Enhancing NetLogo to Simulate BDI Communicating Agents

Ilias Sakellariou[1], Petros Kefalas[2], and Ioanna Stamatopoulou[2]

[1] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
iliass@uom.gr
[2] Department of Computer Science, CITY College, Thessaloniki, Greece
kefalas@city.academic.gr, istamatopoulou@seerc.org

Abstract. The implementation process of complex agent and multi-agent systems (AMAS) can benefit significantly from a simulation platform that would allow rapid prototyping and testing of initial design ideas and choices. Such a platform, should ideally have a small learning curve, easy implementation and visualisation of the AMAS under development, while preserving agent oriented programming characteristics that would allow to easily port the design choices to a fully-fledged agent development environment. However, these requirements make such a simulation platform an ideal learning tool as well. We argue that NetLogo meets most of the requirements that suit our criteria. In addition, we describe two extra NetLogo libraries, one for BDI-like agents and one for ACL-like communication that allow effortless development of goal-oriented agents, that communicate using FIPA-ACL messages. We present one simulation scenario that employs these libraries to provide an implementation in which agents cooperate under a Contract Net protocol.

Keywords: Multi-Agent Systems, Simulation Platforms.

1 Introduction

Development of Agents and Multi-Agent Systems (AMAS) is a challenging and complex task. Due to the complexity that AMAS exhibit, simulation of AMAS models becomes an important step that can facilitate understanding of how the intended system will perform when it will be actually implemented, since it allows rapid prototyping and testing of initial design ideas and choices. It is crucial that simulation output should be meaningful enough for the developers to draw conclusions and drive the actual implementation. For instance, in multi-agent systems with spatial reasoning and behaviour, a visual output which displays agents moving in a two or three dimensional space is necessary.

On the other hand, future developers and researchers must be educated in AMAS theory and trained in practice, which presents an equally challenging task. Firstly, the topic is too broad to fit within specific time constraints. Especially at the University level, a course on AMAS can hardly fit itself among a plethora of other mainstream/popular topics, despite the fact that AMAS is listed in the ACM/IEEE Computing Curricula [1] as part of Intelligent Systems

J. Darzentas et al. (Eds.): SETN 2008, LNAI 5138, pp. 263–275, 2008.

area. Secondly, due to the wide foundations and applicability of AMAS, it is only natural that there is a lot of diversity with respect to learning outcomes and content, teaching and assessment, theory and practice etc. But definitely, educators' common aim is to introduce AMAS as a useful new software paradigm, and for this appropriate educational tools are necessary.

Such educational tools for AMAS should ideally have a small learning curve, easy implementation and visualisation of the system under development, while preserving agent oriented programming characteristics that would allow to easily port the design choices to a fully-fledged agent development environment.

The requirements for choosing a simulation environment that can also act as an educational tool (or vice-versa) are complemented by the following criteria:

- have a simple environment that presents the minimum installation problems,
- provide easy visualization for a better view of the agent behaviour,
- easy to learn and use language thus keeping a small learning curve,
- clearly demonstrate the difficulties in AMAS programming,
- support the basic agent architectures (reactive, BDI, hybrid)
- provide means for communication, message exchange and interaction.

A number of languages and environments are available as options to use in practice for training future AMAS developers and researchers. Apparently, deciding which one to choose is not easy, since all the above criteria are not fully met by existing development or simulation environments. For the work presented in this paper, we chose NetLogo [2]. NetLogo has a number of important features as an educational and simulation tool for AMAS, but lacks support for goal-oriented and communicating agents. Our aim is to present how these issues are dealt with, by enhancing NetLogo with libraries that support BDI architecture and FIPA-ACL message exchange. The libraries have been developed for educational purposes but can also be used for more advanced AMAS simulation.

The structure of the paper is as follows: In section 2, we discuss in more detail the requirements and the choices available for using a simulation tool in education. NetLogo is briefly presented in Section 3 and the extensions we suggest and implemented are listed in Section 4. In the following section 5, we present a case study of forest fires simulation, which demonstrates the applicability of our proposal. Finally, we conclude with discussion and future work.

2 Simulation and Education in AMAS

It is commonly accepted that simulation tools and environments are not only useful for AMAS research and development but also useful for training future AMAS researchers and developers. We will generally refer to the latter as learners; they might be students, researchers or professionals who undergo further training. In order to understand the theoretical concepts and complexity of AMAS, some kind of practical program development is needed. There are several choices educators may follow, depending on the emphasis and focus they give to certain aspects of AMAS, e.g. architectures, communication and interaction

protocols, applications etc. Various tools and environments for AMAS have been reported to assist the educational process, like RoboCup, NetLogo, TAC, FIPA-OS, JADE, JadeX, Jason, Protégé etc. [3,4,5,6,7,8]. All aim to improve active learning in the context of AMAS, some by engaging learners in writing code (e.g. Java), others by allowing development of peripheral to agents structures (e.g. Ontologies).

Many success stories in integrating such environments are reported, but admittedly, most of the times there are concerns about the tool's complexity, the time spent by some educators to build such a tool and the time spent by the learners to reach the required level of skills in order to produce something useful to their eyes. Given the time restrictions for training, which normally last a few weeks, it is rather difficult for them to develop something simple but also meaningful, easy to implement but also challenging, artificial but also realistic enough, at some level of abstraction but also practical.

It would be preferable if learners see a more realistic view of their work in a simulation. If this is the process of a competition like game (e.g. Trading Agent Competition) or a visualisation of the agents' environment (e.g. RoboCup simulation), the understanding and satisfaction seems to be increasing. One could also argue that a simple robotic platform (e.g. Lego Mindstorms, RoboSapiens, i-SOFT etc.) [9,10] could also serve the purpose, since in the average mind perception a robot (from a science fiction perspective) matches with that of an agent. Although, such an approach is feasible and sometimes desirable in more engineering-oriented courses, it still suffers from complexity issues due to the fact that learners need to take into account non-symbolic percepts and additional hardware devices and protocols.

In principle, learners should receive instruction and training on:

- basic AI theory and techniques;
- basic notions of intelligent agents;
- types of intelligent agents, their architecture, strengths and limitations;
- issues involved in AMAS communication and interaction;
- possible application areas of the AMAS technology;
- demonstration on how AMAS revolutionise human-computer interaction;
- advantages of the agent-based approach to engineering complex software systems.

More particularly, the topics covered may include, among others, agent architectures (logic, reactive, BDI, hybrid), communication and interaction protocols (speech acts theory, agent communication languages, knowledge communication, Contract Net protocol, auctions, negotiation), biology-inspired agents (Ant Colony Optimization, Bio-Networking, Artificial Life), planning, learning and mobile agents, agent theories (intentional notions: information, motivation and social attitudes), AMAS software engineering methodologies (AAII methodology, Cassiopeia, Agent UML), Semantic Web basics etc. [11,12,13]. It is important, however, that some hands-on experience is also provided.

In the current work, we argue that NetLogo can act as the basis of an educational tool as well as an environment for AMAS research, provided that it is

enriched with certain features. NetLogo meets several of the requirements listed in Section 1. It is easy to install and provides with a plethora of interesting examples ready to run. The language is functional and the learning curve seems to be small, since someone is ready to produce a descent program in a short period of time. Additionally, it allows visualisation of the developed systems, which is extremely valuable to AMAS with spatial reasoning and behaviour as it helps them both to gain a better understanding as well as get immediate feedback from the simulation. However, NetLogo suffers from not being able to provide ready-made constructs for goal-oriented agents, communication and coordination. In order to compensate for the features that NetLogo lacks, we decided to work towards the implementation of an appropriate set of of extensions that aim to facilitate the development of BDI-like communicating agents.

3 NetLogo as a Modelling Tool

NetLogo is a modelling environment targeted to the simulation of multi-agent systems that involve a large number of agents. The platform aims to provide "a cross-platform multi-agent programmable modelling environment" [2].

The main entities of NetLogo are the patches, the turtles and the observer.[1] The *observer* simply controls the experiment, in which turtles and patches actively participate. *Patches* are stationary "agents", i.e. components of a grid on which *turtles* exist, i.e. agents that are able to move, "live" and interact. Both patches and turtles can inspect the environment around them, for example detect the existence of other agents, view the state of their surrounding patches/turtles, and modify the environment. Agents can be organised in groups under a user specified name, and thus agents of different *breeds* can exist in the simulation world. Probably the feature that most greatly enhances the modelling expressiveness of the platform is the fact that each patch and turtle can have its own user-defined variables: in the case of patches this allows modelling complex environments by including an adequate number of variables that describe it sufficiently and in the case of turtles it simply means that each agent can carry its own state, stored again in a number of user defined variables.

The NetLogo programming language allows the specification of the behaviour of each patch and turtle, and of the control of execution. The language is simple and expressive and has a rather functional flavour. There is an extensive set of programming primitives, good support for floating point mathematics, random numbers and plotting capabilities. Programming primitives include for example, commands for "moving" the turtles on the grid, commands for environment inspection (i.e. the state of other turtles and patches), classic programming constructs (branching, conditionals, repetition) etc. The main data structure is *lists* (following the functional Lisp approach), and the language supports both functions, called *reporters*, as well as procedures. Monitoring and execution of the agents is controlled by the *observer* entity that "asks" each agent to perform a

[1] The recent version (4.0) also offers links, not involved in this work.

specific computational task. The programming environment also offers GUI creation facilities, through which custom visualizations of the studied multi-agent systems can be created with particular ease.

As it has been argued elsewhere [14][2], we found NetLogo to be suitable for modelling agent systems that we are dealing with, since each NetLogo agent:

- perceives its environment and acts upon it,
- carries its own thread of control, and
- is autonomous,

i.e. it falls under the classic definition of agency found in [11]. NetLogo is an excellent tool for rapid prototyping and initial testing of multi-agent systems, particularly suited to systems with agents situated and operating in a restricted space, as well as an excellent animation tool of the modelled system. However, it lacks build-in support for implementing communicating agents with intentions and beliefs, making the task of modelling/testing more complex architectures and protocols rather hard.

4 Extending NetLogo with Libraries for Intelligent Agents

Being a platform that is primarily targeted to modelling social and natural phenomena, NetLogo fully supports the creation and study of reactive agent systems. Indeed, one of the first models that we studied was the Luc Steels Mars Explorer experiment [15] that presents many similarities to the ant colony foraging behaviour experiment included in the library models (ants can be modelled and studied as reactive agents [16,17]). However, the study of BDI agents that are able to communicate with explicit symbolic message exchange was not supported. Thus, taken into consideration the fact that NetLogo fulfilled the majority of our requirements we decided to extend the platform by providing one library for building simple BDI-like agents and one for FIPA-ACL-like message exchange, that are described in the sections that follow.

4.1 BDI-Like Agents in NetLogo

Although, we could have adopted as an option to link through the JAVA interface of NetLogo an already existing BDI development platform, as for example JAM [18], this would have made the installation of the complete platform a lot more difficult and would have significantly increased the learning curve, due to the complexity of such a fully fledged development environment. Thus, we decided to provide a simpler alternative, i.e. to develop limited BDI agents by providing the necessary primitives through a NetLogo library.[3]

[2] Actually, the excellent page that Jose Vidal (`http://jmvidal.cse.sc.edu/`) maintains for NetLogo models, was a starting point for our work.

[3] NetLogo being a simple platform, does not support libraries in the classic sense found in other programming environments; by the term library we refer to a set of procedures and functions that the user is given and includes in its own code.

Intentions in NetLogo. The simple BDI architecture that we have followed, follows a PRS-like [19] model, i.e. there is a set of intentions (goals) that are pushed into a stack; of course the implementation is far from delivering all the features of systems like JAM, but can still be effectively used in implementing simple BDI agents in the NetLogo simulation platform.

An intention (I) consists of two parts: the *intention name (I-name)* and a condition that we call *intention-done (I-done)*. The former maps to a NetLogo procedure (usually user defined), while the latter maps to boolean NetLogo reporter (function) (again usually user defined). The semantics are the standard followed by other architectures: an agent must pursue an intention until the condition described in the intention-done part evaluates to true. For example consider the following intention of an example ground unit agent working at a fire site (see Section 5):

```
["move-towards-dest [23 20]" "at-dest [23 20]"]
```

The above simply states that the agent is currently committed to moving towards the point with coordinates (23, 20) and it will retain the intention until the reporter `at-dest [23 20]` evaluates to true. Note that the user in this case has to specify both the procedure and the reporter, that map to the two parts of the intention, since they are not part of the built-in NetLogo primitives.

As mentioned, the main concept behind the present implementation is the *intention stack* where all the intentions of the agent are stored. Agents execute intentions popping them from the stack as shown below:

```
IF the intention stack is not empty THEN do:
  Get intention I from the top of the stack;
  Execute I-name;
  IF I-done evaluates to true THEN pop I from stack;
ELSE do nothing
```

The NetLogo implementation is rather straightforward: each intention is represented by a list of two elements, one for each part. The intention stack is a list stored in a specific "turtle-own" variable, i.e. each agent carries its own stack. The execution model is encoded in the procedure `execute-intentions`, that is called to invoke the agent's proactive behaviour. Notice that running a NetLogo simulation involves multiple execution cycles, invoking in each a procedure for each participating agent (or group of agents), thus ensuring their "parallel" execution. In consequence, procedure `execute-intentions` concerns only the execution of one intention in each such cycle.

The library also provides a set of reporters and procedures to the user for adding and removing intentions on the stack, inspecting the current intentions, set time-outs as *intention-done* conditions etc. For example, the following line:

```
add-intention "move-towards-dest [23 20]" "at-dest [23 20]"
```

will add the corresponding intention to the stack of the calling agent. A slightly more complicated example is shown below, that allows an agent to participate as a manager in a Contract Net protocol:

```
add-intention "select-best-and-reply" "true"
add-intention "wait-for-proposals" timeout_expired 15
add-intention "send-cfp" "true"
```

Since `add-intention` simply pushes intentions in the stack, such "plans" as the above have to be encoded in reverse order. Intention `send-cfp` concerns the agent broadcasting a call-for-proposals message to all agents and is executed only once, since it has as *I-done* condition the NetLogo reserved word `true`. `wait-for-proposals` remains as the agent's intention for the next 15 execution cycles (*clicks* according to NetLogo jargon), indicated by the provided `timeout_expired` BDI-library reporter. Finally, `select-best-and-reply` will be executed once, selecting the best contractor among the received bids. Note that `wait-for-proposals`, `select-best-and-reply` and `send-cfp` are user defined procedures.

Managing Beliefs. To further support the BDI architecture, facilities for managing *beliefs* were also created. Although, the latter was not really necessary, since it is rather simple to store any information on a related turtle variable, we have designed a set of procedures and reporters that would form an abstraction layer that facilitates the user to manage agent beliefs, without getting into too many details about how to program in the NetLogo language.

A belief consists of two elements: the *type* and the *content*. The former declares the type of the belief, i.e. indicates a "class" that the belief belongs to. Examples could include any string, e.g. "position" "agent" etc. Types facilitate belief management, since they allow to check for example whether a belief of a specific type exists or the removal of multiple beliefs at once. The content on the other hand, is the specific information stored in the belief. It can be any NetLogo structure (integer, string, list etc.). Obviously, there might be multiple beliefs of the same type with a different content, however two beliefs of the same type and content cannot be added. For instance, `["fire-at" [23 15]]` and `["location" [3 7]]` are examples of beliefs that the agent can have.

Belief management is done through a set of reporters and procedures that allow the creation, removal, checking of the current agent's beliefs. For example, the following line:

```
add-belief create-belief "fire-at" [23 15]
```

will include a belief of type "fire-at" with content "[23 15]" in the agent's beliefs. In the current implementation, all agent beliefs are stored in an agent own variable named `beliefs`.

4.2 FIPA-Like Message Passing

The ability to exchange symbolic messages is rather important in order to model agent communication and interaction protocols, such as the Contract Net protocol. Thus, it was necessary to somehow enhance NetLogo with explicit message communication primitives. Messages closely follow the FIPA ACL message format, i.e. are lists of the form:

```
[<performative> sender:<sender> receiver:<receiver>
     content: <content>..]
```

For example, the following message was sent by agent (turtle) 5 to agent 3, its content is "fire-at [23 15]" and the message performative (FIPA) is "inform".

```
["inform" "sender:5" "receiver:3" "content:" "fire-at [23 15]"]
```

A message may include the above fields (performative, sender, receiver, content), omitting others such as the ontology field specified by FIPA, assuming that all agents use the same ontology. The library, however, allows the creation and addition of any custom field that may be considered necessary.

Participating agents are uniquely characterized by an ID, which is in fact an integer automatically assigned by NetLogo at the time of their creation ("who" NetLogo variable). This naming scheme was adopted since it greatly facilitates the development of the message passing facilities. Message passing is asynchronous and a set of library reporters and procedures allow easily creating, sending, receiving, and processing messages between NetLogo agents. For example the code below (assuming that the calling agent is 8):

```
let  somemsg create-message "inform"
set  somemsg add-receiver 5 somemsg
set  somemsg add-content "fire-at [23 15]" somemsg
send somemsg
```

will send to agent 5 the following message:

```
["inform" "sender:8" "receiver:5" "content:" "fire-at [23 15]"]
```

Of course, both libraries (BDI and FIPA) take advantage of functional features of the NetLogo programming language. The exact message can be send by simply issuing the following one-line command:

```
send add-content "fire-at [23 15]" add-receiver 5
     create-message "inform"
```

It should be noted that there are also primitives that allow broadcasting a message to a group (breed) of NetLogo agents. For instance the following line of code sends a cfp message containing the location of a fire (see Section 5) to all agents of the breed "units":

```
broadcast-to units add-content (list "fire-at" list pxcor pycor)
     create-message "cfp"
```

Incoming messages for each agent are stored in the `incoming-queue` variable, which is in fact a list. This is a "user-defined" variable that each agent must have in order to be able to communicate. Sending a message to an agent simply means adding the message to its `incoming-queue` list; it does not require an

explicit receive command to be invoked on the receivers side. At any time the agent has the ability to obtain and process the messages from its queue using the reporters and procedures provided.

Both libraries, FIPA-ACL and BDI, were fully implemented in the NetLogo language and thus, their "installation" is trivial, since users have only to include the given library code in their models. The library offers limited debugging facilities, by allowing the user to inspect the list of messages exchanged and the current intention stack of the agents in the NetLogo programming environment. Limited error checking and debugging facilities was a design choice, in order to avoid having efficiency issues.

We have used both libraries for a number of years in a Intelligent Agents course: students have found them easy to use and did manage to implement multi agent systems that involve communicating BDI agents under an interaction protocol. The interested reader may find the libraries, brief manuals and examples at [20].

5 Forest Fires Scenario: A Case Study

Using the programming facilities described in the previous sections, we have designed and implemented a simulation of a multi-agent system in a fictional scenario that involves forest fires detection and suppression. The aim of the

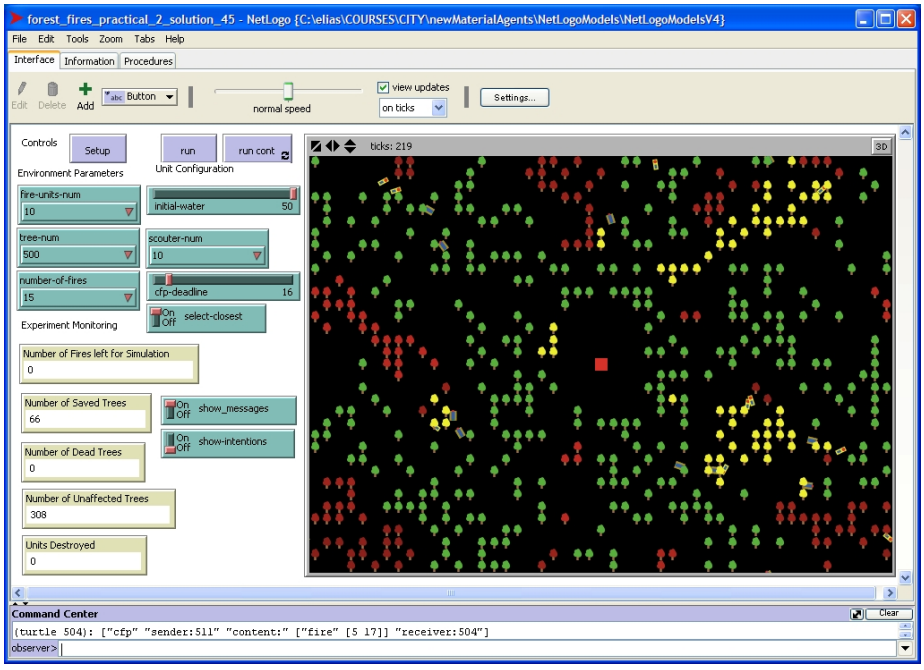

Fig. 1. NetLogo ScreenShot of the Forest Fires Scenario

```
to collect-msg-update-intentions
 let msg []
 let performative []
 while [not empty? incoming-queue]
 [set msg get-message
  set performative get-performative msg
  if performative = "cfp" [reply-to-cfp msg]
  if performative = "accept-proposal" [reply-to-accept-proposal msg]
  if performative = "reject-proposal" [do-nothing]                  ]
end

to reply-to-cfp [msg]
 ifelse current-intention = "find-target-fire"
  [let dist distancexy get-content msg
   send add-content dist create-reply "propose" msg                ]
  [send add-content "busy" create-reply "refuse" msg               ]
end

to reply-to-accept-proposal [msg]
 ifelse current-intention = "find-target-fire"
 [let crds get-content msg
  add-belief msg
  add-intention "send-confirmation"  "true"
  add-intention "put-out-fire" "fire-out"
  add-intention (word "move-towards-dest " crds) (word "at-dest " crds)]
 [send add-content get-content msg create-reply "failure" msg     ]
end
```

Fig. 2. Ground Units NetLogo Code using the BDI and FIPA-ACL libraries

specific AMAS was the constant monitoring of the forest area, as well as taking immediate action in the case of a fire spot, so that fire spreading is disallowed and creation of fire fronts is avoided. We have tried to model in the simulation many real-world features so that it is as close to reality as possible.

The environment (shown in Fig. 1) is inhabited by two types of communicating agents with different capabilities and constraints, that cooperate to perform the above task:

- *Ground units* are fire extinguishing autonomous vehicles that are able to travel around the forest environment and put out any fires they detect. Their speed is rather low and depends on the water supplies carried at each moment. Their sensors have a limited range, and while moving they have to avoid obstacles such as other ground units and forest areas already on fire.
- *Scouters* are small, light weight autonomous vehicles that can move around the environment relatively fast. Although they do not have the ability to extinguish a fire, they have close-range fire detection sensors. The only obstacle for scouters are trees on fire; they are small enough to co-exist in the same area with ground units or other scouters.

In the model, fire spots (i) appear randomly during the execution of the experiment, and (ii) are spreading in adjacent trees over time, if they are not put out, i.e. there is a close to reality fire spread model against which the agents were competing. Various parameters can be set as, for example, the number of fire spots that randomly appear (number-of-fires drop-down menu in Fig. 1), the density of the forest (tree-num drop-down menu in Fig. 1), the number of scouter and ground units, the initial water supplies of each unit etc.

Cooperation in this simple MAS is rather straightforward: scouters patrol the forest area, detect fire spots and inform ground units that move to the specific location and extinguish the fire. Cooperation takes place under the Contract-Net protocol where scouters assume the role of managers and ground units that of contractors: for each announced contract (cfp-message) by a scouter, bids (proposals) are submitted by ground units and one is awarded the contract. There is a single evaluation criterion of the bids, the distance of the ground units from the fire location. Although it could have been possible to directly implement the agents' behaviour in NetLogo, the BDI and FIPA-ACL libraries presented, greatly facilitated the development process. For instance, the code in Fig. 2 shows part of the ground units code, that involves message handling, proposal creation and commitment to an "accept-proposal" message that has as content the coordinates of the forest fire, i.e. a large part of the agent's reasoning. It should be noted that the partial implementation of this simulation was given as a coursework in the context of an Intelligent Agents course: details may be found at [20].

6 Discussion and Conclusions

We started using NetLogo as an educational tool five years ago, in the context of coursework of an undergraduate course in Intelligent Agents. Soon we started to realise the necessity of being able to simulate more complex AMAS than those composed of simple reactive agents. It was also encouraging to see a growing interest of the academic as well as research community towards this simulation environment. Both acted as motivations to create the BDI and FIPA-ACL NetLogo libraries.

In education, we have already seen the benefits that NetLogo provides. We use the same assignment template each year but with a different scenario (case study) such as agents rescuing victims in a disaster area, taxis transporting passengers in a city, vacuum cleaning a floor, luggage carriers solving logistics problems in airports, space satellite aligning to provide telecommunication services etc. [20]. The overall impression was that students enjoyed the practical aspect of the course. It is also clear to us that the overall student performance is increased due to better and deeper understanding of the issues around AMAS. Most of the students attributed much of the success of their solutions to the libraries provided. Learners really enjoyed the sense of seeing and experimenting with the virtual world they developed with minimum programming effort, thus having more time to design and prototype the "real thing".

In research, NetLogo provided us a platform in which we could immediately prototype and test our theoretical findings. Our interest in formal modelling of swarm intelligence and biology inspired AMAS exactly matched the initial NetLogo requirements [16,17]. We are currently investigating how formal models for goal-oriented agents exhibiting dynamic re-organisation can be directly mapped to NetLogo constructs, so that a semi or fully automatic translation would be possible. It is important that the libraries needed for such experimentation have been already implemented.

Through the two libraries described in this paper, we managed to have a platform that facilitates development of something more than a reactive agent. The libraries are by no means complete or fully fledged to meet the complete BDI and FIPA-ACL requirements. Future work is directed towards such extensions. For example, NetLogo offers a JAVA interface, through which a link to complete BDI packages, such as JAM, might be possible. Although we consider it unsuitable for educational purposes, as argued above, the idea could offer new opportunities for using the platform in research. In addition, we are considering a rather ambitious development of an extendible/customisable game platform in NetLogo, in which educators will be able to setup game environments and rules, such as tank battles, RoboCup etc., that would allow an even simpler use of the platform in the context of a course. Towards this direction, we have already implemented a prototype, but the work is not fully completed yet.

References

1. Joint ACM/IEEE Task Force on Computing Curricula: Computing curricula 2001. ACM Journal of Educational Resources in Computing 1 (2001)
2. Wilensky, U.: Netlogo. Center for Connected Learning and Computer-based Modelling, Northwestern University, Evanston, IL (1999), `http://ccl.northwestern.edu/netlogo`
3. Beer, M.D., Hill, R.: Teaching multi-agent systems in a UK new university. In: Proceedings of 1st AAMAS Workshop on Teaching Multi-AgentSystems (2004)
4. Hara, H., Sugawara, K., Kinoshita, T.: Design of TAF for training agent-based framework. In: Proceedings of 1st AAMAS Workshop on Teaching Multi-AgentSystems (2004)
5. Williams, A.B.: Teaching multi-agent systems using AI and software technology. In: Proceedings of the 1st AAMAS Workshop on Teaching Multi-AgentSystems (2004)
6. Beer, M.D., Hill, R.: Multi-agent systems and the wider artificial intelligence computing curriculum. In: Proceedings of the 1st UK Workshop on Artificial Intelligence in Education (2005)
7. Bordini, R.H.: A recent experience in teaching multi-agent systems using Jason. In: Proceedings of the 2nd AAMAS Workshop on Teaching Multi-Agent Systems (2005)
8. Fasli, M., Michalakopoulos, M.: Designing and implementing e-market games. In: Proceedings of the IEEE Symposium on Computational Intelligence in Games, pp. 44–50. IEEE Press, Los Alamitos (2005)

9. Behnke, S., Müller, J., Schreiber, M.: Playing Soccer with RoboSapien. In: Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y. (eds.) RoboCup 2005. LNCS (LNAI), vol. 4020, pp. 36–48. Springer, Heidelberg (2006)
10. Ferme, E., Gaspar, L.: RCX+PROLOG: A platform to use Lego Mindstorms^{TM} robots in artificial intelligence courses. In: Proceedings of the 3rd UK Workshop on AI in Education (2007)
11. Wooldridge, M.: An Introduction to MultiAgent Systems. J. Wiley & Sons, Chichester (2002)
12. Weiss, G. (ed.): Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Cambridge (1999)
13. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs (2002)
14. Vidal, J.M., Buhler, P., Goradia, H.: The past and future of multiagent systems. In: Proceedings of 1st AAMAS Workshop on Teaching Multi-AgentSystems (2004)
15. Steels, L.: 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 1990), pp. 8–14 (1990)
16. Gheorghe, M., Stamatopoulou, I., Holcombe, M., Kefalas, P.: Modelling dynamically organised colonies of bio-entities. In: Banâtre, J.P., Fradet, P., Giavitto, J.L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 207–224. Springer, Heidelberg (2005)
17. Stamatopoulou, I., Sakellariou, I., Kefalas, P., Eleftherakis, G.: Formal modelling for in-silico experiments with social insect colonies. In: Papatheodorou, T., Christodoulakis, D., Karanikolas, N. (eds.) Current Trends in Informatics, Patras, Greece, May 18-20. Proceedings of the 11th Panhellenic Conference in Informatics (PCI 2007), vol. B, pp. 79–89 (2007)
18. Huber, M.J.: JAM: a BDI-theoretic mobile agent architecture. In: Proceedings of the 3rd Annual Conference on Autonomous Agents, pp. 236–243. ACM, New York (1999)
19. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 1987), pp. 677–682 (1987)
20. Sakellariou, I.: Extending NetLogo to Support BDI-like Architecture and FIPA ACL-like Message Passing: Libraries, Manuals and Examples (2008), `http://eos.uom.gr/~iliass/projects/NetLogo`