The TurtleKit Simulation Platform: Application to Complex Systems

Fabien Michel
CReSTIC/LERI
rue des crayères BP 1035
51687 REIMS Cedex 2 FRANCE
fmichel@leri.univ-reims.fr

Grégory Beurier   Jacques Ferber
LIRMM
C.N.R.S. - Université Montpellier II
161 rue Ada, 34392 Montpellier Cedex 5 - France
beurier@lirmm.fr   ferber@lirmm.fr

Abstract

This paper presents the TURTLEKIT *simulation platform. This platform relies on the combination of a Logo simulation model with high level programming languages. Indeed, the agent model and the environment are largely inspired by the StarLogo system but can be easily extended thanks to the use of the Java programming language. This paper presents the* TURTLEKIT *main features and illustrate them through experiments on artificial complex systems.*

1 Introduction

Agent-based simulations (ABSs) is an attractive alternative to equation-based models [17]. ABSs allow to directly model individuals, their behaviors and their interactions and thus to model complex systems [6]. There is today plenty of tools which provide different means to develop, execute and analyze such models. Most of the existing platforms have been designed according to a particular area of research such as ecology (CORMAS [5], ECHO [10]), robotic (MISSIONLAB [14]), multi-agent coordination (MASS [9]), etc. Still, some platforms are not dedicated to a research field but address a particular audience. For instance, SWARM [3] defines a generic simulation mechanism and affords important libraries of environment, agent architectures, simulation tools, etc. The underlying idea is to give advanced programming tools that enable to develop simulators that fit the experiment requirements. So this platform is designed for advanced users. At the opposite, other platforms such as STARLOGO [2] or NETLOGO [1], which are based on the Logo language, have been originally designed to quickly develop simulations for educational purposes. Thus, they intentionally focus on simplicity and do not afford advanced programming tools for extending the platform possibilities. They are designed for newbie users.

This paper presents the TURTLEKIT platform [15], which relies on a Logo-based simulation model. However, TURTLEKIT keeps the simplicity of Logo programming approaches while giving all the possibilities afforded by high level programming languages. So TURTLEKIT is dedicated to both newbie and advanced users. The paper first presents TURTLEKIT main features and then shows how it can be used and extended.

2 TURTLEKIT overview

2.1 Logo programming approach

The Logo language was designed in the late sixties at the Massachusetts Institute of Technology (MIT). The motivation was to provide a learning tool that did not require programming skills. The principle is to manipulate a graphical animat, a *turtle*, typing simple commands to make the turtle move and draw shapes on the screen. For instance the *forward 10* command makes the turtle move forward 10 steps. Logo programs are usually collections of small procedures that define turtle behaviors which can be combined to achieve more complex behaviors

In platforms like STARLOGO or NETLOGO, the idea is to provide a simulation environment made of several turtles (MultiLogo), and thus to model multi-agent systems (MAS). TURTLEKIT is directly inspired by this approach. Turtles are agents that live on a two-dimensional world which is spatially discretized into *patches*. Each patch contains local properties such as its color, pheromone concentrations, etc. Turtles have the ability to interact with the world, other turtles and patches, using *turtle commands*: they can move, perceive their local environment and modify the properties of patches and other turtles. So, implementing a simulation with MultiLogo-like languages relies on defining *procedure commands* which are collections of primitive commands and other *procedure commands*:

```
to wiggle
  fd 1    rt random 50   lt random 50
end
```

For instance the *wiggle* procedure consists in using the *right turn (rt)*, *left turn (lt)*, *forward (fd)* and *random* primitive commands to define a random walk. The wiggle procedure can then be used into another new procedure to define a more complex behavior and so on.

2.2 Using high level programming languages

The main advantage of MultiLogo platforms is that they propose an agent-based programming language that focuses on simple agent/environment models. Despite this simplicity, they provide an exhaustive list of primitive commands and very complex and interesting behaviors can be easily obtained starting from simple primitive commands. However, MultiLogo platforms have been designed to ease programming and they intentionally do not afford advanced programming tools which could be used to extend the platform capabilities. For instance, it is not possible to display the world according to various point of views simultaneously whereas having various graphical representations of the world is crucial when studying complex systems [5]. On the other side, high-level platforms provide code extension facilities (inheritance, modular programming, etc.).

Developing TURTLEKIT, our main goal is to bridge the gap between using Multi-Logo models and using high level programming features. Hence, TURTLEKIT is an open source platform written in Java providing a default set of classes which can be used or extended regarding the simulation requirements. Thus programming facilities such as inheritance are naturally available.

2.3 Agent oriented programming

Initially, TURTLEKIT has been developed to validate the simulation tools we implemented in the MADKIT (www.madkit.org) multi-agent platform [16]. Since MADKIT is a generic MAS development environment, it provides support to design organizations, interactions between agents, debugging, system probing, distribution, etc. TURTLEKIT naturally benefits of these features. Moreover, TURTLEKIT is a MAS itself and every module of TURTLEKIT is a regular MADKIT agent having the ability to communicate with any other agent using messages. For newbie users, all these features are encapsulated and can be ignored. However, advanced users can use these features to extend the default tools of TURTLEKIT (see section 3.2).

2.4 TURTLEKIT turtles

TURTLEKIT turtles are written in Java. Every kind of turtle is implemented in single Java class that inherits of the super class *Turtle* that implements the basic Logo-like procedures. Implementing TURTLEKIT turtles simply consists in defining Logo procedures using the Java programming syntax. So, the *wiggle* procedure code is as follows:

```
public void wiggle(){
fd(1);turnRight(random(50));turnLeft(random(50));
}
```

Each turtle can achieve a collection of actions during a single simulation step. A turtle behavior is defined as an automaton that represents a sequence of atomic behaviors which are Logo primitive sequences. So, when activated, a turtle executes all the primitives associated with an atomic behavior Then the behavior the turtle wants to do next is returned. Figure 1 illustrates this mechanism through a classical example: the termite behavior

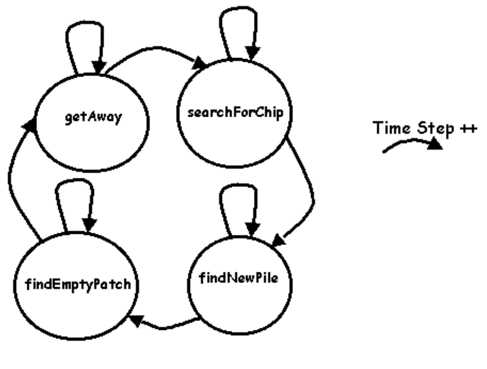

Figure 1. The termite behavior

For instance, the *findNewPile* behavior code is entirely executed when the turtle's turn comes. Then in the next simulation step, the turtle will execute either *findNewPile* or *find empty patch*. The *findNewPile* behavior is implemented as follows:

```
public String findNewPile(){
  if (getPatchColor() == Color.yellow)
    return("findEmptyPatch");
  else{
    wiggle(); return("findNewPile");  } }
```

2.5 The environment model

Although the environment is discretized into a patch grid, turtles move in a continuous way. For instance, if a turtle has a 45 degree orientation and moves of one unit (*fd(1)*), the turtle will have its x and y coordinates increased by $\sqrt{2}$. It is thus possible to make the turtles doing particular moves such as a perfect circle. The patch grid is rather used to define the turtles local environment, enabling them to perceive and act on the world using primitive commands (*setPatchColor(Color.yellow); dropPatchVariable("Pheromone",1500); etc.*). The patch grid also defines a plain diffusion model that permits to simulate the diffusion and evaporation of substances (see section 3).

2.6 Observers and Viewers

As in SWARM, it is possible to probe the model using the *Observer* agent of TURTLEKIT. The observer is also a regular MADKIT agent and can interact with any available agent within MADKIT. Especially, an observer may send data to MADKIT agents which are specialized in doing particular tasks (e.g. drawing line charts, data analysis, etc.) An observer can also create a probe according to a group/role couple to extract data according to organizational characteristic, probing only turtles playing a particular role. Users can extend the *Observer* class regarding their needs. Furthermore it is possible to launch as many observers as required (see sections 3.1.3 and 3.2.4).

To graphically represent the simulated world, we have implemented particular observers: the *Viewers*. TURTLEKIT affords a default viewer that displays the world in a classical way (the color of the patches and turtles). However, many viewers have been implemented to define new point of views: turtles displayed by role, substances displayed by types, etc. This is done by overriding the default viewer's *paintPatch* and *paintTurtle* methods. Then, through multiple representations, using several specialized viewers, it is possible to graphically extract many aspects of a model (see sections 3.1.3 and 3.2.3).

2.7 The Scheduler agent

The *Scheduler* is designed to organize the simulation actors activity: environment, observers, viewers and turtles. This agent can be ignored in the platform default configuration. However it can be extended to achieve a particular scheduling: activating a particular kind of turtle twice per turn or activating the display only when a particular condition is verified.

2.8 The Python Command Center

Java has one shortcoming: the code must be compiled and cannot be interpreted at runtime. But interacting with the simulation is a desirable feature when studying complex systems. To this end, TURTLEKIT permits the use of the Python language: we have implemented the *Python Command Center* (PCC) which allows to take control of the turtles by typing commands or loading a predefined script. Notably, this feature allows to dynamically execute compiled Java behaviors during a regular simulation to interact with the simulation (see section 3.1.5).

3 Experiments

To illustrate the capabilities of TURTLEKIT, we present two experiments involving the design of complex systems. The first experiment involves a newbie use of TURTLEKIT to produce multi-level emergence (MLE) of complex structures within a MAS. The second simulation, implemented with an advanced use of TURTLEKIT, develop an evolutionary approach to construct multi-agents organisms.

3.1 Multi-level emergence

3.1.1 Background

In nature, emergence generally exists in the form of *multi-level emergent structures* (also called multiple emergence) [7]. It is the production of emergence in a system composed of subsystems which are themselves the product of previous emergences. In [4], we have proposed a MAS model to produce MLE. This model distinguishes the environment, the interactions and the agent model. The theoretical model has been developed by simulating each step of its conception on TURTLEKIT. Those simulations involved low-level programming skills.

3.1.2 The agents

The agents interact via emission and perception of pheromones. The structuring of the emergence is the result of attractive/repulsive interactions between the agents. The agents own a state which rules the agents' attraction/repulsion behaviors by modifying the agents' interaction abilities. The agents are able to mutate modifying their own state. To develop the final MLE model, several kinds of agents have been easily implemented and tested thanks to a modular programming. The agents pattern has been define by inheriting the *Turtle* class. The subclass, called *MleTurtle*, implements the agent's behavior automaton. This automaton involves four atomic behaviors: *perception, emission, mutate, move*. Several kinds of agents have been then implemented inheriting the *MleTurtle* class. To design the final prototype, a refining of each behavior has been done through the use of numerous subclasses.

3.1.3 Using the TURTLEKIT viewers and observers

To characterize the emergent phenomenon, standard viewers have been used. The standard viewers permitted us to: (1) observe the agents as a function of their state and (2) observe the diffusion of interactionnal pheromones.

One of the important tasks for the analysis of the system behavior was to observe and characterize the emergent phenomenon. Figure 2 shows a simulation observed by four different viewers: 1, 2 and 3 display the turtles and one pheromone, 4 is a standard view. These viewers permitted us to observe the formation of circular multi-emergent structures while analyzing the role of pheromones in the emergent structuring process.

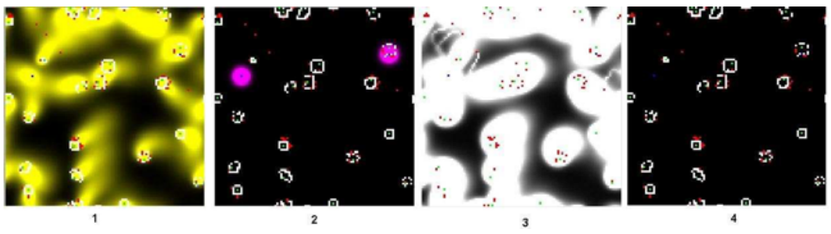

Figure 2. Multiple representations of the world with various TURTLEKIT standard viewers.

TURTLEKIT observers have been used to analytically study the system's behaviors Various observers have been implemented to extract quantitative information from the system: counting particular agents (with a particular state, behavior or position), measuring distances between agents, making computations on quantities of pheromones (average, maxima, standard deviation, etc.), and so on. For instance, figure 3 shows a mathematical measurement of the structural organization in a typical simulation for each level of emergence (level 1 in black, level 2 in gray). This computation is based on the average of distances between agents involved in the same structures. The chart is drawn by a MADKIT agent interacting with the observer.

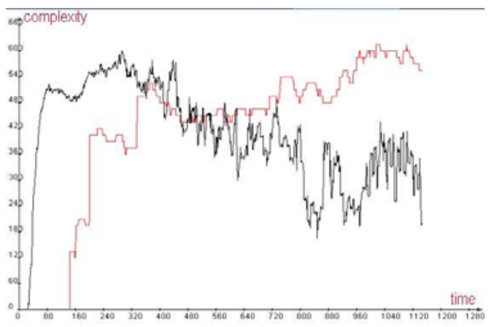

Figure 3. Structural complexity in the system.

```
synchronized public void watch(){
 Turtle[] theTurtles = getTurtles();
 for( int i = 0;i < theTurtles.length;i++ ){
   if( Turtle[i].belongsToAnEmergedStructure() ){
     Turtle myAttr = Turtle[i].attractor();
     double dist = distance(Turtle[i], myAttr);
     sendTo(dist, "ChartTracer"); } } }
```

If a turtle belongs to an emergent structure, the distance between this turtle and the core of the structure is measured and then sent to a standard MADKIT agent in charge of the chart drawing.

3.1.4 Using the TURTLEKIT scheduler

The structuring of emergent organizations relies on the perception of pheromones by the agents. However, the duration of a substance diffusion increases exponentially as a function of the crossed distance. To quicken the emergence process, we have defined a scheduler managing three diffusion processes for each simulation step. It has greatly shortened the simulation durations without altering the global behavior of the implemented systems.

3.1.5 Using the Python Command Center

To study the behavior of the system at runtime, it was important to interact with the agents during simulations. The use of the PCC permitted us to disturb the standard agents' behavior during the simulation process. For instance, to study the robustness of emergent structures, we invoked random agent moves during the simulation. Such an invocation can be done at runtime in two different ways: (1) by invoking an existing procedure in the PCC or (2) by directly using a script within the PCC.

```
def randomMove(self): #(1) Random move procedure
 self.randomHeading() # in a Python agent
 self.fd(10)
------------------------------------------------
public void randomMove(){ //(1) in a Java agent
    randomHeading();  fd(10); }
------------------------------------------------
#(1) direct call of the behavior within the PCC
askTurtles("randomMove")
------------------------------------------------
# (2) Random move using a script in the PCC
askTurtles("randomHeading()")
for i in range (0,10):
     askTurtles("fd(1)")
```

3.2 Evolutionary experiments

3.2.1 Background

Mimicking the mechanisms of complexity appearance in living systems, the field of embryogeny (growth of embryos) explores the possibilities to design artificial complex systems [12]. How a single evolved cell can give birth to complex organisms?

We have developed a bio-inspired model that makes agents evolve to form artificial organisms. This model is inspired by the functioning of particular genes in natural

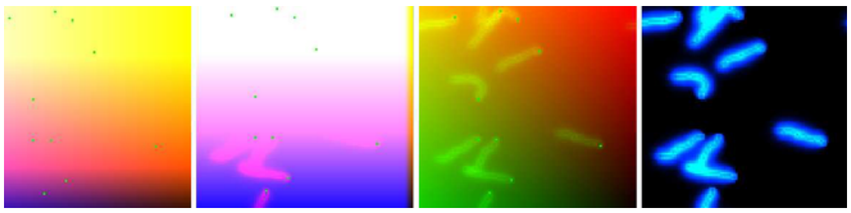

Figure 4. Multiple representations of the world with the universal viewer.

embryos development [13] and implements techniques from the evolutionary computing field [8]. The agents represent cells that replicate and interact to construct organisms.An evolution process is used to explore the space of behaviors encoded in the agents' genetic inheritance. The French flag model of Wolpert [18] has been the inspiration for the first task the model has to achieve: the goal has been to evolve a population of agents able to form organisms that create flag patterns. The evolutionary algorithm works as follows: (1) generate a initial random population (2) assign a note/fitness to the formed organism (3) select individuals according to fitness for reproduction (4) reproduce offspring using two parents (5) evaluate each offspring and assign fitness (6) replace old population by offspring using particular strategies [11] (7) unless a high level of similarity with the predefined pattern is reached, return to step 4.

Dealing with evolutionary computation requires the use of numerous tools: evolution engine, fitness manager, reproduction operators, etc. To develop such simulations, we have greatly improved the TURTLEKIT standard tools.

3.2.2 Extending the TURTLEKIT launcher

The study of evolutionary systems requires a systematic use of the simulation process. To this end, we have improved the formal description of the implemented simulations and developed *Launcher* agents able to parse XML files. The simulations are thus described by defining agents, environment, observers and viewers in XML formalisms. The XML *Launcher* has been implemented by simply overriding the initialization functions of the standard *Launcher* and by providing adequate Java libraries to parse XML.

3.2.3 Extending the TURTLEKIT viewers

New viewers have been implemented to achieve two mains goals: (1) dynamic visualization of diffusing substances and (2) had-oc interventions in the environment. We first have developed a *universal viewer* which allows the mapping of the substances concentrations with the viewer RGB (red, green, blue) canals. This technique has permitted us to provide a dynamic view of diffusing substances by defining the displayed colors as functions of substances quantities. Figure 4 shows a simulation observed by four different viewers. Every viewer monitors a lot of diffusing substances represented by various scales of the RGB colors In the first viewer, the substances dropped by agents are invisible because external substances are present in too great proportions. Thus, when scaling the visualization with the external substances, the substances dropped by agents become visible. It is important to note that the visualizations can be managed and modified at runtime (new viewers, new colors, display agents on/off, etc.).

Secondly, we have developed an environment editor that permits user interactions with the environment via mouse clicks. It allows: the instantiation of new turtles within the environment, the modification of patches colors, the interaction with simulated agents via a simple popup, etc.

These features have permitted us to design in a ad-hoc manner the predefined flag patterns required for the step 2 and 5 of the evolutionary algorithm. To evolve a French flag pattern, we have first drawn the predefined French flag by modifying the colors properties of the patches (See figure 5). Then this predefined pattern has been used as a mask to give fitness. Figure 5 shows the growth of a multi-agents organism from a single evolved agent (after 250 generations).

3.2.4 Extending TURTLEKIT observers

The *Observer* is naturally the main class to extend to manage the evolutionary process. So, we created new classes of observers, the *Managers*. *Managers* are designed to handle various types of artificial genetic objects (genes, genomes, pools, etc.). The *Managers* interact with the scheduler and the launcher to randomly generate pools of genes, give genetic inheritance to simulated agents, manage the reproduction process, save genetic objects to reuse them, and so on. The possibility to design interactions between the observers and other MADKIT agents allowed us to break through the standard functioning of TURTLEKIT. Indeed, the *Managers* have permitted us to develop a coherent evolutionary model without modifying the TURTLEKIT simulation engine.

4 Conclusion

The TURTLEKIT platform aims at providing to advanced users the simplicity of a Logo simulation model while

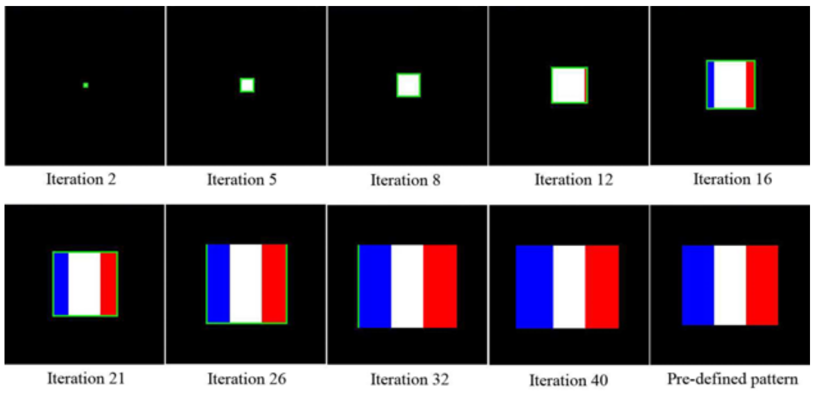

Figure 5. Growth of fittest program from a single agent to a mature French flag organism.

proposing flexibility, modularity and extensibility. We have shown the importance of using multiple views of one single simulation in order to extract the required information. We have also exhibited that agent oriented programming have permitted us to develop sophisticated simulation models without modifying the original simulation engine. In fact, we can clearly distinguish two ways of using TURTLEKIT:

- LOGO-like programming: dedicated to newbie users, this approach simply involves the definition of the behavioral automaton of the agents. All simulations tools are directly managed by TURTLEKIT.
- SWARM-like programming: for advanced users, it is possible to extend greatly the possibilities of TURTLEKIT by inheriting and modifying the existing classes, enabling (1) the extension of the platform modules (scheduling, probing, visualization, agent creation, etc.) and (2) the implementation of additional features required by a particular experiment

TURTLEKIT is available for download on the MADKIT web (www.madkit.org). It features a user guide, the TurtleKit api, tutorial simulations and advanced projects.

References

[1] The NetLogo system. http://ccl.northwestern.edu/netlogo/.

[2] The StarLogo system. http://education.mit.edu/starlogo/.

[3] The Swarm Simulation System. http://www.swarm.org/.

[4] G. Beurier, O. Simonin, and J. Ferber. Model and simulation of multi-level emergence. In *in the 2nd IEEE International Symposium on Signal Processing and Information Technology*, Marrakesh, Morocco, December 2002.

[5] F. Bousquet, I. Bakam, H. Proton, and C. Lepage. Cormas: common-pool ressources and multi-agent systems. *Lectures Notes in Artificial Intelligence*, 1416:826–837, 1997.

[6] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] F. Heylighen. Self-organization, emergence and the architecture of complexity. In *in the First European Conference on System Science*, pages 23–32, 1989.

[8] J. H. Holland. Genetic algorithms and classifier systems: Foundations and future directions. In *ICGA*, pages 82–89, 1987.

[9] B. Horling, V. Lesser, , and R. Vincent. Multi-Agent System Simulation Framework. In *In 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation, EPFL*, Lauzanne, Switzerland, August 2000.

[10] P. T. Hraber, T. Jones, and S. Forrest. The Echology of Echo. *Artificial Life*, 3(3):165–190, 1997.

[11] J. R. Koza. Introduction to genetic programming. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 2, pages 21–42. MIT Press, USA, 1994.

[12] S. Kumar and P. J. Bentley. Computational embryology: past, present and future. *Advances in evolutionary computing: theory and applications*, pages 461–477, 2003.

[13] H. Li, D. Harrison, G. Jones, D. Jones, and R. L. Cooper. Alterations in development, behavior, and physiology in drosophila larva that have reduced ecdysone production. *The Journal of Neurophysiology*, 85(1):98–104, 2001.

[14] D. MacKenzie, R. Arkin, and R. Cameron. Multiagent Mission Specification and Execution. *Autonomous Robots*, 4(1):29–52, January 1997.

[15] F. Michel. An introduction to TurtleKit: a platform for building Logo-based multi-agent simulations with MadKit. Technical Report RR 002215, LIRMM, Montpellier, June 2002.

[16] F. Michel, J. Ferber, and O. Gutknecht. Generic Simulation Tools Based on MAS Organization. In *Proceedings of the 10th European Workshop on Modelling Autonomous Agents in a Multi Agent World MAMAAW'2001*, Annecy, France, 2-4 May 2001.

[17] H. V. D. Parunak, R. Savit, and R. L. Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *Proceedings of the 1st Workshop on Modelling Agent Based Systems, MABS'98*, Paris, France, 1998.

[18] L. Wolpert. The french flag problem: A contribution to the discussion on pattern development and regulation. In *Towards a Theoretical Biology*, volume 1, pages 125–133. Waddington, 1968.