Tutorial
Agent-based Programming using FLAME

Bielefeld University
12-15 Oct 2010

J.-Prof. Dr. Sander van der Hoog
Lerhstuhl für Wirtschaftstheorie
svdhoog@wiwi.uni-bielefeld.de
Overview for today

• Schedule and organization

• What is agent-based modelling?

• Introduction to C programming language

• Introduction to FLAME Framework

• Simple exercises
Schedule and organization

14.00 - 15.00: C - theory
15.00 - 15.45: C - exercises

15.45 - 16.00: Break

16.00 - 17.00: FLAME - theory
17.00 - 18.00: FLAME - exercises
What is FLAME?

• FLAME stands for: Flexible Large-scale Agent Modelling Environment

• An easy agent-based modelling environment

• Based on a formal model of computation: X-machines

• Used in a European research project: EURACE on Agent-based Macroeconomics

• Other projects range from biology to crowd simulation: cells to organisms

• Uses the C language as a basis
What are agents?

• Agents are software objects that have states and rules to change states (transition functions).

• Agents are:
  – Identifiable (distinct from other agents)
  – Autonomous (make decisions)
  – Purpose-driven (have goals)
  – Adaptive (change behavior)
  – Interactive (communicate)
  – Social (part of agent society)
What is an Agent-Based Model?

- Agent-Based Models (ABMs) are software systems consisting of agents.

- A generative (bottom-up) approach

- Benefits:
  - Involves natural description of a system
  - Flexible and extensible
  - Can reproduce emergent phenomena
Model development cycle

ABM Design
- Problem formulation
- Conceptual model
- Quantified model

ABM Development
- Model prototyping
- Architectural Design
- Agent Rule Design
- Implementation
- Verification and Validation

ABM Use
- Experimental design
- Data Initialization
- Simulation
- Analysis, Visualization
- Results Presentation
FLAME Tutorial 1: Understanding FLAME
FLAME consists of two components

• **Xparser**
  - Tool that generates application based on defined model
  - Can generate both serial and parallel simulations
  - Generates state diagrams
  - Generates Makefile (automate compilation)

• **Message Board Library**
  - Supporting library that handles data management
  - Enables agents to interact efficiently with environment
  - Allows the simulation to be run in parallel
Using FLAME

1. Describe the model
2. Code behaviour of each agent
3. Generate simulation code using the Xparser
4. Build the executable using “make”
   • Compiles code to object files
   • Links with necessary libraries, include Message Board library
5. Run the executable on an initial population
6. Observe results
Creating a model

What do you need to define?

- **Agents**
  - Memory
  - Behaviour
  - States

- **Messages** (information flow between agents)

- **Optional extras**
  - Environment constants (model parameters)
  - Custom data types
  - Custom time units
  - Function activation conditions
Agents defined based on formal concept of Communicating Stream X-Machines (CXSM)
Agents communicate through message boards.

Firms post job vacancy messages

Households read list of vacancies

Households can selectively read messages using filters.
- salary > 1000 and skill level = 4
For **efficiency** and **consistency**, there are some constraints:

- No direct agent-to-agent communication
  - All communication goes through the message board
  - Use message filters to achieve same effect

- Agents **cannot** remove messages from boards
  - Boards are automatically cleared after each iteration!

- Agents cannot **both** write and read the **same** board within a function (same type of message)
  - Use separate functions. First write, then read.
Parallelism in FLAME

“Agents interact with other agents only through messages”

Parallelism achieved by:
• Distributing agents across multiple processors
• Ensuring that agents have equal access to messages
Features of FLAME

- Simple C functions are re-usable blocks of code.
- Use messages for agent communication.
- Agent functions are automatically scheduled and activated (by layers, conditions).
- Message filtering based on agent/message variables.
- C source code linked to XML
- Nested hierarchical models
Questions so far?
Tutorial 2:
Functions and stategraphs
Agents are prototypes, not individuals! (E.g. Firms, Households, etc.)

- Functions are separated by states
- Modular code: functions can be re-used multiple times in stategraph
- BUT: no cyclical function dependencies!
Agent interaction by messages

- Agent type 1, Function A sends a message
- Agent type 2, Function D reads the message
- Sending function is always (!) above reading function
- All agent 1s have to finish Function A before Agent 2s can start Function D.
Function layers

• Sending and reading functions cannot be in same function execution layer
• Solution: Function D is shifted down automatically to the next execution layer
Function layers (3)

- Further shifting occurs by multiple messages
**Stategraph**

- **Same agent type** can also send a message to itself (intra-type communication)

- As long as reading function is below sending function

- Examples:
  - social relationships
  - consumer word-of-mouth
  - inter-firm communications
Tutorial 3: Main functionality
Learning Objectives

- Overview of XML structure
- Extra features
- Advanced examples
Model structure

- Model description file: `model.xml`
- Header: descriptive part
- Models
- Environment
- Agents
- Message definitions

```xml
<xmodel version="2" >
  <name>
  <version>
  <description>
  <models>
  <environment>
  <agents>
  <messages>
</xmodel>
```
Nested models

- File: relative path to model.xml
- Enable/disable
- Agents can be defined across different nested model.xml files

```
<models>
  <model>
    <file> [Path/model.xml] </file>
    <enabled> [true|false] </enabled>
  </model>
</models>
```
Environment tag

- constants: global parameters
- functionFiles: link in C code
- timeUnits: user-defined
- dataTypes: user-defined

<environment>
  <constants>
  <functionFiles>
  <timeUnits>
  <dataTypes>
</environment>
Constants

- Defined same as variables
- Type
- Name
- Description

<constants>
  <variable>
    <type>
    <name>
  </variable>
  ...
</constants>
FunctionFiles

**XML:** C source code is linked in by function files

```xml
<functionFiles>
  <file>Source_code_file_1.c</file>
  <file>Source_code_file_2.c</file>
</functionFiles>
```

**C:** In code files, headers are included to link FLAME generated code:

```c
#include "../header.h"
#include "../Agentname_agent_header.h"
#include "../my_library_header.h"
```
Time Units

- XML Environment defines time units
- Functions can have time conditions
- Period: periodicity of time unit, based on fundamental time unit iteration

```
<timeUnits>
  <timeUnit>
    <name>daily</name>
    <unit>iteration</unit>
    <period>1</period>
  </timeUnit>
  <timeUnit>
    <name>monthly</name>
    <unit>daily</unit>
    <period>20</period>
  </timeUnit>
</timeUnits>
```
DataTypes

- Similar to C structures.
- Used in agent memory for variables.
- Can be used in C functions as well.
Agents

- Name (archetype name)
- Description
- Memory
- Functions

```xml
<agents>
  <xagent>
    <name>Firm</name>
    <description></description>
    <memory>
      <functions>
        ...
      </functions>
    </memory>
  </xagent>
</agents>
```
Memory

- Variable
- Type
- Name
- Description

```xml
<memory>
  <variable>
    <type>int</type>
    <name>id</name>
    <description>...</description>
  </variable>
  <variable>
    <type>double</type>
    <name>xvar</name>
    <description>...</description>
  </variable>
</memory>
```
Functions

- Name
- Description
- Current state
- Next state

```
<functions>
  <function>
    <name>Agent_Function_1</name>
    <description>...</description>
    <currentState>00</currentState>
    <nextState>01</nextState>
  </function>
</functions>
```
Messages

- Name
- Description
- Variables

```xml
<messages>
  <message>
    <name>Message_Name</name>
    <description></description>
    <variables>
      <variable>
        <type>int</type>
        <name>firm_id</name>
        <description>...</description>
      </variable>
    </variables>
  </message>
</messages>
```
Extra features of FLAME

- Simple C functions are re-usable blocks of code.
- Use messages for agent communication.
- Agents are automatically activated (turns).
- Function activation based on time schedules, or event-based.
- Message filtering based on agent/message variables.
- Message pre-sorting, randomization by FLAME
Tutorial 4: Function conditions
Function conditions

Functions are activated based on conditions:

- Time-based: add a time condition
- Event-based: add a memory condition

Important note:

- From every state, all possible branching conditions must be mutually exclusive.
- From every state, there must be at least one condition that is true (else the agent reaches its end state).
Function activation: memory conditions

<function>
<name>
<description>
<currentState>start_Person</currentState>
<nextState>end_Person</nextState>
<condition>
  <lhs>
    <value>a.age</value>
  </lhs>
  <op>GT</op>
  <rhs>
    <value>0</value>
  </rhs>
</condition>
</function>
Function activation: time conditions

Time conditions can also refer to agent memory variables:

```xml
<condition>
  <time>
    <period>yearly</period>
    <phase>a.birthdate</phase>
  </time>
</condition>
```
Messages

- Messages are input/output to functions

```
<function>
  <name>Agent_function_1</name>
  <description>...</description>
  <currentState>00</currentState>
  <nextState>01</nextState>
  <inputs>
    <input>
      <messageName>Input_mesg</messageName>
    </input>
  </inputs>
</function>
```
Message loops

- **C**: Messages are added to the message board
  
  ```
  add_<messagename>_message(var1,var2,...)
  ```

- **C**: Agents are looping through the messages
  
  ```
  START_<MESSAGENAME>_MESSAGE_LOOP
  <Messagename>_message->variables
  FINISH_<MESSAGENAME>_MESSAGE_LOOP
  ```

- Note: **all** messages are processed; filtering **can** be done!
Households can selectively read messages using filters.

- salary > 1000 and skill level = 4
Message filters

**XML**: Messages can be filtered using filter conditions on the **input** messages

- Filter operators: EQ (==), NEQ (!=), LT (<), GT (>), IN (arrays)
- Use: value of a memory variable (OP) value of a message variable.

**Example:**

```
<filter>
  <lhs><value>a.id</value><lhs>
  <op>EQ</op>
  <rhs><value>m.id</value></rhs>
</filter>
```
<input>
<messageName>Message_Name</messageName>
  <filter>
    <lhs>
      <value>a.id</value>
    </lhs>
    <op>EQ</op>
    <rhs>
      <value>m.id</value>
    </rhs>
  </filter>
</input>
Tutorial 5: Linking XML and C code
Linking XML to C

- FLAME generated header files included in the C source code
- C source code files are included in the XML model
- C source code:
  - Using memory variables: CAPITALIZED names
  - All memory variables retain their values
  - Looping over messages: FLAME message loop code

- Nested models: can be enabled/disabled in XML
Including Function Files

- Assume: each agent has its own functions file.

- **XML:** C source code is linked in by function files

```xml
<functionFiles>
  <file>Source_code_file_1.c</file>
  <file>Source_code_file_2.c</file>
</functionFiles>
```

- **C:** In C source code files, headers are included to link to FLAME generated code:

```c
#include "../header.h"
#include "../Agentname_agent_header.h"
#include "../my_library_header.h"
```
Message looping in C

- **C**: Messages are added to the message board

  ```
  add_<messagename>_message(var1,var2,...)
  ```

- **C**: Agents are looping through the messages

  ```
  START_message_name_MESSAGE_LOOP
  X = <message_name>_message-> msg_var1;
  Y = <message_name>_message-> msg_var2;
  FINISH_message_name_MESSAGE_LOOP
  ```

- **Note**: by default **all** messages are processed; however: filtering **can** be done!
Using memory variables

- **C**: Memory variables are capitalized in C to retrieve and assign values:

```c
MY_VAR = 0;
int x = MY_VAR;
```

- **C**: Arrays and datatypes:

```c
X = MY_ARRAY_VAR[ i ];

x = MY DATATYPE . var1;
y = MY DATATYPE . var2;
```
Summary

- Simple C functions are re-usable blocks of code in the XML model definition.

- Agent functions are automatically scheduled and activated (by layers, conditions).

- Use messages for agent communication.

- Function activation based on time schedules, or event-based (agent variables).
- Message filtering is based on agent or message variables.
- Message pre-sorting, randomization can be done by FLAME.