Agent-Based Modeling Using Erlang

Eliminating The Conceptual Gap Between The Programming Language & ABM

Gene I. Sher

University of Central Florida

Abstract: There is a conceptual gap between real world systems and their models created using differential equations. Numerous systems, such as traffic flow, financial markets, neural networks... are all naturally composed of agents interacting with each other and the environment, from which the behavior of such systems emerges. The more complex the interactions and agents, the more difficult, and even impossible, it becomes to model them using averages and standard mathematical approaches like sets of differential equations. A more direct approach to modeling such complex systems is by modeling the agents themselves, and letting them interact with each other and the environment, with the system's behavior then emerging naturally. This approach is called Agent Based Modeling (ABM), and the conceptual gap between real world phenomena produced by interacting agents and their respective ABMs is lower than the models created using sets of differential equations. The removal of this conceptual gap through the use of ABMs allows us to tackle larger and more complex problems and explore emergent behavior more easily. Unfortunately, ABMs and the programming languages used to create them do not have a 1-to-1 mapping, and thus, producing a conceptual gap between the two. This paper attempts to present and make a claim that we can eliminate this conceptual gap by using an actor model based concurrency oriented programming language called Erlang. In this paper we discuss how Erlang fits into ABM and Multi-Agent Simulation systems, and what features it possesses that make it such a perfect tool for the job.

1.0 Introduction:

The world is a naturally concurrent system. On almost every level of granularity, the phenomena we perceive around us is emergent from the interactions of agents. On atomic level, it is due to the interaction of leptons and hadrons. On molecular level, due to the interaction of atoms. In the more dynamic and organic systems, the phenomena we observe is due to the interaction of concurrently existing and operating intelligent organisms, animals. The emergent phenomena of intelligence is due to the interaction of Neurons, each of which on its own does not have intelligence, but is merely a spatio-temporal signal integrator. But with 100 billion of them, positioned in a particular topology, allows for intelligence, self awareness, and consciousness to emerge. The financial market for example is also an emergent phenomena of a large number of trading agents interacting with each other by buying and selling financial instruments. Describing any one of these mentioned systems would be difficult when using sets differential equations, and the complexity of the sets of differential equations would grow very rapidly with the complexity of the agents and their interaction. Yet all of these systems can be modeled by creating and modeling the simple agents directly, and letting them interact. The emergent property of those interacting agents is the phenomena we're after.

Agent based modeling is a natural way to represent real world systems. The ABM approach has already generated numerous interesting and useful results [1], and has been used within the industry to understand complex real world systems [2], and optimize the same [3]. Though ABMs are composed of concurrent agents/actors interacting with each other, they are primarily written in programming languages whose architectures are very much different conceptually from the ABMs themselves. The primary programming languages used in the industry for ABM are procedural or object oriented. It would be significantly easier to develop an ABM if the conceptual gap between it and the programming language used is minimized. It is for this reason why programs like NetLogos, Maze, and many others

were created, to ease and automate elements of the development process of ABMs. Nevertheless, these systems themselves are still developed using standard object oriented and procedural languages, building layers on top of layers of programs just to provide the user with the tools that ease ABM development, tools which are already present in programming languages that were created from the start for just such concurrent systems. One such programming language that was created from the very start for the development of multi-agent, concurrent, fully distributed systems, is Erlang.

In the following sections we briefly discuss the features and elements of agent based models, and the features and elements of the programming language Erlang. Once we have demonstrated the perfect matching between the two, we build a few ABMs to demonstrate Erlang's conciseness and perfect conceptual mapping to this domain.

2.0 Eliminating The Conceptual Gap:

Real world, ABM, and Erlang, are all conceptually similar: All three are composed of fully concurrent processes/agents/actors, interacting with each other through messages (whether they be software based structures, or electro-magnetic and gravitational waves). We first break down Agent Based Models and Multi-Agent Systems into their sub-parts, presenting their architectural features. We will then do the same for Erlang. Finally, we will then use these features to demonstrate the 1-to-1 mapping between the two.

2.1 ABM

ABM is a computational model which simulates actions and interactions of numerous autonomous agents so as to asses their effects on a system and its emergent patterns as a whole. Each agent might be relatively simple, but through their interaction, the emergent behavior of the entire system is complex. This approach for example allows us to develop simple models of trading agents, with simple rules defining how and when they sell and buy commodities. But it is through their interactions with each other that a more complex phenomena of market emerges, allowing the researcher to study and analyze that market and how it changes based on changes in agent strategies. The same way complex predator-prey models can be created, by simply modeling predator and prey agents with particular rules for hunting and foraging. When an environment is then populated with such simple and easily developed agents, the emergent behavior is complex, providing more information on environment's ability to sustain a particular population of prey or predators, and for example the effect on sustainable prey population size when we dial up the aggression level of predator agents... ABMs are not so much used as methods to predict something, than as methods to study complex emergent systems, and the change and effect on the same due to changes in agent behavior. For example models of amusement parks have been built to study them and determine which factors improve their throughput [4], models of stores have been built, to determine what aspects could be changed to improve customer satisfaction [5], models of markets, particularly the bidding granularity [6] have been made to study how the increase in bidding granularity affects it (with the surprising result that the spread increases with the increase in bidding granularity, rather than the opposite), and even panic induced crowd stampedes have been modeled using ABMs [7], demonstrating that the throughput is actually higher when we position a pillar in front of the exit, which decreases mob mentality and chaotic trampling during evacuation from a room.

Thus an ABM system is composed of concurrent, autonomous agents. The agents can interact with each other, and the environment (whether it be a 3d environment in an ALife simulation, or some trading system in a market simulation). The agents can communicate with each other by sending messages. When agents communicate with each other, the communication protocol can be designed by

the researcher, though there are also standard protocols for multi-agent based systems, such as: Knowledge Query Manipulation Language (KQML) and FIPA's Agent Communication Language (ACL), for example.

2.2 Erlang

Erlang is an actor model based, message passing paradigm, concurrency oriented programming language developed by Erricson. Erlang was created for the purpose of developing highly concurrent and distributed telecom systems. Such systems have the need for the language to preferably support the following features, as is quoted from [8]:

1. "*The system must be able to handle very large numbers of concurrent activities.*
2. *Actions must be performed at a certain point in time or within a certain time.*
3. *Systems may be distributed over several computers.*
4. *The system is used to control hardware.*
5. *The software systems are very large.*
6. *The system exhibits complex functionality such as, feature interaction.*
7. *The systems should be in continuous operation for many years.*
8. *Software maintenance (reconfiguration, etc) should be performed without stopping the system.*
9. *There are stringent quality, and reliability requirements.*
10. *Fault tolerance*"

Thus when Erlang was created, it was designed to support the following features [9]:

- *"Encapsulation primitives — there must be a number of mechanisms for limiting the consequences of an error. It should be possible to isolate processes so that they cannot damage each other.*
- *Concurrency — the language must support a lightweight mechanism to create parallel process, and to send messages between the processes. Context switching between process, and message passing, should be efficient. Concurrent processes must also time-share the CPU in some reasonable manner, so that CPU bound processes do not monopolize the CPU, and prevent progress of other processes which are "ready to run."*
- *Fault detection primitives — which allow one process to observe another process, and to detect if the observed process has terminated for any reason.*
- *Location transparency — If we know the Pid of a process then we should be able to send a message to the process.*
- *Dynamic code upgrade — It should be possible to dynamically change code in a running system. Note that since many processes will be running the same code, we need a mechanism to allow existing processes to run "old" code, and for "new" processes to run the modified code at the same time.*"

Erlang provides concurrency through independent process primitives. Which are independent micro server/clients that can communicate with each other through messages. Immediately we can see correlations between processes and agents, and the way agents communicate with each other using message passing. And indeed the mapping is close to being 1-to-1, as is discussed next.

2.3 1-To-1 Mapping

As shown in Fig-1, each autonomous and independent agent is a process. And just like agents communicate with each other through messages, in Erlang it is exactly the same for the processes. Each process communicates with another by message passing. A message can be anything, a string, a tuple, a list, a number, or any combination of the same. Thus, the message protocol is completely up to the researcher. This direct and clear mapping from the programming language to the problem domain, allows for Erlang to be used with great ease within this domain and act as a force multiplier. In the next section we will briefly discuss how exactly the processes and message passing work in Erlang.

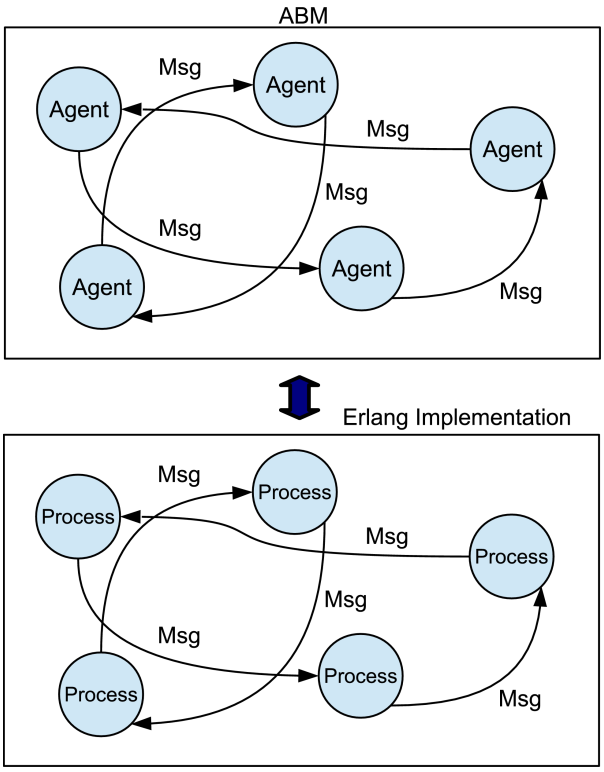

Fig-1: A. shows a population of agents communicating with each other through some communications protocol. B. Presents a population of processes, communicating with each other through message passing, where the message protocol is set by the researcher.

3.0 Multi-Agent Systems In Erlang:

3.1 Processes, Messages, and Concurrency

Whereas in a programming language like C++ the basic building block is an object, in Erlang the basic building block is a process. In some sense, a process is an object as well, you can create multiple versions of it, and other processes can be derived from some original-one using added callbacks... But a process is so much more. A process is a concurrent micro server/client, similar to concurrent processes created by an OS, but much lighter. You can have millions of concurrent processes operating at the same time. Each process has its own memory, state, and its own inbox, letting it receive messages from other processes. Any function can be spawned as its own process, and

then exist independently and concurrently with other processes. For example, the following is a function that when spawned waits for messages from other processes, and depending on those messages, executes some set of procedures. This process can accept 3 types of messages: 1. A message composed of a single word/constant: 'terminate', which asks the process to terminate and makes it shut down when it receives it. 2. The other message is of a tuple form: {area, Radius}, where Radius is a variable, and area is a word/constant. 3. The third message that the process understands is simply a single word/constant 'identify', which when received prompts the process to print the following string to console: "My Process Id (PId) is: <PId_Value>, and I am online". The words: terminate, area, and identify, are a special kind of string-constant values which are called atoms in Erlang.

```
simple_process()->
	receive
		terminate ->
			ok;
		{area,R} ->
			Area = math:pi()*R*R,
			io:format("The area of a circle of radious: ~p is: ~p~n",[R,Area]),
			simple_process();
		identify ->
			io:format(""My Process Id (PId) is: ~p, and I am online~n",[self()]),
			simple_process()
	end.
```

The function io:format/2 prints a string and the parameters specified in the parameter list (bracket enclosed), to the shell. A message sent to a process using its process identification (PId), goes to that process' mailbox. To extract a message from the mail box the keyword 'receive' is used. This allows the process to extract all messages sent to it one at a time, and pattern match them against the patterns/message-forms it understands, which in this case are the 3 above noted message structures: terminate, {area,R}, and identify. Due to Erlang's support for pattern matching, in this manner it is very easy to compose various communication protocols. Finally, each concurrent process has its own space on the stack, and each micro server/client is tail recursive. Meaning, after executing some functionality, if the last call in a function is back to itself (tail recursion), it simply goes back and starts its functionality from the start, looping back on itself. In computer memory, this is a simple jump back to the initial position of its program on the stack, which allows each process to have a very small footprint and be highly efficient.

Spawning such concurrent processes is very simple, done by executing the command: `spawn(ModuleName,FunctionName,ParameterList)`, which returns the PId of the spawned process, which then lets us send it messages. The arguments to the spawn function are: ModuleName which is the name of the file in which the function/process is located, the name of the function FunctionName that we want to spawn as a process, and finally the ParameterList, which is a bracket enclosed list of arguments. One can spawn a single process, or thousands of them, getting back a list of PIds.

These processes are not the same as operating system processes, rather Erlang provides its own shell, with the processes being much lighter. It is this that allows Erlang to support millions of processes on a single node, and there are currently even projects to allow Erlang to run on nearly bare metal [10], due to the fact that the Erlang shell provides most of the features needed for the language to work.

3.2 Message Passing

The sending of a message to a process is also very simple, done using the following syntax: `PId ! Msg`, where PId is the Process Id, and Msg is the message you wish to send to the process, which can be of any form: a string, an atom, a number, a tuple, a list, or even a function (Erlang is a functional

programming language as well). The following three lines present the sending of the 3 messages that the simple_process() is capable of understanding. The below code first spawns the simple_process, and then sends the identify, {area,R}, and finally the terminate message:

```
PId = spawn(simple_module,simple_process,[]),
PId ! identify,
PId ! {area,3},
PId ! terminate.
```

In this manner any process can communicate with any other. Hundreds or millions of processes can be spawned, existing concurrently and communicating with each other through messages, with an easy to define communication protocol.

3.3 Scapes & Agents

From the above two subsections we can see that an agent is a process, and communication between them is done through messages. In just a few lines of code, we can represent these agents and their communication protocol. There is almost no superfluous code, almost all of it maps directly to the ABM. The simple_process/0 that was presented above was an example of a passive agent. We can add a function to this agent that does some form of cognitive processing, potentially through neural networks or some other approaches. In a number of models though, for example ones like Sugar-Scape or Schelling's Tipping Model, there must also be an environment or scape. The scape would also then be a process, with the entire system having the architecture shown in Fig-2.

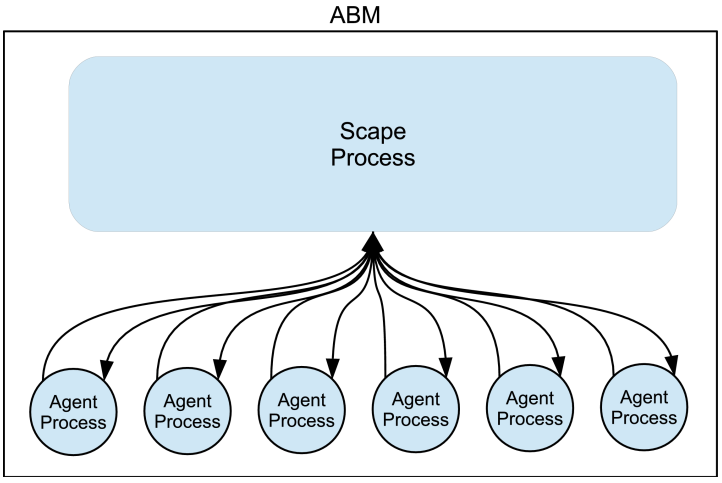

Fig-2. The Scape process and Agent processes interacting with it.

In ABMs like Neural Network systems, the agents which are in this case neurons, communicate with each other directly, and do not need a scape. And even the grid in the game of life can be created through a grid of interconnected agents (cells) following specific rules. In other ABMs, such as a model of city traffic flow, the agents move within an environment, the city. In such models, each agent thinks and processes data, and the environment/scape provides an avatar for the agent in the environment (pedestrian or a car), thus the agents communicate with each other and the environment. In either type of models though, Erlang's process/messaging approach provides a direct and simple implementation.

4.0 Examples:

In this section we build 2 multi-agent based systems to demonstrate how easy this can be when done using Erlang. The first will be the Schelling's tipping model and the second a simple Neural Network system. These two models are as opposite from each other as possible. The first one is composed of multiple agents and requires the evaluation of each agent's state and choice only while others are static, thus though it is a multi-agent system, it is somewhat sequential with agents operating and communicating with the environment rather than directly with each other. Neural Networks on the other hand are the opposite, each agent (a neuron in this case), has an address of other neurons it is connected *to* and *from*, and communicates with them directly. In this manner a neural-network type of system is completely concurrent with multiple neurons can be processing data at the same time.

4.1 Schelling's Tipping Model

Schelling's tipping model [11] is quite simple, and demonstrates macro-behavior emerging from micro-choices. In this model there are two separate species of agents, for example swimmers and surfers, 20 of each, existing on an 8x8 grid. This model was created to demonstrate how even fairly tolerant individuals might lead to segregated neighborhoods.

First the swimmers and surfers are randomly positioned on the grid. Each agent, though fairly tolerant, does have a preference. In the model in question an agent decides whether to stay at his current location or move to another location on the grid. The decision is made based on who his direct neighbors are. For a surfer, if a third of his neighbors are also surfers, the surfer stays put, if not, the surfer moves to a new random open location on the grid. The swimmers are less tolerant, in their case, they stay put if at least half are also swimmers, otherwise they move to a new location on the grid.

Thus the model is composed of the following agents: Swimmers, Surfers, and the Grid. While an agent is making a decision on whether to move or stay, other agents must not be moving around, otherwise the agent in question would not be able to calculate which new location to move to. Thus the model, though multi-agent, is somewhat sequential in nature. To create this model, we simply need to model a grid agent, a surfer agent, and a swimmer agent. Then we define the communication protocol between the agents and the environment (grid). Afterwards, we can spawn a single grid process to act as the environment, and as many surfer and swimmer agents as we want. The entire system takes under 120 lines of code (not counting comments), as is shown in Listing-1.

We can now run the model with default setup by executing schelling:start(). The result will be the final state of the model after 50 rounds, as described in [12]. The result are two numbers, the number of un-occupied sectors controlled by swimmers, and those controlled by surfers. The calculation whether an un-occupied sector is controlled by either of the specie of agents is done by checking whether there are more surfers or swimmers that are its direct neighbors.

Note that this model is highly extendible, can be put on pause and terminated by the user, allows the user to change the duration of the model runtime, the homogeneity of the agents, and the threshold parameters defining whether the agent stays or moves. All of this within roughly 100 lines.

Listing-1: The Schelling Tipping Point Model.

```erlang
-module(schelling).
-compile(export_all).

%start/0 is a wrapper for start/6, in which we define the number of swimmers and surfers to be created, and the grid size. We also define here what the ratio of the neighbors must be of the same type for each agent for it to stay put. The function spawns the environment process.
start()->start(20,0.33,20,0.5,8,8).
start(TotSurfers,SurferRatio,TotSwimmers,SwimmerRatio,XRange,YRange)->
	Scape_PId=spawn(?MODULE,init,[TotSurfers,SurferRatio,TotSwimmers,SwimmerRatio,XRange,YRange]),
	register(schelling,Scape_PId).

%init/6 spawns surfer and swimmer agents, each with its own process, PId, and a random position on the grid. Afterwards, the init function executes grid/5 to enter the environment loop.
init(TotSurfers,SurferRatio,TotSwimmers,SwimmerRatio,X,Y)->
	random:seed(now()),
	Surfer_PIds=spawn_agents(surfer,TotSurfers,X,Y,SurferRatio,[]),
	Swimmer_PIds=spawn_agents(swimmer,TotSwimmers,X,Y,SwimmerRatio,[]),
	Agent_PIds = Surfer_PIds++Swimmer_PIds,
	grid(Agent_PIds,Agent_PIds,X,Y,1).

%spawn_agents/6 function first finds a new empty location on the grid using the find_empty_loc/2, and then spawns an agent process of the particular type (either swimmer or surfer), initializing it with its location and ratio, and then adds that agent's PId to the accumulator. Once the spawn_agents/6 has spawned the required number of agents, it returns the list of accumulated PIds.
	spawn_agents(_Type,0,_XRange,_YRange,_Ratio,Acc)->
		Acc;
	spawn_agents(Type,Index,XRange,YRange,Ratio,Acc)->
		{X,Y}=find_empty_loc(XRange,YRange),
		Agent_PId = spawn(?MODULE,Type,[self(),{X,Y},Ratio]),
		put({X,Y},{Agent_PId,Type}),
		spawn_agents(Type,Index-1,XRange,YRange,Ratio,[Agent_PId|Acc]).

%find_empty_loc/2 checks the process' dictionary if the randomly generated coordinate already has an agent in it, if it does not, the function returns the coordinate, and if it does, the function checks a new random coordinate.
	find_empty_loc(XRange,YRange)->
		Random_X = random:uniform(XRange),
		Random_Y = random:uniform(YRange),
		case get({Random_X,Random_Y}) of
			undefined -> {Random_X,Random_Y};
			_ -> find_empty_loc(XRange,YRange)
		end.

%grid/5 is the environment/scape process, it understands 5 types of messages, 2 from agents, and 4 from the researcher which can put the environment on pause, continue, and terminate it. When the environment is terminated, it sends termination messages to the agents, terminating all agents and then terminating itself. When the scape process receives the message of the form: {Agent_PId,Loc,get_LocalState}, the scape extracts 8 positions directly neighboring the Location (Loc) that the agent sent it, and returns the list back to it. Each location on the gird either contains nothing in it and thus the atom undefined, or an agent, specified by the tuple {Agent_PId,Type}, where the Type is either the atom swimmer or surfer. When the scape process receives the message: {Agent_PId,Type,Loc,Choice} from the agent, it simply uses the atom stored in Choice as a function name, executing it. This can either be stay or move. In this manner the scape process accepts a message from each agent, then calculates for every empty coordinate whether it belongs to the surfers or the swimmers (depending whether there are more surfers or swimmers directly neighboring it). Finally, the scape process loops back, accepting a new set of messages from the agents.
grid(_,MPIds,_,_,50)->
	[APId ! terminate || APId <-MPIds],
	io:format("Terminating scape:~p~n",[self()]);
grid([Agent_PId|PIds],MPIds,XRange,YRange,RoundIndex)->
	receive
		{Agent_PId,{Target_X,Target_Y},get_LocalState} ->
			Local_State = [get({X,Y})|| X <- [Target_X-1,Target_X,Target_X+1],Y<-[Target_Y-1,Target_Y,Target_Y+1]],
			Agent_PId ! {self(),Local_State},
			grid([Agent_PId|PIds],MPIds,XRange,YRange,RoundIndex);
		{Agent_PId,Type,Loc,Choice}->
			U_Loc = ?MODULE:Choice(Agent_PId,Type,Loc,XRange,YRange),
			Agent_PId ! {self(),updated_loc,U_Loc},
			grid(PIds,MPIds,XRange,YRange,RoundIndex);
		pause ->
			receive
				continue ->
					grid([Agent_PId|PIds],MPIds,XRange,YRange,RoundIndex);
				terminate ->
					[APId ! terminate || APId <-MPIds],
					io:format("Terminating scape:~p~n",[self()])
			end;
		terminate ->
```

```
59                          [APId ! terminate || APId <-MPIds],
60                          io:format("Terminating Scape: ~p~n",[self()])
61          end;
62  grid([],MPIds,XRange,YRange,RoundIndex)->
63          {A,B}=gather_stats(1,1,XRange+1,YRange,0,0),
64          io:format("Round Index: ~p A:~p B:~p~n",[RoundIndex,A,B]),
65          grid(MPIds,MPIds,XRange,YRange,RoundIndex+1).
66
67  %stay/5 returns the original location of the agent, since the agent will not be moving.
68          stay(_Agent_PId,_Type,Loc,_XRange,_YRange)->
69                  Loc.
70  %move/5 first finds a new empty coordinate on the grid, and then moves the agent from its current coordinate to
    the new one, by erasing the presence of the agent from its original coordinate, and putting its presence on the
    new one. Finally, it returns the new coordinate of the agent to the function caller.
71          move(Agent_PId,Type,{Target_X,Target_Y},XRange,YRange)->
72                  {X,Y}=find_empty_loc(XRange,YRange),
73                  erase({Target_X,Target_Y}),
74                  put({X,Y},{Agent_PId,Type}),
75                  {X,Y}.
76
77  %gather_stats/6 calculates for each empty coordinate on the grid whether it belongs to surfers or swimmers,
    depending on whether there are more swimmers or surfers directly neighboring it.
78          gather_stats(XRange,YRange,XRange,YRange,AccA,AccB)->
79                  {AccA,AccB};
80          gather_stats(XRange,Y,XRange,YRange,AccA,AccB)->
81                  gather_stats(1,Y+1,XRange,YRange,AccA,AccB);
82          gather_stats(X,Y,XRange,YRange,AccA,AccB)->
83                  case get({X,Y}) of
84                          undefined ->
85                                  Local_State = [get({TX,TY})|| TX <- [X-1,X,X+1],TY<-[Y-1,Y,Y+1]],
86                                  TotSwimmers = lists:sum([1 || {_APId,swimmer} <- Local_State--[{self(),swimmer}
    ]]),
87                                  TotSurfers = lists:sum([1 || {_APId,surfer} <- Local_State--[{self(),surfer}]]),
88                                  if
89                                          (TotSurfers > TotSwimmers) -> gather_stats(X+1,Y,XRange,YRange,AccA
    +1,AccB);
90                                          (TotSurfers < TotSwimmers) -> gather_stats(X+1,Y,XRange,YRange,AccA,AccB
    +1);
91                                          true -> gather_stats(X+1,Y,XRange,YRange,AccA,AccB)
92                                  end;
93                          _ ->
94                                  gather_stats(X+1,Y,XRange,YRange,AccA,AccB)
95          end.
96
97  %swimmer/3 defines the swimmer agent process, it can send the environment/scape agent the request for its local
    state. Based on this local state and the Ratio it then decides on whether to stay or move, and sends its
    decision to the scape, awaiting its new grid coordinate.
98  swimmer(Scape_PId,Loc,Ratio)->
99          Scape_PId ! {self(),Loc,get_LocalState},
100         receive
101                 {Scape_PId,Local_State}->
102                         TotSwimmers = lists:sum([1 || {_APId,swimmer} <- Local_State--[{self(),swimmer}]]),
103                         Choice = case TotSwimmers >= (8*Ratio) of
104                                 true -> stay;
105                                 false -> move
106                         end,
107                         Scape_PId ! {self(),swimmer,Loc,Choice},
108                         receive {Scape_PId,updated_loc,U_Loc} -> ok end,
109                         swimmer(Scape_PId,U_Loc,Ratio);
110                 terminate -> ok
111         end.
112
113 %surfer/3 works the same way as the swimmer agent does, but can be set up to work completely different, and be
    based on a completely different set of rules.
114 surfer(Scape_PId,Loc,Ratio)->
115         Scape_PId ! {self(),Loc,get_LocalState},
116         receive
117                 {Scape_PId,Local_State}->
118                         TotSurfers = lists:sum([1 || {_APId,surfer} <- Local_State--[{self(),surfer}]]),
119                         Choice = case TotSurfers >= (8*Ratio) of
120                                 true -> stay;
121                                 false -> move
122                         end,
123                         Scape_PId ! {self(),surfer,Loc,Choice},
124                         receive {Scape_PId,updated_loc,U_Loc} -> ok end,
125                         surfer(Scape_PId,U_Loc,Ratio);
126                 terminate -> ok
127         end.
```

4.2 Neural-Network

A Neural Network (NN) is a another type of multi-agent system. In such a system each neuron is an agent, and can communicate directly with other neurons it is receiving signals from and sending signals to, as shown in Fig-3. In Listing-2 we create a fully concurrent neural network composed of a single sensor, 2 neurons, and a single actuator, all in under 70 lines of code (not counting comments). In this feedforward neural network, a sensor gets a signal, forwards it to 2 neurons, which concurrently process the signal and forward their outputs to the actuator, which then uses these signals to perform some action, which in this case is simply printing the signals to the shell. There can be hundreds to millions of neurons working at the same time in a larger implementation.

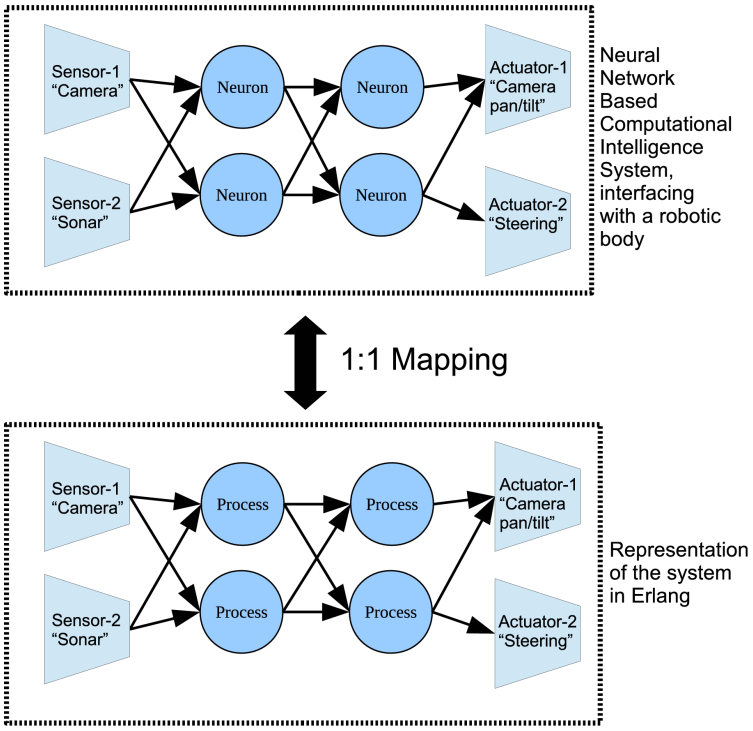

Fig-3. A 1-to-1 mapping between Erlang and a Neural Network based system.

For example, one such large evolving neural network system [13] has already been successfully implemented using Erlang. Thanks to Erlang's features, the neural network is fully concurrent, highly fault tolerant, capable of code hot-swapping, and has proven to be a flexible neuroevolution research platform. Furthermore, the neural networks generated by this system have also been used to control artificial organisms existing in 2d environment in ALife experiments. All of it was done using the simple methods and architectures discussed in this and the previous example, where the neurons and the environment were all modeled using concurrent processes.

Due to how closely Erlang maps to these types of models, the programs are, though fully concurrent, very short and very easy to understand. If visualization is needed, as in such models as Sugar-Scape, Erlang provides a graphical system library. An example module using it for ABM visualization is available on GitHub [14], which further contains the two examples we covered here.

Listing-2: A simple feedforward neural network model.

```
-module(simplest_nn).
-compile(export_all).

%create/0 first generates 3 weights, with the 3rd weight being the Bias. The Neuron is spawned first, and is then sent the PIds of the Sensor and Actuator that it's connected with. Then the Cortex element is registered and provided with the PIds of all the elements in the NN system.
create() ->
        Weights = [random:uniform()-0.5,random:uniform()-0.5,random:uniform()-0.5],
        N1_PId = spawn(?MODULE,neuron,[Weights,undefined,undefined]),
        N2_PId = spawn(?MODULE,neuron,[Weights,undefined,undefined]),
        N_PIds = [N1_PId,N2_PId],
        S_PId = spawn(?MODULE,sensor,[N_PIds]),
        A_PId = spawn(?MODULE,actuator,[N_PIds,N_PIds]),
        N1_PId ! {init,S_PId,A_PId},
        N2_PId ! {init,S_PId,A_PId},
        register(cortex,spawn(?MODULE,cortex,[S_PId,[N1_PId,N2_PId],A_PId])).

%neuron/3 expects to first receive the {init,New_SPId,New_APId} message, which sets its sensor and actuator pids. The neuron then expects to receive a vector of length 2 as input, and as soon as the input arrives, the neuron processes the signal and passes the output vector to the outgoing APId.
neuron(Weights,S_PId,A_PId) ->
        receive
                {S_PId,forward, Input} ->
                        io:format("****Thinking****~n Input:~p~n with Weights:~p~n",[Input,Weights]),
                        Dot_Product = dot(Input,Weights,0),
                        Output = [math:tanh(Dot_Product)],
                        A_PId ! {self(),forward,Output},
                        neuron(Weights,S_PId,A_PId);
                {init,New_SPId,New_APId} ->
                        neuron(Weights,New_SPId,New_APId);
                terminate ->
                        ok
        end.

%dot/3 takes a dot product of two vectors, it can operate on a weight vector with and without a bias. When there is no bias in the weight list, both the Input vector and the Weight vector are of the same length. When Bias is present, then when the Input list empties out, the Weights list still has 1 value remaining, its Bias.
        dot([I|Input],[W|Weights],Acc) ->
                dot(Input,Weights,I*W+Acc);
        dot([],[],Acc)->
                Acc;
        dot([],[Bias],Acc)->
                Acc + Bias.

%sensor/1 waits to be triggered by the Cortex element, and then produces a random vector of length 2, which it passes to the connected neuron. In a proper system the sensory signal would not be a random vector but instead would be produced by a function associated with the sensor, a function that for example reads and vector-encodes a signal coming from a GPS attached to a robot.
sensor(N_PIds) ->
        receive
                sync ->
                        Sensory_Signal = [random:uniform(),random:uniform()],
                        io:format("****Sensing****:~n Signal from the environment ~p~n",[Sensory_Signal]),
                        [N_PId ! {self(),forward,Sensory_Signal} || N_PId <- N_PIds],
                        sensor(N_PIds);
                terminate ->
                        ok
        end.

%actuator/3 function waits for a control signals coming from Neurons. As soon as the signal arrives, the actuator executes its function, pts/1, which prints the value to the screen.
actuator([N_PId|N_PIds],MNPIds) ->
        receive
                {N_PId,forward,Control_Signal}->
                        pts(Control_Signal),
                        actuator(N_PIds,MNPIds);
                terminate ->
                        ok
        end;
actuator([],MNPIds)->
        actuator(MNPIds,MNPIds).

        pts(Control_Signal)->
                io:format("****Acting****:~n Using:~p to act on environment.~n",[Control_Signal]).

%cortex/3 function triggers the sensor to action when commanded by the user. This process also has all the PIds of the elements in the NN system, so that it can terminate the whole system when requested. Because the cortex element was registered when the function create/0 was executed, you can send it the message to ask the sensor to
```

```
    produce data by executing in the shell: cortex ! sense_think_act. This can be done multiple times, after which to
    terminate the whole system you can execute: cortex ! terminate.
67  cortex(Sensor_PId,Neuron_PIds,Actuator_PId)->
68          receive
69                  sense_think_act ->
70                          Sensor_PId ! sync,
71                          cortex(Sensor_PId,Neuron_PIds,Actuator_PId);
72                  terminate ->
73                          Sensor_PId ! terminate,
74                          [N_PId ! terminate || N_PId <- Neuron_PIds],
75                          Actuator_PId ! terminate,
76                          ok
77          end.
```

5.0 Conclusion:

A full coverage of Erlang and its features is beyond the scope of this short paper, with a number of excellent books already covering the subject very thoroughly [15,16]. From this short introduction and two examples though, we can see how Erlang's features are intriguingly appropriate with regards to agent based modeling. Such systems can be prototyped very rapidly, and because of the very close mapping between Erlang and this problem domain, the resulting software is more understandable. We have seen how Erlang's paradigm is based on concurrent processes and message passing, similar to how ABMs are based on agents and their communication with each other through messages. Thus Erlang is certainly worth investigating when it comes to constructing large and complex agent based models, after all, Erlang was created from the very start to support just such systems, composed of millions of concurrent processes.

6.0 References:

[1] Niazi, Muaz; Hussain, Amir (2011). "Agent-based Computing from Multi-agent Systems to Agent-Based Models: A Visual Survey" (PDF). Scientometrics (Springer) 89 (2): 479–499.

[2] Bonabeau, E. (2002). Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences of the United States of America*, *99*(Suppl 3), 7280-7287.

[3] Farrell, W. (1998) How Hits Happen (HarperCollins, New York).

[4] Bonabeau, E. (2000) in Application of Simulation to Social Sciences, eds. Ballot, G. & Weisbuch, G. (Hermes Sciences, Paris), pp. 451–461.

[5] Casti, J. (1997) Would-Be Worlds: How Simulation Is Changing the World of Science (Wiley, New York).

[6] Arthur, W. B., Holland, J. H., LeBaron, B., Palmer, R. G. & Tayler, P. (1997) in The Economy as a Complex Evolving System II, Santa Fe Institute Studies in the Sciences

[7] Still, K. G. (1993) Fire 84, 40–41.

[8] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. November 2000. Licentiate Thesis.

[9] Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors* (Doctoral dissertation, KTH).

[10] Erlang on Xen: http://erlangonxen.org/

[11] Schelling, Thomas. 1978. Micromotives and Macrobehavior. New York: Norton, pp. 137-155.

[12] Axelrod, R. (1997). Resources for agent-based modeling. *Nova Jersey: Princeton University Press*.

[13] Sher, G. I. (2013). *Handbook of neuroevolution through erlang*. Springer.

[14] https://github.com/CorticalComputer/ABM

[15] Armstrong, J. (2007). *Programming Erlang*. Pragmatic Bookshelf.

[16] Logan, M., Merritt, E., & Carlsson, R. (2010). *Erlang and OTP in Action*. Manning Publications Co..