FABLES: A Functional Agent-Based Language for Multi-Agent Simulations

Extended Abstract for PROMAS 2005, Budapest

László Gulyás ♦, Sándor Bartha ♦

♦ AITIA, Inc, H-1117, Budapest, Infopark sétány 1., Hungary
{lgyulas}@aitia.ai

♦ Simulation Centre, Cooperative Research and Education Center, Faculty of Informatics
Loránd Eötvös University, H-1117, Budapest, Pázmány P. sétány 1/c, Hungary
{sanka}@inf.elte.hu

Introduction

This presentation is concerned with the problem of creating computational simulations with agents. Agent-based modeling (ABM) and multi-agent simulation (MAS) has gained popularity in many disciplines, especially in the social sciences. Studying, as it is commonly called, ‘artificial societies’ provides a powerful and unique means to experiment with and to understand the processes underlying many complex phenomena in systems involving many interacting individuals.

Agent-based models are computer programs where special computational entities, the agents, represent the interacting individuals of the system. These agents are independent ‘behavioral engines’ that satisfy the so-called ‘weak notion of agency’. [15] However, their behavior is typically fairly simple and synchronized, and thus they are usually very different from ‘AI-agents’. Especially, since the main challenge in ABM is to understand and to explain the complex behavior of the system given the local rules, the agents’ behavior is typically very simple, usually in lack of sophisticated knowledge representation and communication.

Various software tools and packages exist to support the task of agent-based modelers. While opinions vary about the exact skill-set of an optimal agent-based modeler, there is an apparent want for programming excellence. Unfortunately, most ABM’ers come from the social disciplines and lack a solid programming background. This creates an increased need for an easy-to-use ABM toolset.

Swarm-alike modeling packages (e.g., Swarm, MAML, RePast, Ascape, or MASON [7][8][9][11][13]) require the usage of general-purpose programming languages (Java, Python, Objective-C, etc.) and thus their users are forced to learn programming at some level. On the other hand, various model building tools (such as NetLogo or AgentSheets [1][14]) demonstrate that by limiting the ‘space’ of possible models, the task of modeling can be efficiently assisted. The real challenge is to bridge the gap between the potential open-ended nature of Swarm-alike modeling environments and the ease of use provided by NetLogo-like frameworks. Graphical model building interfaces for Swarm-alike ABM platforms, such as Repast.py (developed for RePast, formerly known as SimBuilder) and the Visual Swarm Builder (VSB) are attempts at this. [10] Still, they impose certain limitations with respect to the models that are possible to build with them, and, at the same time, require a certain level of programming knowledge.
The Functional Agent-Based Language for Simulations (FABLES)

The motivation behind FABLES is to improve upon the current situation. One crucial observation is that a large part of the code of a typical ABM is concerned with observation (collecting statistics, visualization, etc.). On the other hand, the assembly of the observation machinery can be assisted in interactive ways. Both SimBuilder and VSB, or Ascape and NetLogo provide examples to this effect. As a consequence, in FABLES tools for I/O are kept at the minimum, observation is delegated to the Integrated Modeling Environment developed for the language (see below).

Another observation is that part of the difficulty in creating and communicating agent-based models stems from the fact that the formalism used to describe models in research papers or in oral presentations is far from the language of implementation. Moreover, the model's actual source code is full of what may be called ‘accidental representation’. These are algorithms and data structures that translate the conceptual model's notions into programming constructs. These are ‘accidental’ elements, because they are normally developed without much thought, since modelers tend to focus on model details, instead of studying computer science textbooks. They are often based on words-of-mouth information (or more specifically, advices given and taken via e-mail distribution lists), rather than on solid software engineering knowledge. These are the parts of the model where programming skills count the most.

**Design Goals**

The Functional Agent-Based Language for Simulation (FABLES) attempts to improve ‘accidental’ representations by providing a language in which models can be described as close to the conceptual model as possible. Our starting point when designing FABLES was to follow the language that ABM publications use to describe their models. The typical intended user of FABLES is a modeler with limited programming knowledge, but with sufficient skills in algebra and calculus to read a research paper. The design goals of FABLES can be summarized as follows.

1. The FABLES source should be easily readable for readers familiar with the basic mathematical formalism.
2. The language should have a precise semantics and the source should be the exact specification of the model.
3. The FABLES source should be as close to a ‘publishable’ model description as possible.
4. FABLES models should be executable.
5. The model description should focus on the nature of the model and leave implementation to the compiler.
6. The language should be general enough to accommodate possibly any agent-based model, but should focus on the common techniques and methods.

**The Current State of FABLES**

The design goals listed above partially contradict each other. Nonetheless, we attempt to achieve an optimal balance among them. We believe that the ideal model description language is built upon a functional base that is close to the mathematical, algebraic formalism. Such descriptions are typically more concise than those using imperative languages. To get even closer to mathematics, we replace programming constructs like arrays, lists, etc. with respective mathematical concepts like sequences, sets, relations. Since agent-based systems are close to the object-oriented paradigm, FABLES also uses object-oriented concepts: agents are objects, agent types are object classes. The simulation’s behavior in time, however, is hard to describe in functional terms and the object-oriented framework does not help much either. This component
is best described with imperative tools. Separating dynamic behavior from representation also helps achieving a precise semantics. Therefore, the models’ event-based dynamics is described using *schedules*. The schedules contain *imperative* elements like in Swarm, RePast, or MASON. However, these imperative elements (events or actions) are *not embedded in objects* that often confuse novice users of other packages.

FABLES thus can be separated in three main parts: an object-oriented part describing the general structure of the model (the environment, the agents, etc.), a functional part (with mathematical equations, functions, sequences, and sets) providing a standard, concise means to summarize the functional relationships among the various components and their behaviors, and finally, an imperative part (assignments, object creation and destruction) with a schedule that specifies the actual dynamics.

This largely corresponds to what is found in published model descriptions. Typically, object-oriented terminology is used to describe the overall structure of the model (the actors and general concepts), mathematical language is used to picture the components’ mutual dependence, while dynamics is given either in functional form (i.e., difference/differential equations) or in pseudo-code (often using the event-based terminology and/or the concept of the scheduler).

*Examples*

The design and implementation of FABLES is work in progress. Therefore, the syntax of the language is not finalized yet, like the main concepts may also change in the future. The examples below were prepared using FABLES v0.2.

**Random Walk**

The first example is a simple random walk on a two-dimensional lattice, performed by 100 agents.

```plaintext
model randomwalk;
agentNum = 100;

class Agent begin
  pos ; // is Integer x Integer
  schedule step cyclic 1 {
    2 : pos := pos + discreteUniform([-1..1, -1..1]);
  }
end

schedule init {
  0 : seed(0);
  0 : display:=load("user.Display3",-20,20,-20,20);
  1 : [ new Agent[ pos:=[0,0] ] : i is [1..agentNum] ];
}

///////////////////////////////////// OBSERVER //////////////////////////////////////
display;
schedule Observer cyclic 1 {
  2 : display([ a.pos : a is Agent ]);}
end
```

**Conway’s Game of Life**

The second example is Conway’s famous Game of Life model. [2]

```plaintext
model Life;
```
Mousetraps

The next example is the Mousetraps model, known from Swarm distributions, which is a cartoon demonstration of chain-reactions. Mousetraps are located at fixed positions on a 2D lattice, each of them having 2 "ping-pong" balls placed on it. When a mousetrap is hit by a falling ping-pong ball, it is “triggered”: it releases its balls into the air. They will eventually come down at random locations on the grid, hitting and triggering other mousetraps. In the model, tossing balls into the air to trigger other mousetraps is accomplished by picking random locations on the lattice, and scheduling events in the “near” future that will trigger them.

model Mousetrap;

worldSize = 10;  // Model parameter

// Shorthand for the space
world = [1..worldSize, 1..worldSize];

mousetrapsFired;  // Counter
// The agents
class Mousetrap begin
  pos; // is Integer x Integer;
  hasFired; // is Boolean;

  activate = (hasFired == false) => [
    mousetrapsFired := mousetrapsFired + 1,
    println("Activated"),
    hasFired := true,
    generateTriggers
  ]
  otherwise =>
    println("Dummy");
end

// Initialization
schedule init {
  0: graph := load("user.SequenceGraph", "#Mousetraps Fired", 0,200,0,100);
  0: seed(1);
  1: [
      new Mousetrap[ pos := [i, j],
        hasFired := false
      ] :
        i is [1..worldSize],
        j is [1..worldSize]
    ];
  1: mousetrapsFired := 0;
}

// Main schedule
schedule mainSchedule {
  2: discreteUniform(Mousetrap).activate;
}

generateTriggers = addEvent( mainSchedule, 1, triggers );
triggers = { a.activate :
  a is Mousetrap,
  b is RndPositions when a.pos(1)==b(1) and a.pos(2)==b(2)
} where ( RndPositions = {discreteUniform(world) : a is [1..2]} );

/////////////////////////// OBSERVER ///////////////////////////

graph; // variable to store the graph object
schedule observer cyclic 1 {
  2: graph([mousetrapsFired]);
}
end

Schelling’s Segregation Model

Our last example is Thomas Schelling’s famous model of residential segregation.[12] In this model, two kinds of agents (henceforth called ‘reds’ and ‘blues’) are placed on a two-dimensional toroidal lattice. The lattice is interpreted as a city, each square representing a house or a lot. The agents are families of different ethnicities. The neighborhood of an agent occupying any location on the lattice consists of the eight squares adjacent to this location. The agents’ happiness depends on the ratio of different color neighbors, each agent having a specific threshold. Unhappy agents move to the closest empty location that satisfies their expectations.
model Schelling;

// Model parameters
worldSize = 10;
agentNum = 70;
threshold = 0.6;

// Constants
red = 1;
blue = 2;
color = {red, blue};

// Spatial Constructs
world = [1..worldSize, 1..worldSize]; // The world
occupied = {a.pos : a is Resident};  // Occupied positions
empty = setMinus(world, occupied);   // Empty positions

// Helper function to implement a thorus
norm(x) = a<1 => a+worldSize
    otherwise a where (a = x mod worldSize) ;

// Neighborous relation among locations
neighborous(x, y) = { [norm(x+dx), norm(y+dy)] :
    dx is [-1,0,1],
    dy is [-1,0,1]
    when not (dx==0 and dy==0)
};

// Manhattan distance function
d(a, b) = math.abs(a(1)-b(1)) + math.abs(a(2)-b(2)) ;

////////////////////////// INIT ///////////////////////////
schedule initModel {
    0 : seed(0) ;
    1 : [ new Resident[ c := discreteUniform(color),
                        pos := discreteUniform(empty),
                        t := threshold
                        ] : o is [1 .. agentNum ]
        ] ;
}

// The agents
class Resident begin
    pos ; // is world;
    c;    // is color;
    t;    // is [0.0..1.0];

    neighbors = {a is Resident when
        a.pos in neighborous(pos(1),pos(2))};
    sameNeighbors = {a is neighbors when a.c == c};
    utility = try( size(sameNeighbors)/size(neighbors), 1.0 );
    closestEmpty = empty(minPlace({d(pos, loc) : loc is empty}));

    schedule Step cyclic 1 {
        2 : pos := ( utility >= t => pos
                        otherwise => closestEmpty ) ;
    }
}
end
MASS: The Context of FABLES

Multi-Agent Simulation Suite (MASS) is a solution candidate for modeling and simulation of complex social systems. It provides the means for rapid development and efficient execution of agent-based computational models. The aim of the Multi-Agent Simulation Suite project is to create a general, web-enabled environment for versatile multi-agent based simulations. The suite consists of reusable core components that can be combined to form the base of both multi-agent and participatory multi-agent simulations. The project also aims at providing a comfortable modeling environment for rapid simulation development. To this end, the suite will offer a high-level programming language dedicated to agent-based simulations, and a development environment with a number of interactive functions that help experimentation with and the finalization of the model.

The Multi-Agent Simulation Suite (MASS)

The Multi-Agent Simulation Suite has four major components:

---

Figure 1. The architecture of the MASS

The Multi-Agent Core

The Multi-Agent Core (MAC) is an execution environment for agents. This J2SE-based module provides the basic infrastructure (time and event management, agent-agent and agent-environment interactions, logging and playback functions, etc.) for multi-agent simulations. The MAC differs from common packages for agent-based simulation (like Swarm, RePast, Ascape or MASON) in several ways. Most importantly, it is a fully web-enabled environment¹. This means

¹ There will be a standalone version, too.
not only that pre-written simulations can be run from a browser, but also that the assembly and configuration of models (from pre-existing components like agent and environment types) is also possible via the web. This may be especially useful in educational settings, where novice modelers can experiment with model templates and pre-written components. Moreover, the web-enabled nature of the MAC platform lends itself naturally as a base for the participatory extension discussed below.

Figure 2: The administration interface of the MASS

The Participatory Extension

Participatory Simulation is a methodology building on the synergy of human actors and artificial agents, excelling in the training and decision-making support domains. [4] In such simulations some agents are controlled by users, while others are directed by programmed rules. The Participatory Extension (PET) is an add-on to the Multi-Agent Core that allows users to take control over agents in the simulation. The J2EE-based extension provides solutions for communications between the client and the main simulation server, including visualization (two-dimensional ‘bird’s eye’ view, three-dimensional ‘world view’, etc.) at the client side. The user interface of the simulation client uses standard web technology, allowing for easily customizable layout and design.

Figure 3: The simulation selection interface of the MASS

The ‘FABLES’ Simulation Definition Language

As discussed above, the Functional Agent-Based Language for Simulation (FABLES) is intended for the concise and efficient definition of agent-based models. FABLES combines the strengths of functional programming with the object-oriented paradigm, providing unique means to implement agent-based simulations.
FABLES' vision is an abstract formalism to describe agent-based models. Models defined in this language could, in principle, be automatically transformed into agent-based simulations in RePast, Swarm or Ascape, etc. Such a description would be ideal to publish concise definitions of agent-based models. Moreover, independently developed compilers to different modeling platforms in place, the formalism could also help making the replication/docking of computational models a routine task. In the current, prototype version, FABLES models are interpreted. Our future plans include a compiler that generates (optimized) code for the MAC core.

Figure 4: The user interface of simulations in the MASS

Integrated Modeling Environment
The Integrated Modeling Environment makes model development in FABLES more effective by providing a modeler-friendly editor with syntax-highlighting, on-the-fly syntax checking, and a number of exporting options. The environment will also have interactive wizards that help
collecting and charting statistics about the model. This will be completed by wizards that let the modeler interactively set up two- and three-dimensional displays of the model.

Summary
In this paper, we discussed the FABLES language for simulations and the MASS, its development context. In its current prototype version, FABLES is an interpreted language. However, in the long run, models developed in FABLES will be compiled to pure Java code. This way they will be seamlessly integrated in MASS. Obviously, the FABLES compiler may never generate code as efficient as an experienced programmer’s. Still, using FABLES may be a viable option for smaller scale, exploratory models. Also, using FABLES will force making efficiency considerations explicit, especially when they depart from the conceptual model.

References