TURTLES AS STATE MACHINES
Agent Programming in NetLogo using State Machines

Ilias Sakellariou
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
iliass@uom.gr

Keywords: Agent Simulation Platforms: Agent Programming Languages

Abstract: Agent based modelling has received significant attention in the recent years mainly due its wide adoption by scientists in a number of fields. Although agent simulation platforms have proven to be quite mature and expressive for modelling simple agents, little has been done regarding enhancing these platforms by higher level agent oriented programming facilities. This work aims at this direction, i.e. an add-on library to a well known simulation platform aiming at the specification of complex high level agents, using state machines.

1 INTRODUCTION

Agent based modelling and simulation has been extensively used as a technique to study complex emergent social and biological phenomena 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 number of agent modelling and simulation tools (Nikolai and Madey, 2009).

NetLogo (Wilensky, 1999) is such a platform and is regarded as one of the most complete and successful agent simulation platforms (Railsback et al., 2006). Although, excellent for developing reactive agent simulations, it lacks the facilities to easily model more complex agents. There is only one work reported (Sakellariou et al., 2008) towards the latter, that offered a framework for message exchange and a simple mechanism for specifying persistent intentions and beliefs (PRS like).

This paper presents the TSTATES (Turtle-States) library that follows a different approach, similar to those that have been used mainly in robotics and RoboCup teams to control the behaviour of agents (turtles), i.e. we describe the latter through a form of state machines. TSTATES offers a domain specific language to specify agent state machines and an execution layer for running such specifications in NetLogo. Being able to encode more sophisticated NetLogo agent models could extended the platform's applicability to a number of domains.

2 NETLOGO STATE MACHINES

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". Three entities participate in a NetLogo simulation: the *observer*, that initiates/controls the simulation; *patches*, i.e. components of a grid (world), and *turtles* that are agents that "live" and interact in the world, which can be organised in *breeds*. Both patches and turtles carry their own state, stored in a set of system or user defined agent variables, that allow modelling of complex environments and behaviours. Encoding agent behaviour is accomplished in a domain specific programming language and reasoning about time is supported through *ticks*, each corresponding to a discrete execution step. Finally, *tasks* are a significant extension introduced in NetLogo 5, that offer execution of code stored in a variable.

The TSTATES library aims at extending NetLogo by allowing to encode agents controlled by state machines, in which transitions are labelled with by actions and have the following form:

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

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

```
state <StateName>
# when <Cond 1> do <Act 1> goto <Next_State 1>
...
# when <Cond i> do <Act i> goto <Next_State i>
end-state
```

In the above, a state definition is included between keywords `state` and `end-state` and the keywords `#`, `when`, `do` and `goto` specify a transition definition, a condition, an action and the target state respectively.

A string representation of any valid NetLogo boolean reporter (function) can act as a *condition*. Thus the user can develop model specific agent "sensors" or use platform supported reporters to trigger transitions. Special library conditions include:

- `otherwise` that always evaluates to true,
- `for-n-ticks <n>` which evaluates to true for *n ticks* after the state was last entered,
- `after-n-ticks <n>` which constantly evaluates to true *n ticks* after the last activation of the state,
- `on-failure` and `on-success` are two special conditions discussed below.

Similarly to conditions, *actions* are string representations of any valid NetLogo procedure. This integration with the underlying platform, allows definition of agent "actuators" in the latter and organisation of agent behaviour using state machines. The special library action `nothing` defines transitions that are not labelled with an action. Finally, states can share information using turtle's own variables, in a similar manner as in (Konolige, 1997).

The keyword `goto` specifies the transition's target state, one that belongs to the same state machine. Another kind of transition supported is that of invoking a different state machine, using the `activate-machine` keyword. The library supports the concept of *callable state machines*, i.e. machines that can be invoked by a transition 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 callable state machine, has to include at least a `success` or a `failure` pseudostate to terminate its execution. Upon termination, the calling state can optionally activate transitions on the result returned, by employing the special `on-success` and `on-failure` transitions conditions. Currently the implementation allows only one such machine to be invoked by a single state and nested invocations for machines can reach any level. Callable machines can significantly reduce the number of states required and provide the means to define agent template behaviours.

Within a state, the order of transitions is important, since the execution layer evaluates conditions in the order that they appear, firing the first transition whose condition is satisfied. *Transition ordering* allows behaviour encoding using less complex conditions. A machine is a NetLogo list of state definitions, with the first state in the list being the initial state.

2.1 Communicating MARS Explorer

To illustrate the use of the library, a modified version of the Mars Explorer agent (Steels, 1990) is employed. In that a number of robotic vehicles (collectors) have the task of exploring a distant planet and collecting samples that have to bring back to a base. The terrain has obstacles that must be avoided and collectors themselves present obstacles to one another. We assume that collectors are aware of the exact location of the base. The agent model as a state machine is shown in figure 1.

Avoiding an obstacle is a somewhat complex behaviour according to which agents must try for some time (10 ticks) to get far away from an obstacle. If the agent succeeds in reaching a clear area (with no `obstacle-around`) the behaviour (machine) succeeds, otherwise the agent tries to "drive" clear once more. This behaviour has to be invoked by all states that involve moving around in the world and thus it is encoded as a callable state machine (`get-away-from-obstacles`), as shown in figure 1. The corresponding TSTATES code is shown below:

```
to-report state-def-of-get-away-from-obstacle
report (list
state "finding-a-clear-space"
# when "detect-obstacle" do "change-heading"
  goto "finding-a-clear-space"
# for-n-ticks 10 do "fd 0.2"
  goto "finding-a-clear-space"
# otherwise do "nothing"
  goto "check-space-cleared"
end-state
state "check-space-cleared"
# when "obstacle-around" do "nothing"
  goto "finding-a-clear-space"
# otherwise do "nothing" success
end-state) end
```

Having a large number of collectors operating in the environment, leads to a situation where all agents compete in their attempt to reach the base to unload samples. As a consequence a large number of agents gather around the base, preventing each other from achieving their goal. In order to prevent such a phenomenon, a simple coordination mechanism was imposed using symbolic message exchange by employing the library described in (Sakellariou et al., 2008). The coordination scheme is rather simple: the base acts as a central authority and agents have to wait for permission before moving to the base. When a collector drops off its samples, it notifies the base so that the latter can grant permission to another waiting agent.

A fragment of the corresponding NetLogo code is shown below. The name of the NetLogo reporter indicates the *breed* of agents whose behaviour is specified. In the example, checking the same conditions twice at state "move-to-base" is avoided by exploiting the transition ordering and the `otherwise` special

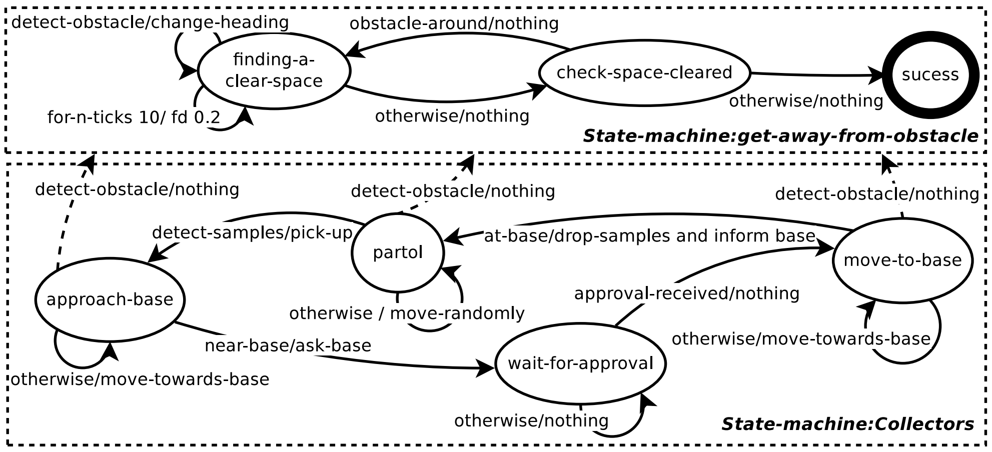

Figure 1: The cooperative Mars collector agent. The state machine above servers as a good example of reusability, since the get-away-from-obstacle behaviour is reused in three other states.

condition. As seen from the code, encoding state machines in TSTATES is a straightforward task.

```
to-report state-def-of-collectors
report (list
 state "patrol"
 # when "detect-obstacle" do "nothing"
   activate-machine "get-away-from-obstacle"
   ...
 end-state
 state "approach-base"
   ...
 state "wait-approval"
   ...
 state "move-to-base"
 # when "detect-obstacle" do "nothing"
   activate-machine "get-away-from-obstacle"
 # when "at-base" do "drop-samples inform-base"
   goto "patrol"
 # otherwise do "move-towards-base"
   goto "move-to-base"
 end-state) end
```

2.2 Implementation

We choose to implement the TSTATES library in the NetLogo programming language, mainly due to the fact that such a choice allows its effortless inclusion in any NetLogo model, and easy modification of the library primitives offered. The implementation heavily depends on the notion of *tasks*: each machine specification is encoded by the user as a NetLogo reporter that is transformed to an executable form (task) by appropriate function invocations and stored in the corresponding data structures.

For each turtle that uses state machines, three stacks are defined as turtle's own variables: (a) the *active-states* stack that holds the set of states that have not yet terminated along with necessary information concerning each state; the *active-states-code* stack that holds the corresponding code for each state, and; (c) the *active-machines* stack that stores the state machine to which each state in the active states stack corresponds to. Obviously the top of each of the stacks determines the behaviour of the agent.

The library procedure `execute-state-machine` is the only thing that needs to be called by the turtle to execute its specified behaviour. The first time the procedure is invoked, it loads the initial state of the machine that matches the breed of the turtle. The procedure executes only one action at each cycle, a necessity imposed by the fact that the *ask turtles* NetLogo primitive imposes a sequential order on the execution of agents, waiting for one to finish before initiating the next. Additionally, ticks would not work otherwise.

3 RELATED WORK

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 concerning complex agent behaviours in dynamic environments. According to the approach, *options*, i.e. state machines, are organised through successive invocations 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. *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. It 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 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 agent *tasks* 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 library provides some of the above mentioned features and lacks others. State machine invocation is possible through the `activate-machine` primitive, but concurrent execution as that is defined in *COLBERT* and *XABSL* is missing. Concurrent actions, although clearly a desired property in a real-world robotic system, might not be 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 executed. 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 at the end to have a single action as the outcome of the process. Finally, although similar agent behaviours could be encoded in the RePast (North et al., 2007) agent simulation platform, such a task would require more effort by a scientist not familiar with JAVA programming to create an experiment.

4 CONCLUSIONS

The TSTATES library is a first attempt to a more sophisticated control of NetLogo turtles, that presents a number of benefits: simple behaviour specification and seamless integration with the NetLogo language primitives, resulting in no expressivity loss 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 and possibly fork and join composition operators on machine invocation,
- investigate how state machine behaviours can be combined with concepts of current BDI approaches to programming agents, (as in eXAT),
- provide debugging facilities in NetLogo.

As a final note, simulation platforms can be an good testbed for the initial evaluation of new agent programming languages, since they allow their assessment in gradually more sophisticated "complete" agent environments.

REFERENCES

Davidsson, P., Holmgren, J., Kyhlbck, H., Mengistu, D., and Persson, M. (2007). Applications of agent based simulation. In *Multi-Agent-Based Simulation VII*, volume 4442 of *Lecture Notes in Computer Science*, pages 15–27. Springer Berlin / Heidelberg.

Konolige, K. (1997). Colbert: A language for reactive control in sapphira. In *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.

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.

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

Steels, L. (1990). Cooperation between distributed agents through self-organisation. In *Towards a New Frontier of Applications, Proceedings of the IEEE International Workshop on Intelligent Robots and Systems (IROS'90)*, pages 8–14.

Stefano, A. and Santoro, C. (2005). Supporting agent development in Erlang through the eXAT platform. In *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.