Controlling Turtles through State Machines: An Application to Pedestrian Simulation

Ilias Sakellariou

Department of Applied Informatics, University of Macedonia, Egnatia 156, Thessaloniki, Greece
iliass@uom.gr

Abstract. Undoubtedly, agent based modelling and simulation (ABMS) has been recognised as a promising technique for studying complex phenomena. Due to the attention that it has attracted, a significant number of platforms have been proposed, the majority of which target reactive agents, i.e. agents with relatively simple behaviours. Thus, little has been done toward the introduction of richer agent oriented programming constructs that will enhance the platforms' modelling capabilities and could potentially lead to the implementation of more sophisticated models. This paper discusses TSTATES, a domain specific language, together with an execution layer that runs on top of a widely accepted agent simulation environment and presents its application to modelling pedestrian simulation in an underground station scenario.

Keywords: Agent Simulation Platforms: Agent Programming Languages: Crowd Simulation

1 Introduction

Agent based modelling and simulation has been widely adopted as a new technique for studying complex emergent phenomena in areas such as economics, biology, psychology, traffic and transportation, etc. [1]. This inevitably lead to the introduction of a plethora of agent modelling and simulation platforms (ABMS) [2] [3], that offer modelling environments of different complexity and characteristics, in terms of programming language, scalability, extensibility, ease of use, user support (documentation, tutorials, example models), etc.

The NetLogo multi agent modelling environment [4], has been regarded as one of the most successful and complete ABMS platforms [5, 6], offering an IDE with extensive visualization tools, and a simple domain specific agent programming language; this"one-stop" approach allows users to arrive to a simulation experiment with a relatively small effort; a fact that has a definitive advantage toward the adoption of the platform by the community. However, although NetLogo is excellent for "modelling social and emergent phenomena" consisting of a large number of reactive agents, it lacks the modelling facilities to accommodate more complex agent behaviours.

The problem originally has been addressed in [7] that presents an approach toward building higher level intention driven communicating NetLogo agents. That work offers a framework for message exchange and a simple mechanism for specifying persistent intentions and beliefs, in a PRS like style.

A different approach was adopted in the TSTATES (Turtle-States) domain specific language (DSL) [8]. The latter supports agent behaviour specification through state machines, an approach similar to those that have been mainly used in robotics [9] and RoboCup simulation teams [10]. TSTATES provides a small and simple domain specific language (DSL) on top of NetLogo and an execution layer, and thus allows users to encode and execute more sophisticated agent models. The present paper builds on the work described in [11], by discussing in more detail TSTATES, through a complete specification of the DSL, an evaluation of its performance and its application to metro station simulation scenario.

The rest of the paper is organised as follows: Section 2 acts as a brief introduction to NetLogo and its terminology, necessary for placing the rest of the paper in the right context. Section 3 provides a description of TSTATES by presenting its primitives, its grammar and an evaluation of its performance through a motivating example. In section 4 a complete example of TSTATES to a multi agent model concerning crowd simulation is described. Section 5 presents the work reported in the literature that is closely related to the current approach. Finally, section 6 concludes the paper and discusses future extensions.

2 The NetLogo ABMS Environment

NetLogo is "a cross-platform multi-agent programmable modelling environment" [4] aiming to multi-agent systems' simulation with a large number of agents. There are four entities participating in a NetLogo simulation:

- The *Observer*, that is responsible for simulation initialisation and control.
- *Patches*, i.e. components of a user defined static grid that is a 2D or 3D world, which is inhabited by turtles. Patches are useful in describing environment behaviour, since they are capable of interacting with other agents and executing code.
- *Turtles* that are agents that "live" and interact in the above world. They are organised in *breeds*, i.e. user defined groups sharing some characteristics, such as shape, but most importantly breed specific user defined variables that hold the agents' state.
- *Links* agents that "connect" two turtles representing usually a spatial/logical relation between them.

Patches, turtles and links carry their own *internal state*, stored in a set of system and user-defined variables local to each agent. By the introduction of an adequate set of patch variables, a sufficient description of complex environments can be achieved. The definition of turtle specific variables allows the former to carry their own state and facilitates encoding of complex behaviour.

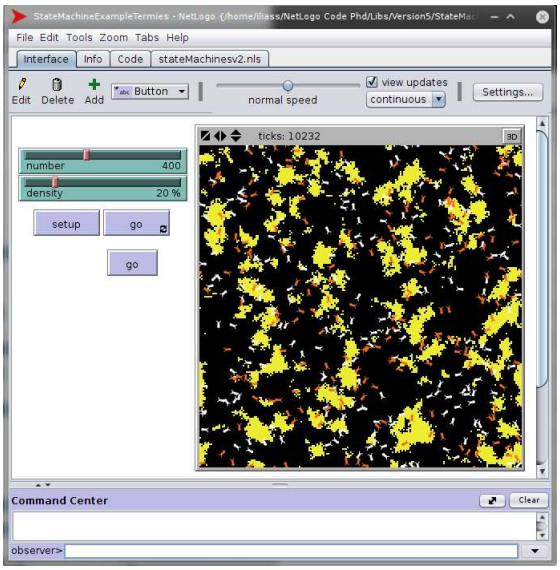


Fig. 1. Termites Model. Yellow patches represent wood-chips, black patches free space, white "bugs" termites not carrying anything, while orange "bugs" termites carrying a wood-chip. Example is taken from the NetLogo models library

Agent behaviour can be specified by the domain specific NetLogo programming language, that has a rather functional flavour and supports functions (called *reporters*) and *procedures*. The language includes a large set of primitives for turtles motion, environment inspection, classic program control (ex. branching), etc. NetLogo v5 introduced *tasks*, a significant extension to the language, since through the former users can store code in variables to be executed at a later stage. Reasoning about time is supported through *ticks*, that are controlled by the observer, each tick corresponding to a discrete execution step. Finally, the programming environment offers simple GUI creation facilities that minimizes the time required to develop a simulation. A simple example of a model that can be built in it is shown in figure 1. The model shown is the "termites" model that will serve in the following as the running example in order to present the TSTATES DSL.

3 State Machines for Specifying Behaviour

The requirements for developing TSTATES were a) to provide the modeller with the means to easily encode complex agent behaviours, and b) to seamlessly integrate with the programming environment, maintaining the advantages of the latter. Thus, its was decided that the TSTATES DSL consists of *a set of primitives* to specify turtle behaviour as a state machine, and *an execution layer* for directly executing these specifications in NetLogo. The domain specific language

is tightly coupled with the platform's own language, thus allowing the developer to use all the language primitives of the latter in an transparent way.

TSTATES adopts a rather common form of state machines, in which transitions from a state, are labelled with a condition/action pair, i.e

$(State, Condition_1) \Rightarrow (Action_1, Next\_State_1)$.

The library allows directly encoding such transitions in NetLogo in the following form:

```
state <StateName>
  # when <NetLogo Condition 1> do <NetLogo Action 1>  goto <StateName 1>
  ...
  # when <NetLogo Condition i> do <NetLogo Action i>  goto <StateName i>
end-state
```

In the above, a state definition is included in the keywords `state` and `end-state` and `<StateName>` is a string acting as the unique name of the state. Each transition in a state begins with the symbol `#`. The keywords `when`, `do` and `goto` specify a transition condition, an action and the target state respectively.

A string representation of any valid logical expression of NetLogo reporters preceded by the keyword `when` can act as a *condition*. Thus, model specific agent "sensors" or platform defined reporters (NetLogo has a large set of the latter) can be used to trigger transitions. Special library conditions include:

- `otherwise`, in the form of `otherwise do <Action> goto <State>`, that always evaluates to true.
- `for-n-ticks <n>`, which evaluates to true for *n ticks* after the state was last entered. This allows agents to perform an action for a certain amount of time upon entering a state.
- `after-n-ticks <n>`, which constantly evaluates to true *n ticks* after the last entry (activation) of the state. It is useful to encode timeouts related to a state activation.
- Finally, conditions `invoked-from <state>`, `previous-active-state <state>`, `on-failure <Machine>` and `on-success <Machine>` are special conditions related to machine invocation and will be discussed in section 3.1.

Similarly to conditions, *actions* are string representations of any valid NetLogo sequence of procedures preceded by the keyword `do`. The special library action ``` ``nothing'' ``` defines transitions that are not labelled with an action.

The keyword `goto` specifies the transition's target state, one that belongs to the same state machine. There is also another kind of target state transition, that of invoking a different state machine that is discussed in more detail in section 3.1. Two target pseudostates exist `success` and `failure` that both represent final states of the machine and have no transitions attached.

The *execution layer* evaluates transition conditions in a state in the order that they appear, firing the first transition in that list whose condition is satisfied (triggered), i.e. imposing a *transition ordering*. Prioritizing transitions based on their order allows behaviour encoding using less complex conditions, at the cost

of demanding special care from the user and allows conditions like `otherwise` to be semantically clear.

A *state machine* is a (NetLogo) list of state definitions, with the first state in this list being the initial state. TSTATES grammar is depicted in figure 2.

Having NetLogo primitives acting as conditions and actions leads to a tight integration with the underlying platform, allows a NetLogo user to easily define all the necessary components of the agents in the model under study and specify the behaviour of the agents using state machines. It also permits adaptation of existing NetLogo models easily. Finally, it should be noted that states (and machines as described later) can communicate information using the turtle's own variables, as for example is reported in [9] as well as through parameter passing of reporters and procedures used in transition definition.

3.1 Callable State Machines

TSTATES supports the concept of *callable* state machines, i.e. state machines that can be invoked by a transition from any state and terminate returning a boolean result. The concept is similar to nested functions, in the sense that when such a machine terminates, control returns to the state that invoked the machine. This feature aims at reducing the number of states required for encoding complex agents, through "code" re-usability. In effect, callable machines allow encoding of a form of agent behaviour templates, i.e. actions to be taken in order to cope with a specific situation that are applied in multiple cases.

A callable state machine, returns whether such a behaviour has succeeded through a boolean value that is signalled by a special transition to a pseudo-state. Thus, each such machine has to include at least a `success` or a `failure` pseudo-state to terminate its execution. Upon termination, the calling state can optionally activate transitions on the result returned by the invoked machine, by employing the special `on-success <MachineName>` and `on-failure <MachineName>` transition conditions. Before the invocation of the corresponding callable machine both these conditions evaluate to false. Machines are invoked through appropriate transition using the `activate-machine <MachineName>` and just as ordinary programming functions, nested invocations for machines can reach any level. The number of different machines that can be invoked from transitions belonging to a single state is unlimited.

To allow encoding of more flexible state machines, i.e. machines the behaviour of which might differentiate based on the "calling" state, two new conditions were introduced:

- `invoked-from <state>`, which evaluates to true if the state that invoked the current state is that stated in the parameter.
- `previous-active-state <state>`, which evaluates to true if the state `<state>` is active (in stack).

The complete TSTATES grammar is depicted in figure 2.

Machine = **‘state-def-of-’**MachineName **‘report (list’** State+ **‘)’**
State = StateDef+
StateDef = **‘state’** StateName Transition+ **‘end-state’**
Transition = **‘#’** Condition **‘do’** Action StateChange
Condition = **‘when’** ReporterExp | **‘otherwise’**
| **‘for-n-ticks’** N | **‘after-n-ticks’** N
| **‘invoked-from’** StateName
| **‘previous-active-state’** StateName
| **‘on-failure’** MachineName
| **‘on-success’** MachineName
Action = Procedures | **‘nothing’**
StateChange = **“goto’** StateName
| **‘activate-machine’** MachineName
| **‘success’** | **‘failure’**
N = ⟨INTEGER⟩
StateName = ⟨STRING⟩
MachineName = ⟨STRING⟩
Procedures = ⟨STRING⟩
ReporterExp = ⟨STRING⟩

Fig. 2. TSTATES DSL grammar. Please note that Procedures and ReporterExp are string representations of a set of NetLogo procedure and an expression of Netlogo reporters respectively.

3.2 Coding a Simple Behaviour: The Termites Model

To illustrate the use of TSTATES, a version of the “State Machines” NetLogo library model [4] is employed. The model is an alternative version of the “Termites” model, originally introduced to the platform to illustrate the use of the new concept of tasks, and concerns an example drawn from biology, i.e. simulation of termites gathering wood chips into piles. Termite behaviour is governed by simple rules: each termite wanders randomly until it finds a wood chip, then picks up a chip and carries it until it locates a clear space near another wood pile, where it “drops” the chip its carrying. Eventually, all chips initially scattered in the world are collected in large piles. The state machine model of termites is depicted in figure 3.

The corresponding TSTATES NetLogo code is shown below.

```
to-report state-def-of-turtles
 report (list
  state  "search-for-chip"
   # when "pile-found" do "pick-up" goto "find-new-pile"
   # otherwise do "move-randomly" goto "search-for-chip"
  end-state
  state "find-new-pile"
   # for-n-ticks 20 do "fd 1" goto "find-new-pile"
   # when "pile-found" do "nothing" goto "put-down-chip"
   # otherwise do "move-randomly" goto "find-new-pile"
```

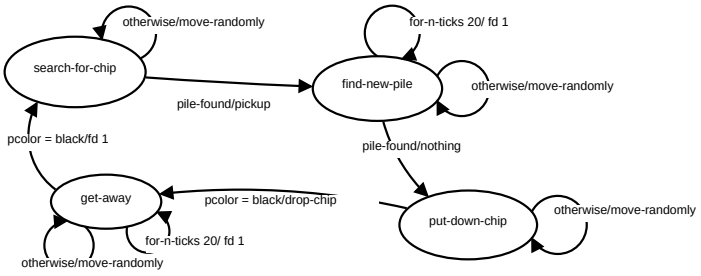


Fig. 3. The Termites State Machine Model. The transitions are labelled by a *condition / action* pair.

```
  end-state
  state "put-down-chip"
   # when "pcolor = black" do "drop-chip" goto "get-away"
   # otherwise do "move-randomly" goto "put-down-chip"
  end-state
  state "get-away"
   # for-n-ticks 20 do "fd 1" goto "get-away"
   # when "pcolor = black" do "fd 1" goto "search-for-chip"
   # otherwise do "move-randomly" goto "get-away"
  end-state )
end
```

The reader should notice the name of the NetLogo reporter that "stores" the state machine indicates the *breed* of agents whose behaviour is specified (i.e. `state-def-of-turtles` specifies the behaviour of the "turtles" breed). In the model, chips are represented as yellow patches, where free space as black. The conditions "pile-found", corresponds to a simple Netlogo reporter that returns true is the patch the turtle is located on is coloured yellow. Obviously, this could be easily achieved by simply including the `pcolor = yellow` as a condition, as in the case of finding a free space (`pcolor = black`). It was chosen to be included as a reporter in order to demonstrate some aspects of the library and make the model more readable. As seen from the above simple example, encoding state machines in the proposed library is a straightforward task. A comparison and a listing of the two code examples (TSTATES and the original NetLogo) can be found in [11].

3.3 Implementation

A major design choice was to implement TSTATES using the NetLogo programming language. This decision stems from the fact that such a choice allows easy inclusion in any Netlogo model, transparent integration with the underlying platform's language and also easy modification of the primitives offered. The implementation heavily depends on the notion of *tasks* that were introduced

in NetLogo version 5. Each such machine specification encoded by the user is compiled at run time to an executable form, that employs directly executable tasks by appropriate function invocations and stored in the corresponding data structures.

For each turtle that uses state machines, four stacks must be defined as turtle's own variables:

- The *active-states* stack that holds the set of states that have not yet terminated along with necessary information concerning the state, i.e. when the state was last entered, results of invoking other machines, etc.
- the *active-states-code* stack that holds the code for each state in the active-states stack,
- the *active-machines* stack that stores the state machine to which each state in the active states stack corresponds to, and finally,
- the *active-machine-names* stack that hold the names of the machines currently active. Obviously the top of each of the stacks is the active state, code and machine respectively that determine the behaviour of the agent.

The DSL includes two procedures. The procedure `initialise-state-machine` performs machine initialisation, i.e. it loads the initial state of the state machine that matches the breed of the turtle. After that, the procedure `execute-state-machine` is the only thing that needs to be "asked" by the turtle in order to execute its specified behaviour. The latter invokes an execution cycle that includes determining all transitions whose condition holds and selecting the first one in order to execute its action part. Although evaluation of all transition conditions seems unnecessary in this step, it was chosen so that transiton prioritisation can be easily implemented in future extension of TSTATES.

Loading a state involves popping the previous state from the active-states stack and pushing the new state. Similar operations occur for the state code in the active-states-code stack. Each normal transition, i.e. to a state that belongs to the current active state machine is loaded in a similar manner. Machine invocation is slightly different in the sense that the execution layer locates the initial state of the invoked machine and structures are simply updated by pushing the new state information.

It should be noted that the `execute-state-machine` procedure selects and executes only one transition at each cycle. This choice was made so that the NetLogo models could be developed more easily since (the *ask turtles*) NetLogo primitive imposes a sequential order on the execution of agents, waiting for one to finish before initiating the next. Additionally, such an approach allows the use of ticks in the simulation.

Obviously, managing such structures impose an overhead to the execution. In order to investigate the former, a set of experiments were performed involving the termites model presented above. The experiments involve running a full simulation, i.e one in which the world is updated on every tick, for a varying number of turtles and allow each experiment to run for a number of ticks. Time concerns the execution of the top-level procedure that invokes an experiment

cycle and was measured using the profiler of NetLogo. The results are shown in table 1, and as indicated when a large number of agents exist in the simulation the execution time is increases, to even a factor of 2 when the number of agents reaches 1000.

Table 1. Experiments with world updates. Columns TSTATES and NetLogo depict the execution time in ms, an AMD Athlon X2, Linux Machine.

Termites (turtles)	Ticks (Run)	TSTATES Time (ms)	NetLogo Time (ms)
300	100	3706.06	3522.7
300	500	17529.38	16968.68
300	1000	35069.38	33930.1
500	100	5274.99	3406.17
500	500	26449.79	16974.99
500	1000	52884.82	33923.64
1000	100	10314.49	5803.46
1000	500	52154.88	29004.4
1000	1000	101169.16	58325.72

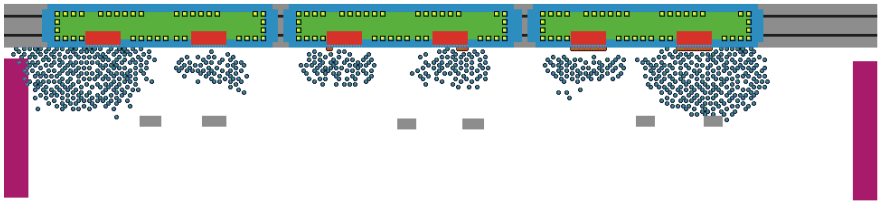
It is expected that when NetLogo runs "headless", i.e. without world updates, the overhead introduced would be more significant. Thus, one of the future directions in this research is to provide a more efficient compilation scheme, possibly through directly compiling state machine specifications to NetLogo code. Nevertheless, the aim of this work was to provide modelling facilities for complex behaviours, and in that respect we argue that the modelling complex agents in NetLogo can be greatly facilitated by TSTATES. The case study that follows further supports this claim.

4 Underground Station Simulation

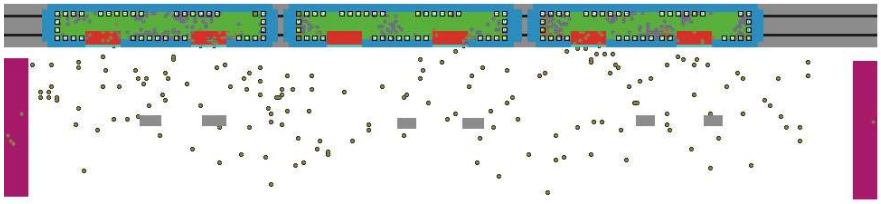
A model of pedestrian simulation in an underground station was developed in order to demonstrate the implementation of more complex behaviours. The example was drawn from [12], where authors use the Situated Cellular Agent model to simulate crowd behaviour while boarding and descending a metro wagon in an underground station. In this paper, as depicted in figure 4, we have tested the passenger model in an environment that has multiple wagons, instead of one as in [11]. Space is discrete, that is agents move on a grid formed by the underlying patches, although continuous space could also be supported.

The simulation concerns a complete passenger cycle, i.e. passenger boarding and descending. This was done, since we wanted to investigate how boarding passengers affect the behaviour of passengers descending the wagon, and in order to have a richer state machine to encode.

Passenger behaviour is specified by a state machine, as the latter is depicted in figure 5. This is almost the same machine that we used in [11], that with



(a) Passengers waiting for doors to open



(b) Heading for the exit

Fig. 4. The Underground Station Environment. Different areas are coloured coded in the simulation: the entrance is marked with colour magenta, the wagon area green, and the red coloured patches represent the door area.

a few minor modifications copes well with the current environment. Informally and rather briefly, each passenger:

- Upon entering the station walks towards a random platform point and then selects a wagon door to walk to.
- When close to the door and doors open, boards the wagon by selecting a door area to walk toward to. If there are any passengers descending the passenger steps back to facilitate their exit.
- When in the door area, selects a clear spot in the wagon to move to. Upon arriving at the spot, the passenger has completed boarding.
- If the passenger "sees" an empty seat, he/she tries to get seated.
- After a while starts to descend from the wagon. This involves selecting the nearest door area for un-boarding and walks towards that door.
- When at the door area, the passenger selects an exit, walks towards this new target and "leaves" the simulation.

There is a number of interesting points in the state diagram of figure 5. The first point to notice is that the passenger's walking behaviour is invoked by each state that requires movement: thus it is encoded as a separate state machine ("walk-toward"), reducing the total number of states. There are two things worth mentioning here. Firstly, the state machine is called with two parameters, proximity and time. The first concerns how close to the target should the agent be in order to consider the task successful and the second concerns

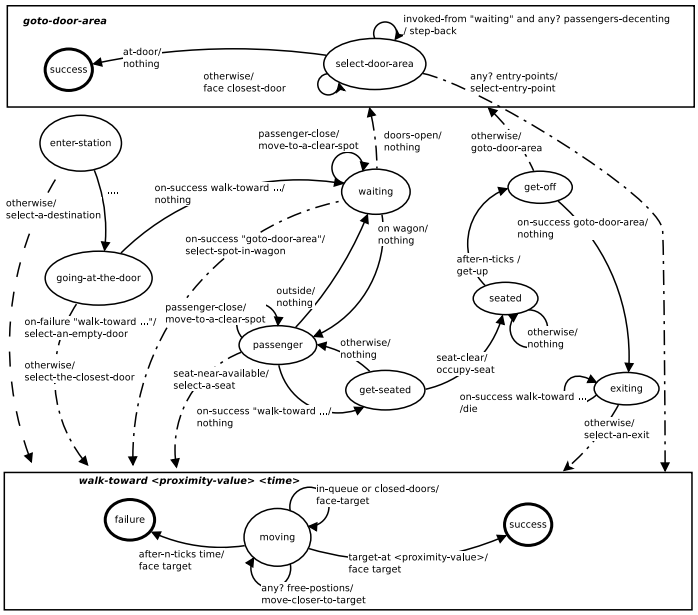


Fig. 5. State Diagram of the Passenger. Please not that dotted arrows indicate machine invocations, where normal arrows simple state transition.

how long the agent would try to achieve its goal of moving towards the target, before dropping its goal. Thus, the TSTATES allows encoding of parametrised agent plans and a form of intention persistence. Secondly, the target location is communicated between states through an agent (turtle) variable. The sole purpose of this choice was to show that the tight integration of TSTATES with the underlying platform; The same effect could have been easily done by having one more parameter in the state machine. Both the above show how TSATES allows for easy encoding of complex agent behaviour.

A second point to notice concerns the "goto-door-area" state machines. The latter encodes passenger behaviour when moving to the door area, an intermediate target during boarding and descending the wagon. The behaviour is differentiated in the two cases mentioned: if the passenger is boarding, then he must step back to allow other passengers to descend (a polite passenger); if not this behaviour does not occur. This differentiation is achieved by having a transition guarded by a condition that check which state invoked the "goto-door-area" machine, as shown in the code below (numbered (1)):

```
state "select-door-area"
 # when invoked-from "waiting" and any? passengers-descenting"
       do "step-back" goto "select-door-area" (1)
 # when "at-door" do "nothing" success
 # when "any? entry-points" do "select-entry-point"
       activate-machine "walk-toward near 15"
 # otherwise do "face closest-door" goto "select-door-area"
```

```
end-state
```

Finally, it should be noted that the "goto-door-area" invokes the "walk-towards" in order the passenger reaches its selected target. Thus, as shown from the example above, TSTATES can indeed meet most of the needs such complex agent simulations demand. Results of the simulation can be viewed in figures 4(a) and 4(b), corresponding to passengers boarding and descending from the wagon.

5 Related Work

The work described in this paper relates both to state machine specification of intelligent agents and programming languages for agent simulation platforms. Thus, in the following we report on the relevant literature on both these areas.

Many approaches reported in the literature adopt finite state machines to control agent behaviour. For example in [10] [13] authors describe a specification language, *XABSL* for defining hierarchies of state machines for the definition of complex agent behaviours in dynamic environments. According to the approach, *options*, i.e. state machines, are organised through successive invocations (one option state can invoke another option) in a hierarchy, an acyclic graph consisting of options, with the leaf nodes being basic behaviours (actions). Traversal of the tree based on external events, state transition conditions and past option activations, leads to a leaf node that is an action. It should be noted that *XABSL* was employed by the German RoboCup robot soccer team with significant success.

COLBERT [9] is an elegant C like language defining hierarchical concurrent state machines. *COLBERT* supports execution of activities (i.e. finite state automata) that run concurrently possibly invoking other activities and communicate through a global store or signals. Agent (robot) actions include robot actions and state changes, and all agent state information is recorded in the Saphira perceptual space.

eXAT [14], models *tasks* of the agent using state machines, that can be "activated" by the rule engine of the agent. eXAT tasks can be combined sequentially or concurrently, allowing re-usability of the defined state machines. Fork and join operators on concurrent state machine execution exist that allow composition of complex tasks.

TSTATES provides some of the above mentioned features and lacks others. State machine invocation is possible through the `activate-machine` primitive, but concurrent execution of state machines, as that is defined in *COLBERT* and *XABSL* is missing. Concurrent actions, although is clearly a desired property in a robotic system that operates in the real world, might not be that suitable for agent simulation platforms and especially for NetLogo. In the latter, fairness among agents in the simulation is provided by ensuring that at each cycle one action is selected and executed in the environment. However, having multiple concurrent active states is a future direction of the TSTATES library, possibly incorporating some sort of priority annotation on the actions that would allow in the end to have a single action as the outcome of the state machine.

There is a large number of agent simulation platforms that have been developed in the past decade [2] [3]. Out of these, state machine like behaviour encoding is offered in two of them, Sesam [15] and RePast [16]. In Sesam a visual approach to modelling agents is adopted, where users develop activities that are organised in using UML-like *activity diagrams*. RePast offers agent behavioural modelling through flowcharts (along with JAVA, Groovy and ReLogo) that allow the user to visually organise tasks. While both approaches are similar to the TSTATES, the latter offers callable states and machine invocation history that, to our opinion, facilitate the development of sophisticated models, as presented above. Furthermore its tight integration with the NetLogo platform and given the latter's simplicity in building simulations, allows users to build models more easily. However, since among some user categories, visual development of state machines is a rather attractive feature, we consider the inclusion of such a facility in the future.

6 Conclusions

This work reports on extensions regarding the TSTATES DSL and on the use of the latter in a more complex example. The approach presents a number of benefits: complex behaviour definition using state transitions and transparent integration with NetLogo platform's language primitives is transparent, thus loosing not expressivity w.r.t. the agent models that can be encoded. We intend to extend the current approach in a number of ways:

- Support the execution of concurrent active states as discussed in section 5 and possibly fork and join composition operators on machine invocation. However, this is a issue that requires further research and outside the scope of this paper.
- Provide facilities for debugging/authoring state machines in NetLogo, as for example visual tools to encode state machines, like in [15] and [16]. The latter we expect to increase the adoption of TSTATES and the platform itself.
- Investigate alternative compilation techniques for more efficient integration with NetLogo.

We are also considering other agent programming language paradigms as well, such as AgentSpeak(L) [17] and their integration to NetLogo. This direction will allow to extend the environment to even more complex simulation scenarios.

Finally, it should be noted that both the library TSTATES and the examples presented in this paper, can be found at `http://users.uom.gr/~ iliass/`.

References

1. Davidsson, P., Holmgren, J., Kyhlbck, H., Mengistu, D., Persson, M.: Applications of agent based simulation. In Antunes, L., Takadama, K., eds.: Multi-Agent-Based Simulation VII. Volume 4442 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 15–27 10.1007/978-3-540-76539-4_2.

2. Nikolai, C., Madey, G.: Tools of the trade: A survey of various agent based modeling platforms. Journal of Artificial Societies and Social Simulation **12**(2) (2009) 2
3. Allan, R.J.: Survey of agent based modelling and simulation tools. Technical Report DL-TR-2010-007, DL Technical Reports (2010)
4. Wilensky, U.: Netlogo. Center for Connected Learning and Computer-based Modelling. Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo. (1999)
5. Railsback, S.F., Lytinen, S.L., Jackson, S.K.: Agent-based simulation platforms: Review and development recommendations. SIMULATION **82**(9) (2006) 609–623
6. Lytinen, S.L., Railsback, S.F.: The evolution of agent-based simulation platforms: A review of netlogo 5.0 and relogo. In: Proceedings of the Fourth International Symposium on Agent-Based Modeling and Simulation, Vienna, Austria (2012)
7. Sakellariou, I., Kefalas, P., Stamatopoulou, I.: Enhancing Netlogo to Simulate BDI Communicating Agents. In Darzentas, J., Vouros, G., Vosinakis, S., Arnellos, A., eds.: Artificial Intelligence: Theories, Models and Applications. Volume 5138 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 263–275
8. Sakellariou, I.: Turtles as state machines - agent programming in netlogo using state machines. In Filipe, J., Fred, A.L.N., eds.: ICAART 2012 - Proceedings of the 4th International Conference on Agents and Artificial Intelligence, Volume 2 - Agents, Vilamoura, Algarve, Portugal, 6-8 February, 2012, SciTePress (2012) 375–378
9. Konolige, K.: COLBERT: A language for reactive control in sapphira. In Brewka, G., Habel, C., Nebel, B., eds.: KI:Advances in Artificial Intelligence. Volume 1303 of Lecture Notes in Computer Science., Springer (1997) 31–52
10. Loetzsch, M., Risler, M., Jungel, M.: Xabsl - a pragmatic approach to behavior engineering. In: Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on. (oct. 2006) 5124 –5129
11. Sakellariou, I.: Agent based modelling and simulation using state machines. In Pina, N., Kacprzyk, J., Obaidat, M.S., eds.: SIMULTECH, SciTePress (2012) 270–279
12. Bandini, S., Federici, M.L., Vizzari, G.: Situated cellular agents approach to crowd modeling and simulation. Cybernetics and Systems **38**(7) (2007) 729–753
13. Risler, M., von Stryk, O.: Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. In: AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems, Estoril, Portugal (2008)
14. Stefano, A., Santoro, C.: Supporting agent development in erlang through the exat platform. In Unland, R., Calisti, M., Klusch, M., Walliser, M., Brantschen, S., Calisti, M., Hempfling, T., eds.: Software Agent-Based Applications, Platforms and Development Kits. Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhuser Basel (2005) 47–71
15. Klügl, F., Herrler, R., Fehler, M.: Sesam: implementation of agent-based simulation using visual programming. In: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. AAMAS '06, New York, NY, USA, ACM (2006) 1439–1440
16. North, M.J., Howe, T.R., Collier, N.T., Vos, J.R.: A declarative model assembly infrastructure for verification and validation. In: Advancing Social Simulation: The First World Congress, Springer, Heidelberg, FRG (2007)
17. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., Sandewall, E., eds.: Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91), Morgan Kaufmann publishers Inc. (1991) 473–484