must be possible for a transition to happen, i.e. it must be possible for every agent to transition from the start state to an end state. Each possible transition must be mutually exclusive, i.e. the order that the function conditions are tested is not defined. A function named 'idle' is available to be used for functions that do not require an implementation.

Conditions (that are not just time unit based) take the form:

- lhs – left hand side of comparison
- op – the comparison operator
- rhs – the right hand side of the comparison

Or in tags:

```
<lhs></lhs><op></op><rhs></rhs>
```

Sides to compare (lhs or rhs) can be either a value, denoted by value tags, a formula, currently also in value tags, or another comparison rule. Values and formula can include agent variables which are preceded by 'a'.

The comparison operator can be one of the following comparison functions:

- EQ – equal to
- NEQ – not equal to
- LEQ – less than or equal to
- GEQ – greater than or equal to
- LT – less then

- GT – greater than
- IN – an integer (in lhs) is a member of an array of integers (in rhs)

or can be one of the following logic operators as well:

- AND
- OR

The operator 'NOT' can be used by placing 'not' tags around a comparison rule. For example the following tagged rule describes the condition being true when the 'z' variable of the agent is greater than zero and less than ten:

```
<condition>
 <lhs>
  <lhs><value>a.z</value></lhs>
  <op>GT</op>
  <rhs><value>0.0</value></rhs>
 </lhs>
 <op>AND</op>
 <rhs>
  <not>
  <lhs><value>a.z</value></lhs>
  <op>LT</op>
  <rhs><value>10.0</value></rhs>
  </not>
 </rhs>
</condition>
```

A condition can also depend on any time units described in the environment. For example the following condition is true when the agent variable 'day_of_month_to_act' is equal to the number of iterations since of the start, the phase, of the 'monthly' period, i.e. twenty iterations as defined in the time unit:

```
<condition>
    <time>
    <period>monthly</period>
    <phase>a.day_of_month_to_act</phase>
    </time>
</condition>
```

Functions can have input and output message types. For example the following example the function takes message types 'a' and 'b' as inputs and outputs message type 'c':

```
<inputs>
 <input><messageName>a</messageName></input>
 <input><messageName>b</messageName></input>
</inputs>
<outputs>
 <output><messageName>c</messageName></output>
</outputs>
```

Message filters can be applied to message inputs and allow the messages to be filtered. Filters are defined similar to function conditions but include message variables which are prefixed by an 'm'. The following filter only allows messages where the agent variable 'id' is equal to the message variable 'worker_id':

```
<input>
 <messageName>firing</messageName>
 <filter>
  <lhs><value>a.id</value></lhs>
  <op>EQ</op>
  <rhs><value>m.worker_id</value></rhs>
 </filter>
 <random>false<random>
</input>
```

The previous example also includes the use of a random tag, set to false, to show that the input does not need to be randomised, as randomising input messages can be computationally expensive. By default all message inputs are randomised.

Using filters in the model description enables FLAME to make message communication more efficient by pre-sorting messages and using other techniques.

3.4 Messages

Messages defined in a model must have a type which is defined by a name and the variables that are included in the message. The following example is a message called 'signal' that holds a position in 3D.

```
<messages>

 <message>
  <name>signal</name>
  <description>Holds the position of the sending agent</description>
  <variables>
    <variable><type>double</type><name>x</name>
```

```
     <description>The x-axis position</description>
    </variable>
    <variable><type>double</type><name>y</name>
     <description>The y-axis position</description>
    </variable>
    <variable><type>double</type><name>z</name>
     <description>The z-axis position</description>
    </variable>
  </variables>
 </message>

</messages>
```

4 Model Implementation

The implementations of each agent's functions are currently written in separate files written in C, suffixed with '.c'. Each file must include two header files, one for the overall framework and one for the particular agent that the functions are for. Functions for different agents cannot be contained in the same file. Thus, at the top of each file two headers are required:

```
#include "header.h"
#include "<agentname>_agent_header.h"
```

Where '<agent_name>' is replaced with the actual agent name. Agent functions can then be written in the following style:

```
/*
 * \fn: int function_name()
 * \brief: A brief description of the function.
 */
int function_name()
{
   /* Function code here */

   return 0; /* Returning zero means the agent is not removed */
}
```

The first commented part (four lines) is good practice and can be used to auto-generate source code documentation. The function name should coordinate with the agent function name and the function should return an integer. The functions have no parameters. Returning zero means the agent is not removed from the simulation, and one removes the agent immediately from the simulation.

4.1 Accessing Agent Memory Variables

After including the specific agent header, the variables in the agent memory can be accessed by capitalising the variable name:

```
AGENT_VARIABLE
```

To access elements of a static array use square brackets and the index number:

```
MY_STATIC_ARRAY[index]
```

To access the elements and the size of dynamic array variables use '.size' and '.array[index]':

```
MY_DYNAMIC_ARRAY.size
MY_DYNAMIC_ARRAY.array[index]
```

To access variables of a model data type use '.variablename':

```
MY_DATA_TYPE.variablename
```

4.1.1 Using Model Data Types

The following is an example of how to use a data type called *vacancy*:

```
/* To allocate a local data type */
vacancy vac;

/* And initialise */
init_vacancy(&vac);

/* Initialise a static array of the data type */
init_vacancy_static_array(&vac_static_array, array_size);

/* Free a data type */
free_vacancy(&vac);

/* Free a static array of a data type */
free_vacancy_static_array(&vac_static_array, array_size);

/* Copy a data type */
copy_vacancy(&vac_from, &vac_to);

/* Copy a static array of a data type */
copy_vacancy_static_array(&vac_static_array_from,
                          &vac_static_array_to, array_size);
```

If the data type is a variable from the agent memory, then the data type variable name must be capitalised.

4.1.2 Using Dynamic Arrays

Dynamic array variables are created by adding '_array' to the variable type. The following is an example of how to use a dynamic array:

```
/* Allocate local dynamic array */
vacancy_array vacancy_list;

/* And initialise */
init_vacancy_array(&vacancy_list);

/* Reset a dynamic array */
reset_vacancy_array(&vacancy_list);

/* Free a dynamic array */
free_vacancy_array(&vacancy_list);

/* Add an element to the dynamic array */
add_vacancy(&vacancy_list, var1, .. varN);

/* Remove an element at index index */
remove_vacancy(&vacancy_list, index);

/* Copy the array */
copy_vacancy_array(&from_list, &to_list);
```

If the dynamic array is a variable from the agent memory, then the dynamic array variable name must be capitalised.

4.2 Sending and receiving messages

Messages can be read using macros to loop through the incoming message list as per the template below, where 'messagename' is replaced by the actual message name. Message variables can be accessed using an arrow '->':

```
START_MESSAGENAME_MESSAGE_LOOP
 messagename_message->variablename
FINISH_MESSAGENAME_MESSAGE_LOOP
```

Messages are sent or added to the message list by

```
 add_messagename_message(var1, .. varN);
```

5 Model Execution

FLAME contains a parser program called 'xparser' that parses a model XML definition into simulation program source code that can be compiled together with model implementation source code. The xparser includes template files which are used to generate the simulation program source code.

The xparser takes as parameters the location of the model file and an option for serial or parallel (MPI) version, serial being the default if the option is not specified.

5.1 Generated Files

The xparser then generates simulation source code files in the same directory as the model file. These files include:

- Doxyfile – a configuration file for generating documentation using the program 'doxygen'
- header.h – a C header file for global variables and function declarations between source code files
- low_primes.h – holds data used for partitioning agents
- main.c – the source code file containing the main program loop
- Makefile – the compilation script used by the program 'make'
- memory.c – the source code file that handles the memory requirements of the simulation
- xml.c – the source code file that handles inputs and outputs of the simulation
- <agent_name>_agent_header.h – the header file containing macros for accessing agent memory variables
- rules.c – the source code file containing the generated rules for function conditions and message input filters
- messageboards.c – deprecated?
- partitioning.c – still used?

and in parallel the additional files:

- propagate_messages.c – deprecated?

- propagate_agents.c – still used?

The simulation source code files then require compilation, which can be easily achieved using the included compilation script 'Makefile' using the 'make' build automation tool. The program 'make' invokes the 'gcc' C compiler, which are both free and available on various operating systems. If the parallel version of the simulation was specified the compiler invoked by 'make' is 'mpicc' which is a script usually available on parallel systems.

The compiled program is called 'main'. The parameters required to run a simulation include the number of iterations to run for and the initial start states (memory) of the agents, currently a formatted XML file.

5.2 Start States Files

The format of the initial start states XML is given by the following example:

```
<states>
<itno>0</itno>

<environment>
<my_constant>6</my_constant>
</environment>

<xagent>
<name>agent_name</name>
<var_name>0</var_name>
...
</xagent>

...

</states>
```

The root tag is called 'states' and the 'itno' tag holds the iteration number that these states refer to. If there are any environment constants these are placed within the 'environment' tags. Any agents that exist are defined within 'xagent' tags and require the name of the agent within 'name' tags. Any agent memory variable (or environment constant) value is defined within tags with the name of the variable. Arrays and data types are defined within curly brackets with commas between each element.

When a simulation is running after every iteration, a states file is produced in the same directory and in the same format as the start states file with the values of each agent's memory.

A XML DTD

```
<!ELEMENT xmodel
 (name,version,description,models?,environment?,agents,contexts?,messages?)>
<!ATTLIST xmodel version CDATA #REQUIRED>

<!ELEMENT models (model*)>
<!ELEMENT model (file,enabled)>

<!ELEMENT environment (constants?,functionFiles?,timeUnits?,dataTypes?)>
<!ELEMENT dataTypes (dataType*)>
<!ELEMENT dataType (name,description,variables)>
<!ELEMENT variables (variable*)>
<!ELEMENT variable (type,name,description)>
<!ELEMENT constants (variable*)>
<!ELEMENT functionFiles (file*)>
<!ELEMENT timeUnits (timeUnit*)>
<!ELEMENT timeUnit (name,unit,period)>

<!ELEMENT agents (xagent*)>
<!ELEMENT xagent (name,description,memory?,roles?,functions?)>
<!ELEMENT memory (variable*)>
<!ELEMENT roles (role*)>
<!ELEMENT role (name,description,functions)>
<!ELEMENT functions (function*)>
<!ELEMENT function
 (name,description,code?,currentState,nextState,condition?,inputs?,outputs?)>
<!ELEMENT condition ((not)|(lhs,op,rhs)|(time))>
<!ELEMENT not ((lhs,op,rhs)|(time))>
<!ELEMENT lhs ((not)|(lhs,op,rhs)|(value)|(time))>
<!ELEMENT rhs ((not)|(lhs,op,rhs)|(value)|(time))>
<!ELEMENT time (period,phase,duration?)>
<!ELEMENT inputs (input*)>
<!ELEMENT input (messageName,filter?,sort?)>
<!ELEMENT filter (lhs,op,rhs)>
<!ELEMENT outputs (output*)>
<!ELEMENT output (messageName)>
<!ELEMENT contexts (xcontext*)>
<!ELEMENT xcontext (name,description,messages)>
<!ELEMENT messages (message*)>
<!ELEMENT message (name,description,variables)>

<!ELEMENT code (#PCDATA)>
<!ELEMENT currentState (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

```
<!ELEMENT enabled (#PCDATA)>
<!ELEMENT file (#PCDATA)>
<!ELEMENT messageName (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT nextState (#PCDATA)>
<!ELEMENT op (#PCDATA)>
<!ELEMENT sort (#PCDATA)>
<!ELEMENT statement (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT unit (#PCDATA)>
<!ELEMENT period (#PCDATA)>
<!ELEMENT phase (#PCDATA)>
<!ELEMENT duration (#PCDATA)>
```