Agent Based Modelling and Simulation using State Machines

Ilias Sakellariou
Department of Applied Informatics, University of Macedonia, Egnatia 156 Str, GR-54006, Thessaloniki, Greece
iliass@uom.gr

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

Abstract: Although agent based modelling has drawn significant attention in the recent years, with a significant number of simulation platforms proposed, the latter target usually relatively simple reactive agents. Thus, little has been done toward enhancing the modelling capabilities of platforms with richer agent oriented programming constructs that could potentially lead to the implementation of more sophisticated models. This paper presents an extension to the TSTATES library that aims to support the implementation of state based intelligent agents and a more complex example drawn from the pedestrian simulation domain that demonstrates the potential of the library in programming complex agent systems.

1 INTRODUCTION

Agent based modelling and simulation has been extensively used as a technique to study complex emergent social and biological phenomena and has set a new trend in many areas, such as economics, biology, psychology, traffic and transportation etc. (Davidsson et al., 2007). This growing interest led to the introduction of a large number of agent modelling and simulation tools (Nikolai and Madey, 2009) (Allan, 2010) that offer programming environments of different complexity and different characteristics in terms of programming language employed in modelling, completeness w.r.t. documentation, tutorials, ease of use, scalability, extensibility, etc.

NetLogo (Wilensky, 1999) is regarded as one of the most complete and successful agent simulation platforms (Railsback et al., 2006; Lytinen and Railsback, 2012), in the sense that it provides a simple domain specific language for agent programming, an IDE, and the necessary experiment visualisation tools for quick development of simulation user interface. Although, excellent for "modelling social and emergent phenomena", i.e. agent based simulations that consist of a large number of reactive agents, it lacks the facilities to model easily more complex goal oriented agent behaviours. This problem originally was been addressed in (Sakellariou et al., 2008), that presents an approach towards building higher level communicating NetLogo agents, with goals and plans and 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) (Sakellariou, 2012), that supports the definition of agent behaviour participating in the simulation through state machines, an approach similar to those that have been mainly used in robotics (Konolige, 1997) and RoboCup simulation teams (Loetzsch et al., 2006). TSTATES provides a small and simple domain specific language (DSL) on top of the NetLogo programming language and an execution layer that allows users to encode and execute more sophisticated agent models.

In the original work of TSTATES a number of extensions were described. This paper deals with some of these extensions and describes a more complex simulation example using TSTATES, that aims to show how the latter extends the NetLogo platform applicability to a number of domains, consisting of more sophisticated agents.

The rest of the paper is organised as follows: Section 2 introduces the basic components of NetLogo, and introduces the platforms terminology, necessary for placing the rest of the paper in the right context. Section 3 provides a description of the TSTATES library and the extensions implemented by presenting its primitives through a motivating example. In section 4 a more 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 PLATFORM

The NetLogo platform is "a cross-platform multi-agent programmable modelling environment" (Wilensky, 1999) aiming to multi-agent systems' simulation with a large number of agents. In any NetLogo agent simulation, four entities participate:

- The *Observer*, that is responsible for simulation initialisation and control.
- *Patches*, i.e. components of a user defined static grid (world) 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 world formed by patches. Turtles are organised in *breeds*, that are 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.

Both patches, turtles and links carry their own *internal state*, stored in a set of system and user-defined variables local to each agent. By introducing an adequate number 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 the encoding of complex behaviour.

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 variety 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 the possibility to create code stored in a variable 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. An 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.

NetLogo can be an ideal platform for initial prototyping and simulation of multi-agent systems, provided these systems have some spatial dimension and consist of relatively simple agents that react to environment "events". On the other hand, modelling more sophisticated agents that exhibit more complex behaviour, is a challenging task in the platform since there are no language primitives aiming towards this direction.

3 STATE MACHINES FOR SPECIFYING BEHAVIOUR

The TSTATES DSL offers a set of primitives to specify turtle behaviour as a state machine, together with 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.

In the TSTATES library, a rather common form of state machines is adopted, in which transitions from a state, are labelled with a condition/action pair and have the following form:

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

...

$(State, Condition_i) \Rightarrow (Action_i, Next\_State_i)$

The library supports encoding in NetLogo transitions like the above in the following form:

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

In the above, the keywords `state` and `end-state` signal the beginning and the end of a state definition and `<StateName>` is a string acting as the unique name of the state. Each transition in a state begins with the symbol #. The `when`, `do` and `goto` are the keywords that 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. A number of special library conditions are offered:

- `otherwise`, in the form of `otherwise do <Action> goto <State>`, that always evaluates to true.

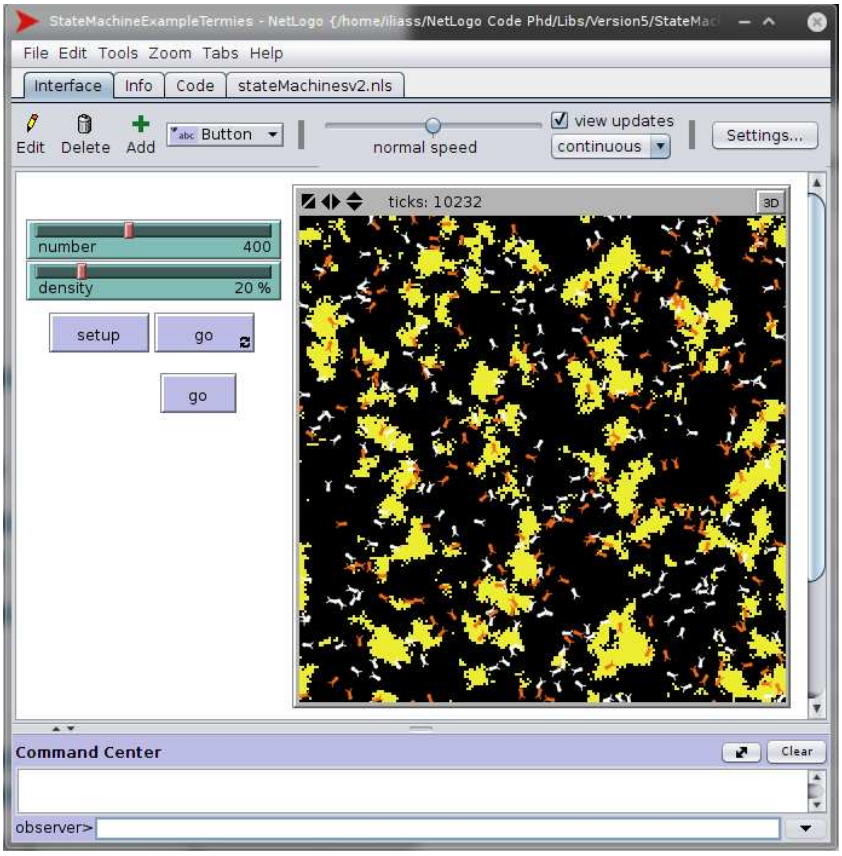


Figure 1: Termites Model. Yellow patches represent wood-chips, black patches free space, white "bugs" termites not carrying anything, while orange "bugs" termites carrying a wood-chip. Example is taken from the NetLogo models library

- `for-n-ticks <n>`, which evaluates to true for *n ticks* after the state was last entered. This was introduced since it was discovered that it is often required from an agent 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 3.2.

Similarly to conditions, *actions* are string representations of any valid NetLogo sequence of procedures preceded by the keyword `do`. The special library action `''nothing''` can be employed to define transitions that are not labelled with an action. This 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 (Konolige, 1997) as well as through parameter passing of reporters and procedures used in transition definition.

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, using the `activate-machine` keyword that is discussed in more detail in section 3.2. 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 part and allows conditions like `otherwise` to be semantically clear.

Finally, a *state machine* is a (NetLogo) list of state definitions, with the first state in this list being the initial state.

3.1 The Termites State Machine Model

To illustrate the use of TSTATES, a version of the "State Machines" NetLogo library model (Wilensky, 1999) is employed in the following. 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 chip, 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 2.

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"
  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 simplyu 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 of the two code examples (TSTATES and the original NetLogo) can be found in the Appendix.

3.2 Invoking State Machines

Code re-usability is an important issue in any programming language. Especially in the case of state machines, it can significantly decrease the number of states needed for encoding the agent behaviour. The library 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. Each such callable state machine, has to include at least a *success* or a `failure` pseudostate to terminate its execution. Upon termination of execution, 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. It should be noted that before the invocation of the callable machine both these conditions evaluate to false. Machines are invoked 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. The latter presents one of the extensions introduced in this paper, compared to the work described in (Sakellariou, 2012).

Additionally, two new conditions were introduced to the library, increasing its expressivity:

- Condition `invoked-from <state>`, which evaluates to true if the state that invoked the current

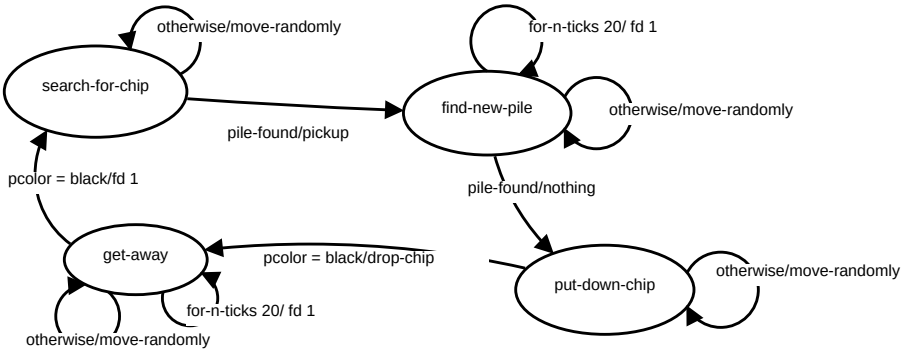


Figure 2: The Termites State Machine Model. The transitions are labelled by a *condition / action* pair.

state is that stated in the parameter.

- Condition `previous-active-state <state>`, which evaluates to true if the state `<state>` is active (in stack).

Thus depending on the "calling" state the behaviour of the invoked machine can be differentiated. This allows the introduction of more flexible state machines, enabling encoding of behaviours which differ slightly depending on the calling state.

Callable states can significantly reduce the number of states of a machine and furthermore can provide the means to define templates, i.e. predetermined agent behaviours that can be used with possibly minor modifications in various contexts. One such characteristic case, can be the Contract Net Protocol, where the roles of the contractor and the manager could be encoded as two independent state machines.

3.3 Implementation

The TSTATES library was implemented in the NetLogo programming language, to allow easy inclusion in any Netlogo model, modification of the library primitives offered and transparent integration with the underlying platform's language. Its implementation depends on the notion of *tasks* and each machine specification is transformed to an executable form employing directly executable tasks by appropriate function invocations and stored in the corresponding data structures.

The execution layer allows for one state transition at a time. This is to ensure fairness in the simulation, in which all agents get to perform one action at simulation step. Additionally, such an approach allows the use of ticks in the simulation.

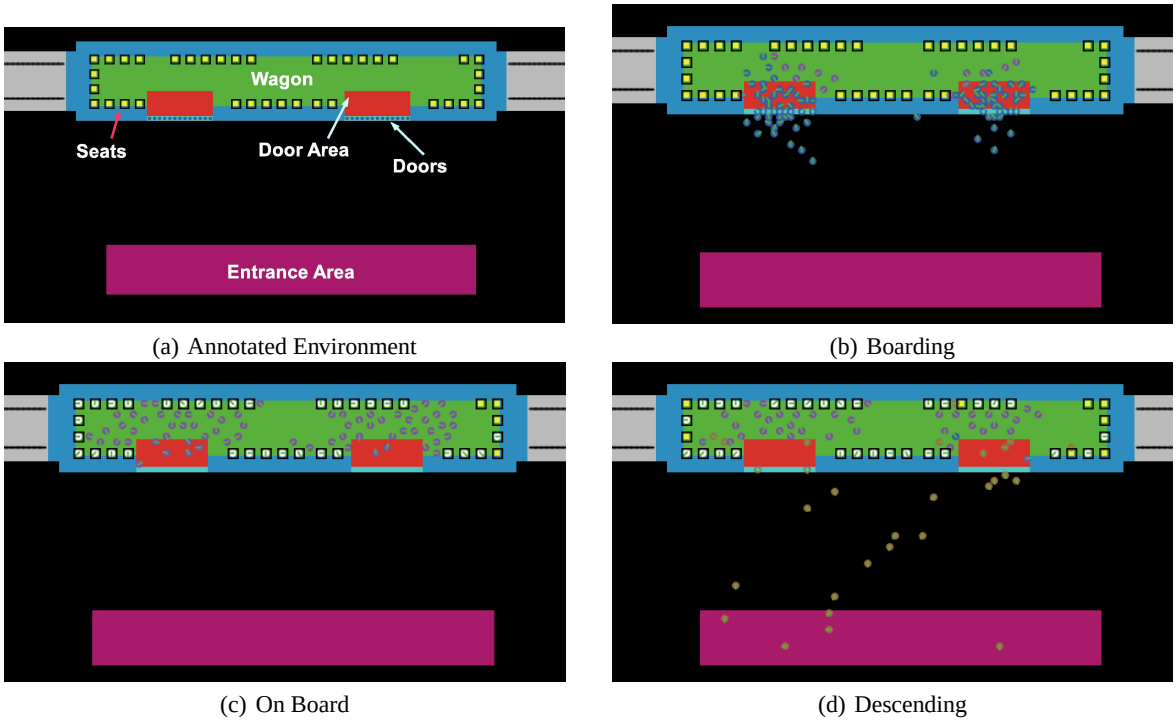
4 Underground Station Simulation

In order to demonstrate the implementation of more complex agent behaviour, a model of an underground station was developed. The example was drawn from (Bandini et al., 2007), where authors use the Situated Cellular Agent model to simulate crowd behaviour while boarding and descending a metro wagon in an underground station. The simulation environment is depicted in figure 3(a), in which different areas are annotated. 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, in the sense that the simulation models not only the boarding but also the descending of passengers in the wagon. 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 4. Informally and rather briefly, each passenger:

- Upon entering the station, selects its closest door and walks towards that target.
- When close to the door and doors open, boards the wagon by selecting a door area (coloured red) to walk towards. 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 top get seated.
- After a while (determined differently for each passenger), begins to descend from the wagon. This involves selecting the nearest door area for unboarding and walks towards that door.



(a) Annotated Environment (b) Boarding

(c) On Board (d) Descending

Figure 3: The Underground Station Environment. Different areas are coloured coded: the entrance is marked with colour magenta, the wagon area green, and the red coloured patches represent the door area.

- 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 4. The first point to notice walking behaviour of the passenger is encoded as a separate state machine ("walk-toward") that is invoked by each state requiring the passenger to move to a specified location. 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 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 3(b), 3(c) and 3(d), corresponding to passengers boarding, on-board and descending from the wagon.

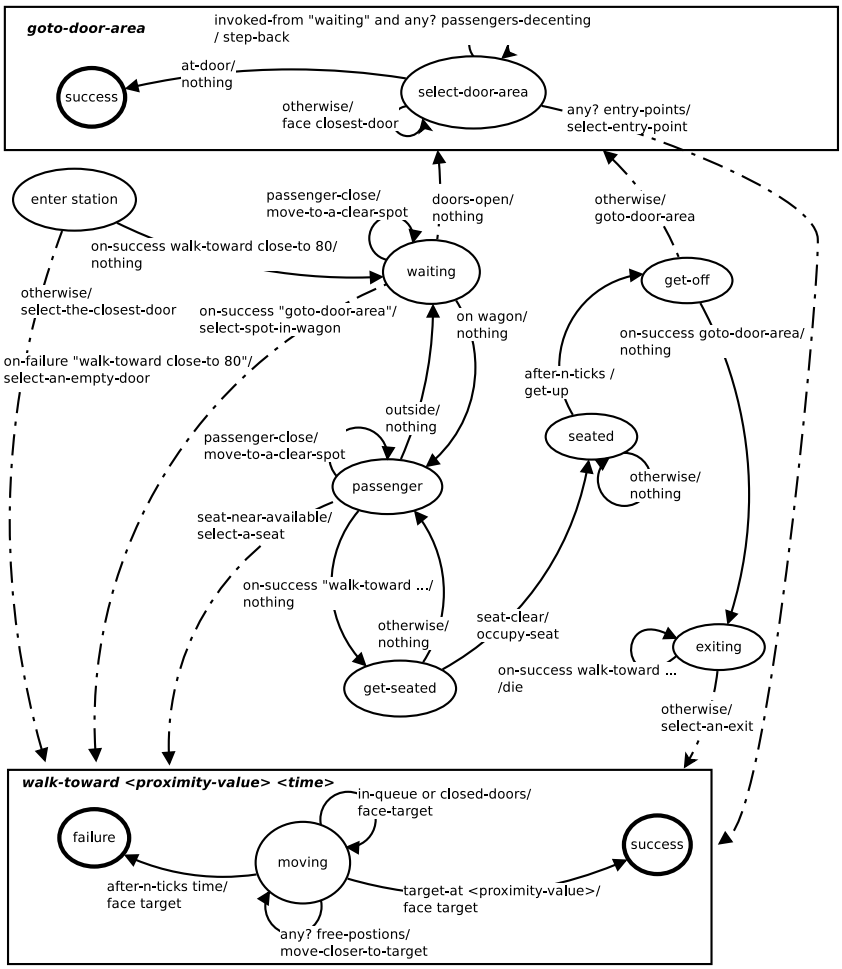


Figure 4: State Diagram of the Passenger. Please not that dotted arrows indicate machine invocations, where normal arrows simple state transition.

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 research areas.

Many approaches reported in the literature adopt finite state machines to control agent behaviour. For example in (Loetzsch et al., 2006) (Risler and von Stryk, 2008) 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 (Konolige, 1997) 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 (Stefano and Santoro, 2005), 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 (Nikolai and Madey, 2009) (Allan, 2010). Out of these, state machine like behaviour encoding is offered in two of them, Sesam (Klügl et al., 2006) and RePast (North et al., 2007). 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 AND DISCUSSION

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: determining complex behaviour using state transitions is simple and 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 and authoring state machines in NetLogo, as for example visual tools to encode state machines, like in (Klügl et al., 2006) and (North et al., 2007). The latter we expect to increase the adoption of TSTATES and the platform itself.

We are also considering other agent programming language paradigms as well, such as AgentS-

peak(L) (Rao and Georgeff, 1991). However, these approaches usually require the definition of an event queue from which an event is selected and the corresponding rule fires. In the case of NetLogo, such an approach presents a number of problems: since the agent is allowed to monitor a large number of global variables, own variables and the environment around it through a large set of language primitives (reporters and procedures), it might be the case that the event queue list grows to a size that makes its manipulation inefficient. Thus, state machine oriented approaches allow to focus on a smaller set of changes in the world and consequently lead to a more efficient execution.

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

Allan, R. J. (2010). Survey of agent based modelling and simulation tools. Technical Report DL-TR-2010-007, DL Technical Reports.

Bandini, S., Federici, M. L., and Vizzari, G. (2007). Situated cellular agents approach to crowd modeling and simulation. *Cybernetics and Systems*, 38(7):729–753.

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

Klügl, F., Herrler, R., and Fehler, M. (2006). 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, pages 1439–1440, New York, NY, USA. ACM.

Konolige, K. (1997). COLBERT: A language for reactive control in sapphira. In Brewka, G., Habel, C., and Nebel, B., editors, *KI:Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*, pages 31–52. Springer.

Loetzsch, M., Risler, M., and Jungel, M. (2006). Xabsl - a pragmatic approach to behavior engineering. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 5124 –5129.

Lytinen, S. L. and Railsback, S. F. (2012). 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.

Nikolai, C. and Madey, G. (2009). Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2.

North, M. J., Howe, T. R., Collier, N. T., and Vos, J. R. (2007). A declarative model assembly infrastructure for verification and validation. In *Advancing Social Simulation: The First World Congress*. Springer, Heidelberg, FRG.

Railsback, S. F., Lytinen, S. L., and Jackson, S. K. (2006). Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623.

Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., and Sandewall, E., editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.

Risler, M. and von Stryk, O. (2008). 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.

Sakellariou, I. (2012). Turtles as state machines - agent programming in netlogo using state machines. In Filipe, J. and Fred, A. L. N., editors, *ICAART 2012 - Proceedings of the 4th International Conference on Agents and Artificial Intelligence, Volume 2 - Agents, Vilamoura, Algarve, Portugal, 6-8 February, 2012*, pages 375–378. SciTePress.

Sakellariou, I., Kefalas, P., and Stamatopoulou, I. (2008). Enhancing Netlogo to Simulate BDI Communicating Agents. In Darzentas, J., Vouros, G., Vosinakis, S., and Arnellos, A., editors, *Artificial Intelligence: Theories, Models and Applications*, volume 5138 of *Lecture Notes in Computer Science*, pages 263–275. Springer Berlin / Heidelberg.

Stefano, A. and Santoro, C. (2005). Supporting agent development in erlang through the exat platform. In Unland, R., Calisti, M., Klusch, M., Walliser, M., Brantschen, S., Calisti, M., and Hempfling, T., editors, *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 47–71. Birkhuser Basel.

Wilensky, U. (1999). Netlogo. Center for Connected Learning and Computer-based Modelling. Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo.

APPENDIX

The following table provides a side by side comparison of the original code of the "termites" model to that of TSTATES for illustration purposes. Note that TSTATES allows state machine encoding to be more readable and thus easier to debug and develop.

Table 1: Code for the Termites model, between TSTATES and original NetLogo code.

TStates Code

```
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"
 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

to move-randomly
   rt random 50
   lt random 50 fd 1
end

to pick-up
  set pcolor black set color orange
end

to drop-chip
  set pcolor yellow set color white
end

to-report pile-found
 report pcolor = yellow
end

to-report empty-space
 report pcolor = black
end
```

NetLogo Original Code

```
to go
  ask turtles
    [ ifelse steps > 0
        [ set steps steps - 1]
        [ run next-task
          wiggle ]
      fd 1 ]
  tick
end

to wiggle ; turtle procedure
  rt random 50
  lt random 50
end

;; "picks up chip" by turning orange
to search-for-chip
  if pcolor = yellow
    [ set pcolor black
      set color orange
      set steps 20
      set next-task task find-new-pile
    ]
end

;;  look for yellow patches
to find-new-pile
  if pcolor = yellow
   [set next-task task put-down-chip]
end

;; finds empty spot & drops chip
to put-down-chip
  if pcolor = black
   [ set pcolor yellow
     set color white
     set steps 20
     set next-task task get-away
   ]
end
;; get out of yellow pile
to get-away
  if pcolor = black
   [set next-task task search-for-chip]
end
```