MALSIM Detailed Proposal

Drew Wicke

February 10, 2012

The multi-agent learning simulator (MALSIM) will be a simulator and benchmarking tool for multi-agent learning algorithms written in Java. The goal of MALSIM will be to provide multi-agent learning researchers a common framework in which to compare and test various MAL algorithms. MALSIM will help to eliminate some of the empirical testing issues that are prevalent in the multi-agent learning community.

MALSIM will be written in Java using the Netbeans 7.0.1 IDE. I plan to use the following packages:

- Gamut - Game theory game generator [6]
- MPJ - MPI for Java [4]
- XStream - XML serialization of Java objects [3]
- JChart2D or LiveGraph - real time graphing libraries [1, 2]

Currently, MALSIM allows the user to interact with the GUI to:

- Create a *Batch* of *Tournaments*
- Save a *Batch* to XML
- Load a *Batch* from XML
- Choose a *Game, Eliminator* and *AgentSelector* from a list
- Choose what *Agent*(s) are to play in the *Game*
- Set properties for the *Game*, *Agent*, *Eliminator,* and *AgentSelector*
- Select a *Game* from the Gamut library
- Start a *Batch* as a threaded operation

A simplified view of how Batch, Tournaments, Game and Agent objects are structured is shown in figure 1. Also, in figure 4, the current GUI that allows the user to create Tournaments and set properties is shown.

I plan to extend current functionality by integrating MPJ, a Java implementation of MPI. I am choosing to add multi processing functionality using MPJ

in order to shorten the runtime when benchmarking algorithms. In order to use MPJ, the user must have MPJ installed and start MALSIM as an MPJ process. MPJ will be used to create and manage *Games*. MPI classes will be added to use MPJ, as seen in figure 2. *MPITourn* will create an *MPIGameRunner* using the *GameRunnerFactory*. *MPITourn* will create a user defined number of *MPIGameClient* processes. Therefore, when instantiating *MPIGameRunner* the id of the *MPIGameClient* process must be provided along with the *Game*. Then the *MPIGameRunner* can handle the MPI communication with the *MPIGameClient*. *MPIGameClient* will act as a wrapper for received *Game* objects. By using the *GameRunnerFactory,* I am abstracting the required interaction with *Games* and therefore, depending on whether the *Tournament* is MPI enabled, the correct communication protocol is followed.

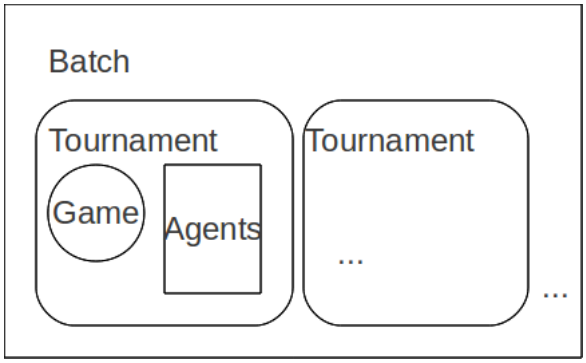

Figure 1: Simplified *Batch* structure. *Batch* is composed of *Tournaments* and *Tournaments* are composed of *Games* and *Agents*.

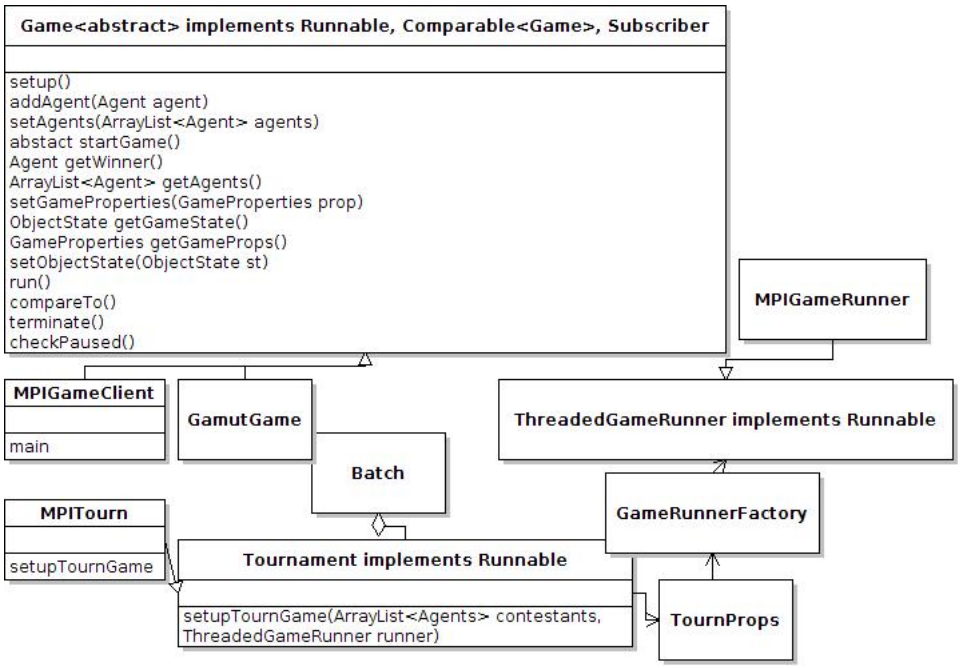

Figure 2: MPJ integration design diagram.

Once MPJ features are fully functional, the *MainView* screen will display a list of running *Tournaments* with options to cancel, pause and resume *Tournaments*. A use case would be:

User Pauses a Tournament

Pre-condition: Tournament is started

1. User chooses to pause a *Tournament*
2. System is notified of the user's selection
3. System pauses the *Tournament*
4. System notifies user that the *Tournament* is paused

The use cases for canceling and resuming *Tournaments* are similar. The class diagram is shown in figure 3. When the user starts the *Batch* the *BatchStateView* will display the running *Tournaments*. The *BatchStateView* will allow the user to change the state of *Tournaments* through the *BatchStateController*.

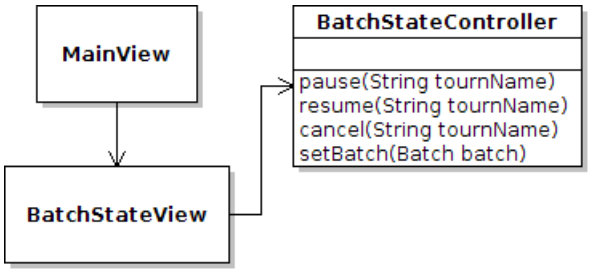

Figure 3: BatchStateView design diagram.

The next step is to write various multi-agent learning algorithms in order to better understand their capabilities. The first algorithm I plan on writing is the Adaptive Dynamic Learner algorithm [5]. The algorithm uses a fixed-size lookup table of Q-values that represents the utility of previous joint actions in order to decide what action to take. I then plan to augment the algorithm to allow for multiple levels of history. Therefore, the agent can have both a long-term and a short-term memory. I will then create rating and scoring mechanisms for the algorithms. Following the creation of the algorithms, a graphing library will be integrated in order to view statistics about the algorithms and game progress in real time and allow the user to save the charts as images.

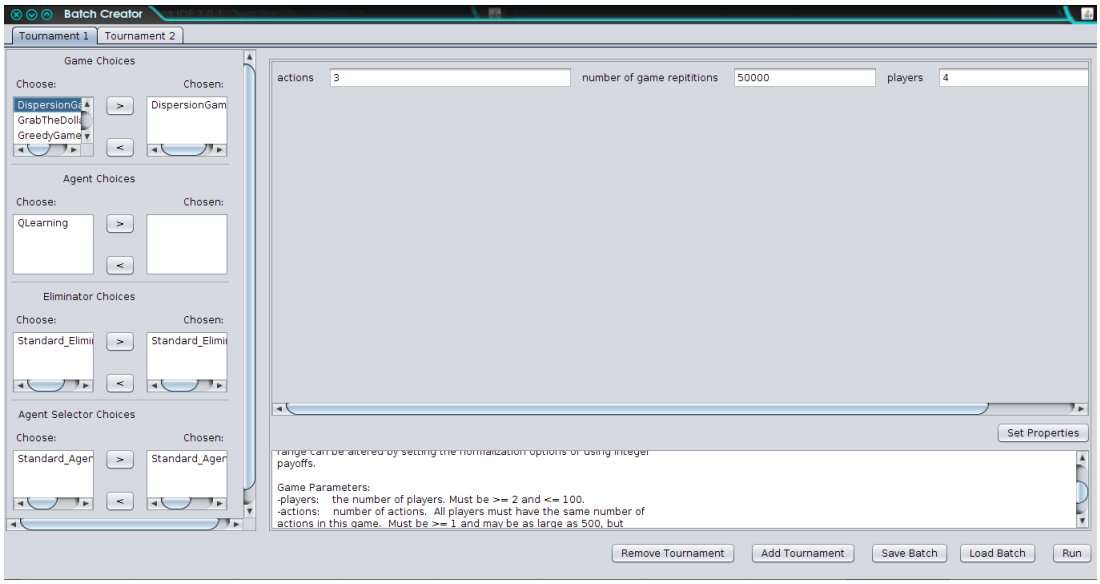

Figure 4: Current GUI that allows the user to create a *Batch* of *Tournaments*.

References

[1] Jchart2d. *http://jchart2d.sourceforge.net/*.

[2] Livegraph. *http://www.live-graph.org/*.

[3] Xstream. *http://xstream.codehaus.org/*.

[4] M. Bornemann, R. van Nieuwpoort, and T. Kielmann. Mpj/ibis: a flexible and efficient message passing platform for java. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 217–224, 2005.

[5] A. Burkov and B. Chaib-draa. Multiagent learning in adaptive dynamic systems. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 41. ACM, 2007.

[6] E. Nudelman, J. Wortman, Y. Shoham, and K. Leyton-Brown. Run the gamut: A comprehensive approach to evaluating game-theoretic algorithms. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 880–887. IEEE Computer Society, 2004.