FLAME
Developer Manual

Simon Coakley
Mariam Kiran

University of Sheffield
Unit - USFD

October 12, 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.

The report presents a developer manual for the FLAME framework. The manual describes the management of the agent memory, execution and agent communication through the libmboard. The document also contain a details description of the implementation of the xparser and how it traverses through the code. The document also summarises the various versions of the xparser and the changes updated to them.

Contents

1 Introduction

The FLAME framework is a tool which enables creation of agent-based models that can be run on high performance computers (HPCs). The framework is based on 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 designed by the modeller. This information is automatically read by the FLAME framework and generates 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 a simulation package for running the simulations. Various tools have to be installed with the Xparser to allow the simulation program to 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 the STFC unit (Science and Technology Facilities Council) and more details of the results obtained can be found in '*Deliverable 1.4: Porting of agent models to parallel computers*'.

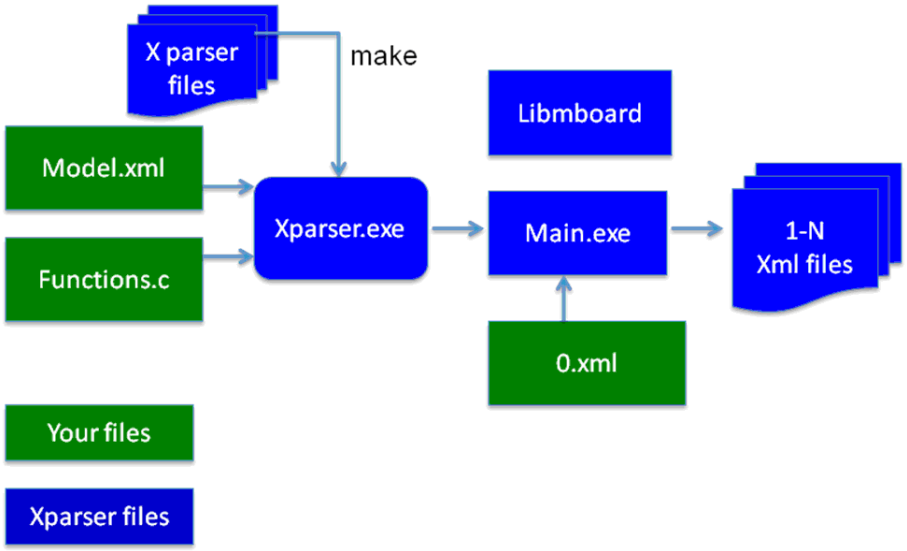

Figure 1: Block diagram of the Xparser, the FLAME simulation program. Blocks in blue are the files automatically generated. The green blocks are modeller files.

2 XParser Distribution

The xparser is distributed as a series of template files accompanied with a few header files. These template files can be downloaded in to the desired directory. The freely available GCC compiler is then used to compile the files on the machine.

The libmboard (or the Message Board) is an additional feature of the xparser which is being developed to increase the efficiency of parallel communication of large computers. This file can also be downloaded and compiled on the machine for running the simulations. Details of installation and tools have been provided in an accompanying guide '*Getting started with Flame*'.

3 XParser generated files

Reading the accompanying document '*FLAME User Manual*', it is explained that when executing the xparser with the model, a number of files are generated as part of the simulation package. These files are as follows,

- Doxyfile - Generated documentation for the model.
- Header files for each agent memory - Contains pointers for accessing agent memory during simulation.
- Header.h - Memory for the xparser.
- Low_primes.h - For partitioning of the agents.
- Main.c - The main C code for running the simulation.
- Main.exe - The simulation file.
- MakeFile - Makefile contains the details of the locations of the files, flags associated etc.
- Memory.c - contains the memory functions like reading through messages or agents.
- Messageboard.c - deprecated?
- Partitioning.c -deprecated?
- Rules.c - deprecated?
- Xml.c - Contains functions to parse through the xml file.

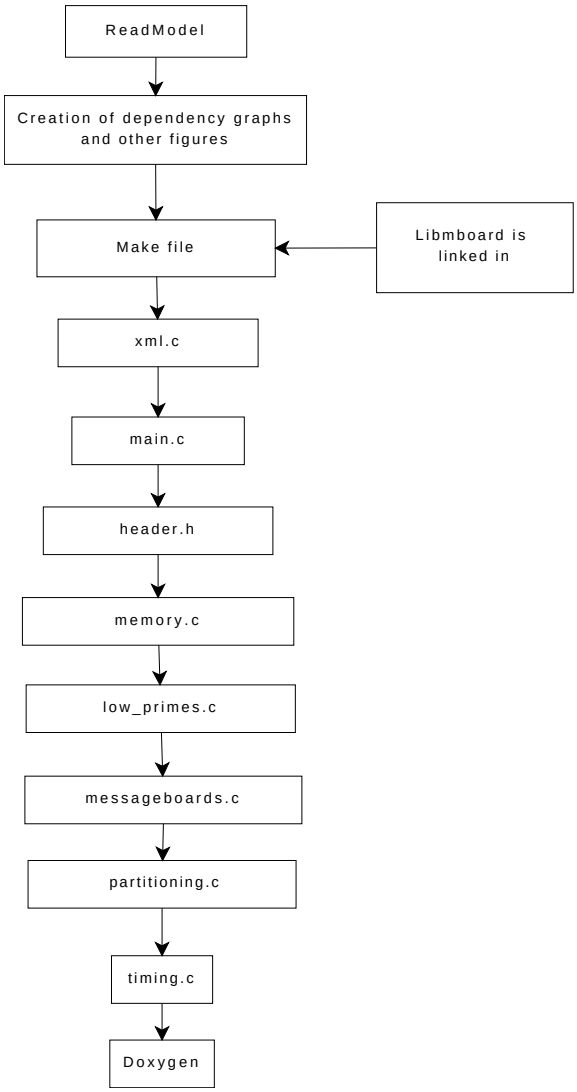

Figure 2: Block diagram of the series of files read for creating the model simulation package.

Figure 2 describes a series of steps which the xparser goes through to generate a simulation package of the model. These steps have been explained as follows:

1. Reading in the model. The template Readmodel.tmpl provides these functions. This file allows reading of the various tags in the model xml file.

2. Creating of dependency graph. The dot files are generated which are known as a series of stategraph files. These diagrams display the description of the model, which order the functions are called in and the different layers which denote the synchronisation points among the agent functions due to communication dependencies.

3. Writing out the make file. This file contains the location of various files, like the libmboard and more.

4. Writing out the xml.c file. This file contains functions for reading specific data variables like static arrays, ints or doubles with in the agent memory. It also contain functions for reading the data structures within the agent memory, for example:

   ```
   read_mall_strategy(char * buffer, int * counter, mall_strategy * tempdatatype)
   ```

   The file also contains functions on reading the initial starting states file for the agent memories. For this purpose it opens the '0.xml' file and reads these values.

   For parallel computation, an array is initialised for allowing round robin distribution of the agents.

5. Writing out the main.c file. This is the main file which contains the complete xparser functions being called. It reads in the number of iterations needed for the simulation, the initial start states file, generate partitions for parallel computing and saves the iteration data in progressive xml files. This file also performs additional functions, like

   - traverses through the different agent states.
   - checks conditions of the agent functions before calling them.
   - calls the synchronisation code for the message boards. These are specific MB functions which have been documented in the libmboard documentation.
   - creates iterator for the messages.
   - freeing agents when moving to next states.
   - clears the message board at the end of the iteration.
   - clean up.

6. Header.h file. This file provides the names being used in the simulation. For instance, the xmachine agent memory, the states and the prototypes for the agent functions. Various definitions for the messageboards like names of the iterators are also defined here, along with prototypes for reading and writing various agent memory variables.

7. Writing the memory.c file. Memory.c file contains the actual function code of the functions being used by the xparser. These are functions like free agents, or freeing messages by calling MB_Clear. The file contains various functions to initialize memory variables like arrays or data structures. Functions for adding and freeing agents are also contained here.

8. Writing the rules.c file. Rules .c contains the rules being used in the model. For instance,

- For function conditions,

```
iteration loop%20==6 return 1 else return 0;
```

- Or for agent memory conditions,

```
a->learningwindow==0 return 1 else return 0;
```

9. Writing the low_primes.c file. Low primes file defines the arrays which are used for partitioning of the data.

10. Writing the messageboards.c files. Messageboard.c is used for writing functions which allows access to the messageboard. For instance,

```
/*for adding messages*/
MB_AddMessage(b_messagename,&msg)

/*Rewinding an iterator*/
MB_Iterator_Rewind(i_mall_strategy_to_use)

/* getting a message*/
MB_Iterator_GetMessage(i_mall_strategy_to_use, (void**)&msg);
```

11. Writing the partitioning.c file. Partitioning.c file contains details for generating partitions as geometric nodes and saves this data.

12. The timing.c returns the time it takes to run the code.

13. The Doxygen file writes out data about the model file.

Details of the message board functions are available at `www.softeng.cse.clrc.ac.uk/wiki/EURACE/MessageBoards/ImplementationNotes/API`

4 XParser Versions

During the development process, the Xparser has gone through a series of development versions, each being modified to include more features for making use easy for the modellers and increasing efficiency.

Change logs for the different versions has been stored at the CcpForge repository for the developers. The XParser has a number of versions, with the latest version 0.15.13 which containing the following additions:

- Checks added for environment variables and data type names.
- Fixes of writing out of output settings to command line.
- Merging of messages filters.
- stategraph colour version added.
- fix bug where import file not taken relative from 0.xml location.
- merge of sync filters.
- bug fix for nested filter rules.
- added 'IN' operator for filter rule.
- added random tag to message inputs.
- added sort tag to message inputs.
- parallel application can read pre-partitioned input files.
- added constant tag for agent memory variables.
- add option -f to xparser for final production run.

The current final version xparser 0.15.13 has been tested and used for various simulations of the economic EURACE model and has been proved to be very stable and good for economic modelling.

This version has thus been tagged to be released as FLAME version 1.0. Any future updates to this version will be announced as new versions of FLAME.

5 Memory Allocation and its Problems

Memory allocation for the agents and the messages is done as a continuous block size of memory. The command *sizeof* is used to return a byte size of the agent memory in use. This is an important facet for parallelisation when using MPI. Sending data from one node to the other requires the program to know how many bytes have to be sent across to package it up in small packets. Thus it becomes important to determine its size.

5.1 Dynamic Arrays

FLAME also allows the use of dynamic arrays which causes a hindrance to this area of parallelisation. It is strongly discouraged for dynamic arrays to be used as part of the agent memory, if the agents have to be moved around in parallel. Dynamic arrays also prevents the associated memory to be allocated as blocks of continuous memory. Messages are another factor which discourages the use of dynamic arrays within the messages. The size of the message becomes difficult to be determined and sent to and fro for this reason.

5.2 Data Types

User-defined data types are allocated as pointers in agent memory but this has been modified in a new version to be released. This means that instead of user using an arrow '->' to dereference variables, a dot '.' is used to access the data structure.

Dynamic array data structures are also not allocated as a pointer (but the actual dynamic array is) which means functions to interact with a dynamic array data structure need to pass a pointer. This means the use of the ampersand '&' to reference the data structure.

5.3 Agent Memory Management

Each agent has an associated memory data structure. Since the early versions of the framework all agents have been managed in one list. This was so that the list could be randomised and therefore remove any chances of agents having priority over other agents by always being executed first. In essence, the same effect can be achieved by randomising the messages output and therefore the message inputs into agents. The current framework has a generic agent memory stucture that can point to any specific agent type.

With the introduction of the new message board library the action of randomising (or now also sorting and filtering) messages the need to randomise the agent list is redundant. Also redundant is the need to have a single list of all the agents. The generic agent memory structure is therefore not needed and each agent type can have it's only seperate list.

6 Execution

Agents have a number of functions to perform. The order that these functions are run is defined by the states associated with each function.

Figure 3 depicts an example of how the states can link functions together. In the first case, the agent performs two functions during the iteration step. The current state and the next state determine the order of the functions. *Function A* is followed by *Function B* by simply assigning the current and next states to link the function chain together. Case 2, presents another scenario, where the *Function A* is run twice during a simulation step. The same function can be run twice by linking if to different current and next states.

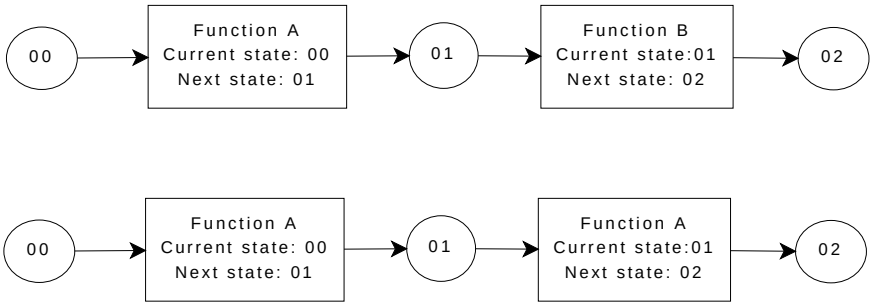

Figure 3: Using states to form a function chain during one iteration step.

These order of states also determines the internal dependency between the functions. This is only true if only one agent is being discussed. But if there is a dependency between more than one agent, communication dependencies are generated. These are denoted by the messages being sent and read by other functions.

The communication dependency sets up a synchronisation point as shown in Figure 4. This means that all agent As have to finish running their *Function A* before they can start running the *Function D* for agent B.

Using the X-machine methodology, the agents traverse through the states to run the defined functions. These functions are also the transition functions which are defined in the model XML file with the,

- current state: the current state of the agent
- input: the inputs the function is expecting
- m_{pre}: the conditions on memory of executing the function
- name: the name of the function
- m_{post}: the changes in the memory (i.e. the function code)

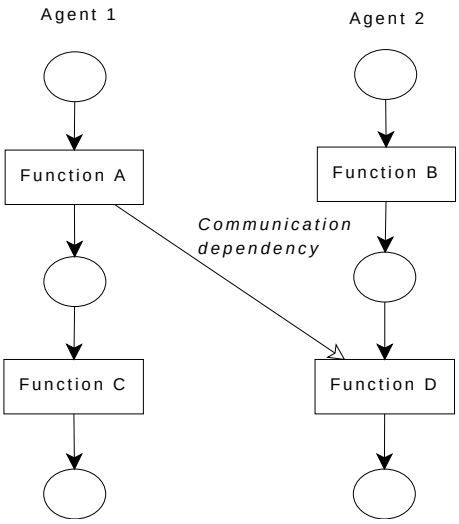

Figure 4: Function D of Agent 2 depends on all Agent 1s to finish their Function A.

- output: possible outputs of the function
- next state: the next state to move the agent to

By producing an order of function execution, this also provides a way to manage the processing of agents. By providing a link to an agent list for each possible agent state, agents can be moved between these agent state lists until they reach an end state.

7 Communication

MPI or Message Passing Interface is used to handle the communication between the agents. Using MPI has a number of advantages:

- MPI allows language independent communication, which means that different platforms can be linked together to communicate messages.
- Synchronisation between message channels.
- Shared or distributed memory.
- Packaging messages into blocks of memory to be sent across.

7.1 Libmboard

The communication handling was decoupled with the FLAME implementation and programmed as a separate message library. This was done to provide more flexibility with different parallelisation strategies to the current FLAME modellers.

The Message board library defines a set of routines and functions which can be integrated with the FLAME code to parse messages. This was called the libmboard and uses MPI to communicate between processors. Details of the Message Board can be found in its Reference Manual at `http://www.softeng.cse.clrc.ac.uk/libmboard`.