FLAME
User Manual

Simon Coakley
Mariam Kiran

Unit - USFD

August 22, 2008

Abstract

The report presents a manual for the FLAME framework. How to design a model, create a description of the model, and write the implementation of the model. Details are also included about how to execute a model.

Contents

0.1 Introduction

The FLAME framework is an enabling tool to create agent-based models that can be run on high performance computers (HPCs). Models are created based upon extended finite state machines that include message inputs and outputs. This information is used by the framework to automatically generate a simulation program that can run models efficiently on HPCs.

0.2 Model Design

Extended finite state machines or X-Machines are used to define agents within a model. The basic definition of an agent would thus, in accordance to the computational model, contain the following components:

1. A finite set of internal states.
2. A set of transition functions that operate between states.
3. An internal memory set. In practice, the memory would be a finite set and can be structured in any way required.
4. A language for sending and receiving messages between other agents.

$$X = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0) \tag{1}$$

where,

- Σ are the set of input alphabets
- Γ are the set of output alphabets
- Q denotes the set of states
- M denotes the variables in the memory.
- Φ denotes the set of partial functions ϕ that map and input and memory variable to an output and a change on the memory variable. The set ϕ: $\Sigma \times M \longrightarrow \Gamma \times M$
- F in the next state transition function, $F : Q \times \phi \longrightarrow Q$
- q_0 is the initial state and m_0 is the initial memory of the machine.

0.2.1 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 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.

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

$$Message = \{\emptyset, < data >\} \tag{2}$$

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

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.

Specifying software behaviour have traditionally involved finite state machines which allow modelling a system in terms of its inputs and outputs. More abstract system descriptions include UML which has already been proposed as a way to design agent-based models [?, ?, ?, ?] but these techniques lack precise descriptions needed for generating simulation code and for testing. Testing a system specified as a finite state machine makes it easier for the behaviour to be expressed as a graph and allow traversals of all possible and impossible executions of the system [1]. 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. The advantages of this approach have been highlighted in Section **??**. Describing a system would thus include the following individual stages for creating a model:

- Identifying the system functions
- Identify the states which impose some order of function execution
- Identify the input messages and output messages
- For each state identify the memory as the set of variables that are accessed by outgoing and incoming transition functions

[1] This is similar to branch traversal testing.

0.3 Model Description

Models descriptions are formated in XML to be human and computer readable.

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

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

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

```
<?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>
```

Models can contain:

- other models (enabled or disabled)
- **environment**
  - constant variables
  - function files
  - time units
  - data types
- **agents**
  - name
  - description
  - memory
  - functions
- **messages**
  - name
  - description
  - variables

0.3.1 Environment

The environment tag in the model.xml file hosts additional tags for information that may be required by the parser for efficient simulation of the model. Following are the tags that can be defined in it.

Constant Variables

Constant Variables refers to the global values used in the model. These can me defined in a separate header file which can then be included in one of the functions file.

The header file would look as follows:

```
#define <varname> <value>
```

If this file was saved as a 'my_header.h' file, include this file into one of the function files so that the compiler knows about these arguments.

Function Files

Function files are where you can place 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>
```

Time Rules

Time rules refer to the possibility of restricting the functions to only function at particular iterations. An iteration refers to the smallest unit the models are set up on. In EURACE, we are assuming every iteration to represent one day in the calender.

Following table represents how these time units can be defined in the model.xml file (example to appear)

Thus if there is any function which functions at only particular days of the calender, we can define them as a condition rule in the function definitions.

The example below depicts a function which executes every... (example to appear).

The condition for which iteration to execute can also be given as part of the agent memory. The following example depicts this:

(example to appear)

Time rules can be applied to function conditions instead of a condition rule.

Time rules are defined by a time period and a phase.

A time period needs to be defined as a time unit in the environment of a model.

For example: daily, monthly, yearly.

A time phase is the offset from the start of a period.

For example: an integer or an integer agent memory variable.

It can be defined thus:

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

These rules are then parsed into rule functions and placed in a file called rules.c.

Data Types

Data types are user defined data types that can be used in a model.

Data types can contain C data types or other predefined user data types.

```
<dataTypes>
 <dataType>
  <name>Histogram</name>
  <description>ADT Histogram</description>
  <variables>
   <variable><type>double</type><name>prob[30]</name><description></description>
   </variable>
   <variable><type>double</type><name>values[30]</name><description></description>
   </variable>
   <variable><type>double</type><name>max</name><description></description>
   </variable>
  </variables>
 </dataType>
 <dataType>
```

```
  <name>Belief</name>
  <description>ADT Belief</description>
  <variables>
   <variable><type>double</type><name>expectedPriceReturns</name><description></description>
   </variable>
   <variable><type>double</type><name>expectedTotalReturns</name><description></description>
   </variable>
   <variable><type>double</type><name>expectedCashFlowYield</name><description></description>
   </variable>
   <variable><type>double</type><name>volatility</name><description></description>
   </variable>
   <variable><type>Histogram</type><name>hist</name><description></description>
   </variable>
  </variables>
 </dataType>
</dataTypes>
```

0.3.2 Agent

Function Condition and Message Input Filter Rule Tags

Comparison Rules

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

lhs and rhs can be either a value, denoted by value tags:

```
<value></value>
```

or another rule.

Values can include agent and message memory variables, which are denoted by either:

```
a->agent_var
m->message_var
```

op can be either 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

or logic operators:

- AND
- OR

the operator NOT is used by placing 'not' tags around a rule:

```
<condition>
 <lhs>
  <lhs><value>a.employee_firm_id</value></lhs>
  <op>GT</op>
  <rhs><value>-1</value></rhs>
 </lhs>
 <op>AND</op>
 <rhs>
  <not>
  <lhs><value>a.on_the_job_search</value></lhs>
  <op>EQ</op>
  <rhs><value>0</value></rhs>
  </not>
 </rhs>
</condition>

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

0.3.3 Messages

0.4 Model Implementation

If agent functions are written in a separate functions file a separate file is required for each agent type.

At the top of each file two headers are required:

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

After this agent functions take the form:

```
int function_name()
{
   /* Function code here */

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

0.4.1 Accessing agent memory variables

After including the specific agent header, the variables for the agent can be accessed by capitalising the varaible name.

```
MY_SINGLE_VARIABLE
```

To access elements of a static array just use square brackets and index number as normal.

```
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[i]
```

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

```
MY_DATA_TYPE.variablename
```

Using dynamic arrays

Dynamic array types are created by adding '_array' to a data type. When passing a dynamic array variable to the following functions place an & in front of the array.

```
/* Allocate own array */
vacancy_array vacancy_list;
/* And initialise */
init_vacancy_array(&vacancy_list);
/* Reset an array */
reset_vacancy_array(&vacancy_list);
/* Free an array */
free_vacancy_array(&vacancy_list);
/* Add an element to the array */
add_vacancy(&vacancy_list, var1, var2, var3);
/* Remove an element at index index */
remove_vacancy(&vacancy_list, index);
/* Copy the array */
copy_vacancy_array(&from_list, &to_list);
```

Using model data types

```
/* Allocate own 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);
```

0.4.2 Sending and receiving messages

Messages can be read using macros to loop through the incoming message list as per the template below. Message variables can be accessed using an arrow '->'

```
START_MESSAGENAME_MESSAGE_LOOP
```

```
 messagename_message->variablename
FINISH_MESSAGENAME_MESSAGE_LOOP
```

0.5 Model Execution

.1 XML DTD