COLLEGE OF INFORMATION SCIENCES AND TECHNOLOGY
THE PENNSYLVANIA STATE UNIVERSITY

A Users Guide to dTank 4.0

Frank Ritter, Damodar Bhandarkar, Bil Lewis, and Mark Cohen

Technical Report No. ACS 2007 - 2

March 22, 2007

---

acs.ist.psu.edu

Phone +1 (814) 865-4455 Fax +1 (814) 865-6426

Applied Cognitive Science Lab.
College of Information Science and Technology.
The Pennsylvania State University, University Park, PA 16802

Abstract

dTank 4.0 is a physical world based simulation environment that provides a highly usable adversarial environment. dTank was inspired by Tank-Soar and ModSAF systems. The primary applications of dTank are that it provides architecture and platform neutral test-bed for adversarial real-time cognitive models, and it also serves as a tool for teaching cognitive modeling, and agent creation. This report focuses on how to use dTank, with sections on controls, map-making, and its I/O, including as an example a way to provide Jess and Soar agents access.

Acknowledgements

This report was supported by the Office of Navy Research, contracts N00014-06-1-0164 and N00014-02-1-0021.

Table of Contents

1.0 Introduction to dTank 4.0

dTank was originally inspired by Tank-Soar (ftp.eecs.umich.edu/~soar/tanksoar.html), developed by Mazin As-Sanie at the University of Michigan. Tank-Soar provides an environment for Soar models to drive simulated tanks against other models. As with Tank-Soar, the goal of dTank 4.0 (dTank from here on) is to provide a test-bed for adversarial real-time cognitive models. Similar in many ways to AI-based computer game opponents, dTank was developed to take advantage of the flexibility of Java graphics and networking. Unlike Tank-Soar, in addition, dTank provides an agent architecture-neutral interface to the game server, so that humans and artificial entities built on virtually any platform can interact within the same environment over a LAN or the internet. dTank is a physical world based simulation environment that provides a highly usable adversarial environment.

Because dTank is built using Java's development tool kit, it presents uniform capabilities to models, agents, and humans alike operating from diverse computer architectures. Theoretically, it can be used for examining performance variability in situation awareness (Ritter, Kase, Bhandarkar, Lewis, & Cohen, in press) and architectural comparisons of competitive agents. From an application perspective, dTank is intended to serve in three distinct roles, teaching, cognitive modeling, and agent creation.

1. **As teaching tool.** dTank provides a simple, well-documented environment for experimenting with agent programming. A sample API for Jess and Soar agents is included with the dTank distribution, and instructions exist within this manual for creating interfaces for other types of agents. dTank supports protocol capture from human players. Students can use dTank to study cognitive modeling techniques including model-to-data fitting, in addition to general AI applications.

2. **As a modeling tool.** Due to the protocol capture facility, as well as dTank's communication layer, dTank provides a good environment for modeling both individual and team phenomena. Any cognitive architecture that can be made to support socket communication can interact with dTank. The communication layer is general enough that virtually any theory of multi-agent communication can be tested. One could, for example, study how a user works with a social environment defined by a set of agents with known characteristics, knowledge, and behavior.

3. **As a developmental test-bed for advanced AI applications.** The primary reason for the development of dTank was to create a tool to investigate the usability of distributed AI (multi-agent) systems. In particular, the ACS Lab is using dTank to inform the development of tools that help to explain the behavior (both actual and intended) of complex cognitive models to that of the modeler and human opponents of these models. This effort is aimed at improving the usability and usefulness of applied agent technologies, as well as to provide improved facilities for the validation of model behavior.

The rest of this document serves as a users guide that explains the basics of the dTank environment and provides some incites on how to make modifications for research investigations.

1.1 Installing dTank

1.1.1 dTank system requirements

dTank requires Java Virtual Machine (JVM), version 1.5.x or higher. We have tested it on Windows, Linux, and Mac OSX. dTank will load on a Mac running OSX that has the JVM, but it runs very slowly. If you understand this problem, or can help fix this, we welcome your input.

```
Note: We recommend Java version 1.5.11 on Windows. Later versions (e.g. Java
1.6 ) are known to have graphical flickers (only on Windows)  at the time of
this release.
```

1.1.2 Installation instructions

To install dTank, visit the ACS website http://acs.ist.psu.edu/dTank/ and follow the download instructions. The download file is in the form of a jar (e.g., dTank.jar). The easiest way to run dTank is to double click on the jar file.

However, you may be interested in making modifications on dTank environment to suite your research needs. In this case you may have to extract the dTank files from the jar for making such modifications. There are also several dependencies that must be noted. However, to run the basic dTank (without Soar or Jess), the only requirement is that you have Java installed in your machine. Secondly, configuration and map files must be available in the root directory that the simulation is being run from. The configuration file is used for allocating values to simulation parameters, where as the map file contains features regarding the terrain. However, for simple investigations, the simulation can run using inbuilt configuration and map files.

1.1.3 Battle Overview

To understand the dTank simulation, one must first understand the battle that is being studied. In dTank, the battle is a time-limited action between two opposing battalions, called the Allies and the Axis. (They are called nationalities, hence `Nationality.AXIS` and `Nationality.ALLIED`). In each battle, the battalion commanders are arbitrarily assigned to either the allies or the axis. Based on this, the simulation decides which battalion starts on which side of the battlefield (e.g., axis on the east, allies on the west).

Each tank has a commander that decides the tactics for that tank. The commander is a code object that runs in its own thread or process. It receives information from the tank in the real world (operating condition, location, what can be seen from the turret) and sends commands to that tank.

The battalion configurations are declared in the configuration file (read at startup) and comprise the commander code, the type of tank being commanded, and the number of tanks of that type. There are no limits on the number of tanks in a battle, nor which side gets which type of tank. It is perfectly legal (if rather odd) for a commander to have three Tigers, two Sherman's, a Matilda, and a T-34.

At the start of the battle, the tanks are lined up at opposite sides of the field and a commander is started for each. The simulation engine is then started. It moves the tanks and any projectiles fired according to the laws of physics. It prevents tanks from moving off the

battlefield or running over impenetrable objects, and assigns damage from hits as appropriate. Tanks that have been destroyed remain on the field, smoking. It is not necessarily obvious to other tanks that they have been destroyed.

If a display of the battlefield is desired, a window will be created and updated in a separate thread in the server process. Individual commanders may also create their own windows and display anything they desire. Sample commanders are included in dTank that parse the information messages from the tanks, implement simple tactics, and run displays. The programmer does not have use this sample code, although it is very convenient. Details about writing commanders are below.

1.2 Elements of the dTank simulation

When dTank is run, it starts with a startup display. This display screen is used to generate a battle, and allows the user to select a wide number of characteristics that he wishes to simulate within the battle. The display is shown in figure 1. The characteristics are described in the following sections.

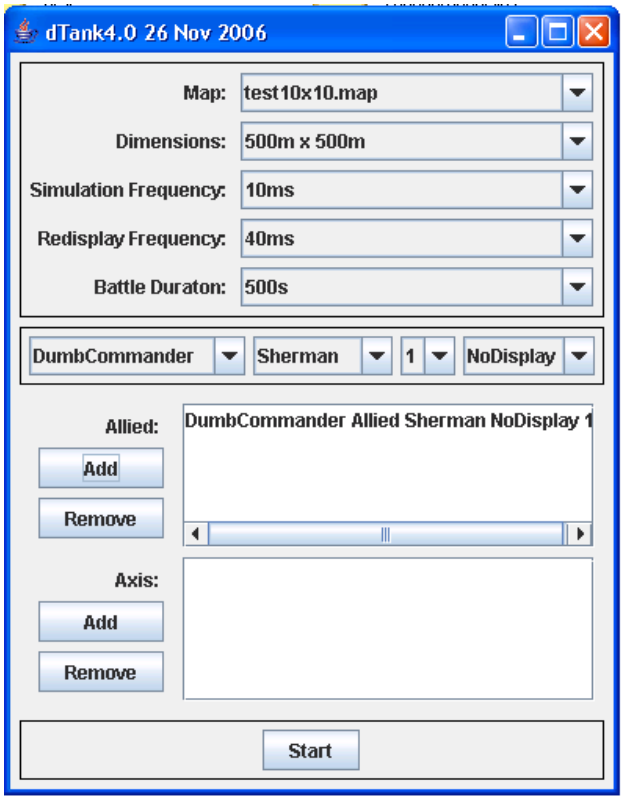


Figure 1. The Startup screen for dTank.

1.2.1 Map

The maps specify the size and the terrain that will be presented in the scenario. The basic version of dTank has four pre-defined maps. The maps `test10x10, test20x20, test50x50`

differ in the geographical area of the battle, while the map `El_Alameen` contains features such as rocks and hills placed in a predetermined order.

One can easily customize a map to suit their needs. A map can be created in a text file, with an extenstion `".map"`. To create a map, think of the terrain layout as a two dimensional grid over rows and columns. In a simplest case, the map with no features may look like this.

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

However, the terrain may consist of various features at different locations on the map. The features affect such things as whether or not a shell explodes, how much it hinders movement or sight, etc. This description of the map is loaded from a text file of size `[WidthInFeature, HeightInFeatures]`, with a set of rows defining the terrain features. Here's a 4x4 map with three low hills and one high one.

```
0 0 0 0
0 L L 0
0 0 L 0
0 0 H 0
```

dTank currently contain the following features. These features with their attributes are given listed below.

```
G = GRASS (Not Navigable, Fully Transparent)
L = Low Hill (Low Navigability, Fully Transparent)
H = High Hill (Not Navigable, Not Transparent)
R = ROAD (Navigable, Fully Transparent)
W = WOODS (Low Navigability, Low Transparency)
S = STONE (Not Navigable, Not Transparent)
```

1.2.2 Dimensions

The dimensions parameter specifies the world dimensions. The user can select among 6 possible dimensions: `500 m X 500 m, 1Km X 1Km, 5 Km X 5 Km, 10 Km X 10 Km, and 20 Km X 20 Km`. Once the dimension is specified at start up, the simulation automatically adjusts the features of the map into this dimension. For a more spread-out map, select a smaller dimension. When a smaller dimension (e.g., 500 m X 500 m) is selected, tank movements appear more rapid. In contrast, a higher dimension (20 Km X 20 Km) denotes a more zoomed out world, and movements appear to be slower.

1.2.3 Simulation Frequency

Simulation frequency is the frequency with which the world updates in the simulation. It denotes the frequency that the simulator advances the battle time. In each frequency update, all world changes occurred in the duration is aggregated and simulation parameters are updated. For example, a simulation frequency of 6 s will aggregate all changes from each of the 6 seconds into one update at the end of the 6^{th} second. This, in essence, gives a realtime delay in the main

simulation loop that allows the set of tank commanders to run. In the current dTank version, simulation frequency ranges from 10 ms to 100 ms.

1.2.4 Redisplay Frequency

While the simulation frequency is responsible for controlling the simulation updates, the redisplay frequency controls the interface. To be precise, the redisplay frequency controls the frequency with which the world is updated on the interface. This is independent of the simulation frequency. For a discretized version of the world, the user may want to select a higher redisplay frequency, and for a more continuous version of the world, a lower redisplay frequency is recommended. The redisplay frequency ranges from 40 ms to 1000 ms.

1.2.5 Battle Duration

The Battle Duration specifies total duration for which the simulation will run. This duration can range from 500 s to 10 min. At the end of this interval, simulation will shut down and all the data from the battle is recorded into the output files.

1.2.6 Commanders

Commanders are in essence controllers for each tank. In the dTank world, each tank has a Commander that tells it what to do. The Commander is completely independent of the simulation system and has very little information about the world itself. In dTank, each Commander runs in its own thread and communicates to the Server over a socket. All communications between commanders is through sockets. This happens based on the parameter `message_frequency` set by the simulation, and typically ranges from 1 to 5 s. In particular, the commander has no knowledge of other events on the battlefield. During the battle, the commander has no knowledge of how many opposing tanks are present nor how many have been destroyed. The only communication, the tanks have is through confederate tanks is via radio messages that are sent to the server, then to the tank controller and finally to its fellow tank controllers, and back to their commanders. Commanders are highly customizable, and creating a new commander is fairly easy. The later sections will show how new commanders, containing user-defined characteristics, can be created in dTank.

1.3. Components of dTank's Operation

1.3.1 Scanning the Environment

Once per `Parameters.messageFrequency`, each tank will execute a scan of its environment and send that to its commander. A typical scan includes only what a tank in the turret (hatch down) could see. Things that a tank can see are: other tanks, rocks, hills, and explosions in an arc from the turret heading. Many real world constraints may be maintained, in that sight may be reduced at distance by haze, and objects may be reported inaccurately. Upon scanning, the commander code may specify an appropriate response. As a note, it would also be reasonable to redefine scanning to look only for tanks and projectiles. One may start up the commanders by sending them the complete map layout and they would then keep that as their

permanent map. This would significantly reduce the computational cost of the scan. Of course if the computation costs are low enough, then it may not be worth the effort.

1.3.2 Tank Communications

Commanders may send radio messages to the other tanks on their side. The commander constructs a message string, and then sends the message across the socket to the controller for that tank. The controller then forwards the message to the controllers for all allied tanks. Those controllers then decide on what actions are to be conducted with the incoming message. A typical radio message looks like this:

```
<From:BattalionCmdr0 To:Tiger2
SetRadioMessage:From|BattalionCmdr0|FollowEnemyAt|324.4|224.6>
```

The parser for messages is very simple (it splits the string into an array of strings, splitting on space). Thus the above message would be split into an array of strings: `{"from", "Tiger2Commander", "To", "Tiger2", "RadioMessage", "EnemyAt|324.4|224.6"}`. It would then be parsed into a RadioMessage by the commanders.

Because a radio message must first be sent by a commander (as above) and then relayed to all the other commanders, the radio message itself must contain all required information. Hence, in the message below, which is what is sent from the tank to the commander, the only way to know who sent the radio message (`BattalionCmdr0`) is to include the sender in the radio message itself.

```
<From:Tiger0 To:TankCommander3 Time:72400
ReceiveRadioMessage:From|BattalionCmdr0|FollowEnemyAt|324.4|224.6>
```

```
Note: Because the radio messages are strictly for commander-to-commander
communications, the format of those messages is left completely up to the
programmer writing the commander code. As part of the sample code for dTank
commanders, the above format and two entries (From and FollowEnemyAt) are
supported as part of the class RadioMessage. The programmer writing
commanders is welcome to add new entries.
```

1.3.3 The Physical World

All the major real world considerations are possible to simulate: shells slow down as they travel and lose penetration power; 88 mm shells from a Tiger are much more potent than 75mm shells from a Sherman; nondestructive hits on a tank may damage armor, destroy the radio, or give the crew headaches, making them less accurate in following instructions; gravity and wind may be simulated. Turrets rotate at real speeds (24 degrees/sec on a Sherman), turns take longer at speed than at rest, and tanks have a realistic size (3m x 6m). The tank size means that on the display a tank's bitmap may overlap a feature or another tank when passing close by.

The current definition of wind resistance is simplistic - a shell loses 20% of its speed every second of flight (This is done by the `ProjectileMover`). The probability of hitting a tank is also simplistic- if the shell passes within `Parameters.hitRadius` (typically 5-10m) of a tank, it is considered a hit. Once hit, the damage to the tank is decided on by taking the impact energy

(mass x speed^2) and armor thickness (different for front, sides, and rear) into account, along with the amount of previous damage. This is defined in the `DamageController`. You may wish to build a new subclass of `DamageController` redefine how damage is assessed.

1.3.4 Operation Modes

There are two modes of operation. Realistic mode does lots of fancy stuff concerning visibility, and allows tanks to pass close to each other. Magic mode is for making the display behave as a person would expect in a video game. In particular, tanks occupy an entire feature-sized square, so the bitmaps of different tanks never overlap. The mode is set in the configuration file and is used to choose between using `RealisticAFVMover` and `MagicAFVMover` and ditto for `RealisticProjectileMover` and `MagicProjectileMover`. Thus, for magic mode, `Parameters.collisionRadius` is set to `featureSize` to prevent bitmaps from overlapping.

1.3.5 Startup

When the program starts up, it first reads the configuration file (optional name on command line `default.config`) that specifies the map file to use, several optional parameters (sleep quantum for display loop and simulation loop, seconds per cycle, Realistic or Magic mode, etc.), and the details for the combatants (battalion commander's name, type of commander, type of AFV, number). It then creates the map, builds a battlefield, and starts the Server thread, the Simulation thread, and the display thread.

1.3.6 The Adversaries

The two sides in a battle are called Battalions and determined by the name of their battalion commander. If more than two battalions are listed in the configuration file, then a series of pair wise matchings are created so that each battalion will fight each other battalion. For example, this configuration file says there will be three battalions (Jane, Jan and John), hence three battles.

```
Combattant: Jane JanesCommander Tiger Display 1
Combattant: Julie SmartCommander Sherman NoDisplay 5
Combattant: John SmartCommander Sherman NoDisplay 5
Combattant: John SmartCommander Matilda NoDisplay 5
```

The server then cycles through the battles, one-by-one, running the battle (`Battle.runBattle()`) to conclusion (`Parameters.battleDuration`) and writing out the results. `Battle.runBattle()` creates the combatants and starts up each Commander. The Commander may run in the same process as the server or may spawn a new process. In either case, it connects to the server via `port 3500` and starts a thread to communicate with the server via a `SocketCommunicator`. (If a Commander is in the same process as the server, it may shortcut the socket system by requesting a `DirectCommunicator`, which just passes the message strings directly.)

It is perfectly legal to start the commander process independently. (It probably is not useful to do so except when testing or running from a remote machine.) If the commander is written in Java, it may run in the server process. To spawn a new process automatically, you

need to write a RemoteCommander class that will just spawn the new process. This is how you run commanders written in different languages. Here's a remote commander setup to run a SmartCommander in a separate process:

```
Combatant: RemoteSmartCommander John Matilda 5
```

In RemoteCommander:

```
// Send this to a shell prompt:
// /usr/bin/java alcs.dTank.commander.SmartCommander Jon Sherman
Display

public static void startRemoteProcess(...) {
  ...
  Runtime r = Runtime.getRuntime();
  Process p = r.exec(new String[]{programName, commanderType,
            battalionCommander, afvType, number, withDisplay, arg0,
arg1...});
}
```

1.3.7 Time, Distance, and Headings

All time is measured in milliseconds from the start of the battle. These are battle times, not real times. Real time is used only in the display system and only for such things as deciding how long to display each bitmap in an explosion.

All measurements are in meters or meters/second. All headings are in degrees from North (top of the map), with East being 90.0. They are always normalized (0.0 < heading < 360.0). There are interfaces to translate speeds to KPH and all communications with commanders is done in KPH.

1.4 The dTank Display

The Simulation display consists of two main parts. The Game panel and the menu panel. The game panel displays the map, the tank's bearing, the tank turret's bearing, and the current position of the tank on the map. The menu panel displays the Battle information.

The display runs in a separate thread in the server process, but has no interaction with the simulation system at all. The display server runs as a thread and sleeps for `Parameters.displayFrequency` between each refresh. The display system is strictly for the human observer, and the simulation itself has no bearing without it. In each refresh, however, the `ServerDisplayLoop` thread grabs a lock on the `Battlefield`'s list of tanks and flying projectiles, maps each one onto the display, and updates the information labels. Keystrokes and mouse clicks in the control panel are forwarded to the commander, which may perform user-defined actions using them.

Figure 2 shows the display of the dTank simulation. The simulation starts with an initial position selected for each tank in the battle. The tanks move based on their initialization parameters and the strategy created by the developer. As the simulation progress, a score-card is maintained by the simulation that tracks the total number of tanks in each side, the number of tanks damaged, and the number of tanks destroyed in the ensuing battle. A Message box lists a chronological order of events in the simulation.

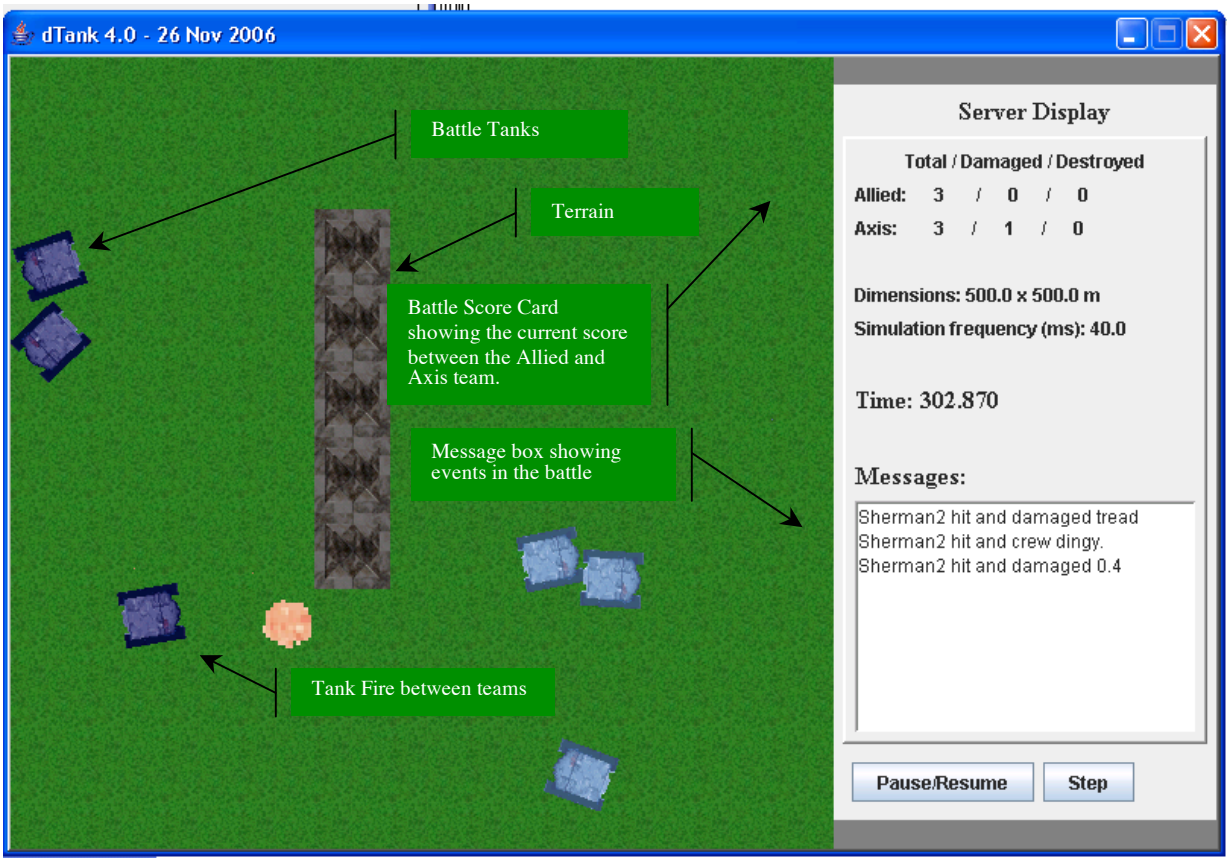


Figure 2. dTank Simulation Interface.

1.4.1 Plan-View Display Controls

The controls of dTank are not complex. However, it is important to note that all tank movement commands, as well as firing, are mutually exclusive. In other words, you cannot move in two directions at once, or fire while moving in any direction. Also, while shields can be up while you move, you cannot fire with shields on. However, aiming the tank's turret barrel can be performed at any point, during the execution of a keyboard command or not.

Controls are not operational for a tank until it completely enters the game (becomes opaque). Figure 3 lists the controls and their in-game effects that can be incorporated inside a commander.

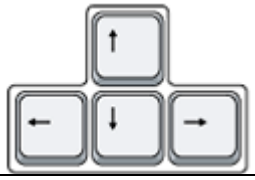
Control(s)	In-game Effect(s)
	[←] turns counter-clockwise; [→] turns clockwise; [↑] is move forward. [↓] makes the tank cannot go in reverse.
	[**X**] activates the shields for a short period of time. You do not need to hold down this button. Shields cost energy.
	[**Spacebar**] fires a shot from the tank. Holding down the spacebar allows multi-firing. If your target is moving, spread-fire by re-aiming the barrel while multi-firing.
	Left-clicking the mouse turns the turret towards the direction clicked.

Figure 3 Controls for the Plan-View Display

1.4.2 Logging player and agent behavior

To create a detailed log file of your dTank sessions (human, Jess or Soar interface), you will simply need to add or modify a line in your clients' Config.txt file. Make sure your Config.txt contains the following:

```
LOGGING     1
```

This will cause the server to log all communications between the client and server to a file named after the agent id. The file will be located in the directory in which dTank was started on the server machine (see Table 1 for example log format). If you are using your own API to the dTank server you will need to send commands to the server that will cause the server to start and stop logging. These commands are:

```
startLogging|
stopLogging|
```

The logging commands can be sent at any point during game play. To log entire sessions simply send the `startLogging` command immediately after agent connection.

Table 1. Portion of an example log file, including time (in ms), sender, and message.

```
1083247478749 SERVER ACK:nok:I245:|
1083247478968 CLIENT rotate|1|I248
1083247478968 SERVER ACK:ok:I248:|
1083247478968 SERVER INPUT:blocked:|bye
1083247479015 CLIENT rotate|1|I250
1083247479015 SERVER ACK:nok:I250:|
1083247479109 SERVER ACTION:2Dhuman:moveForward:|bye
1083247479171 SERVER VISUAL:Stone:X|6:Y|0:Stone:X|1:
Y|2:Stone:X|2:Y|4:Stone:X|5:Y|8:Tank:ID|2Dhuman:Color
|grey:X|8:Y|6:Orient|3:TurretRot|5.5566920060369425:
ShieldStat|no:moving|true:rotating|false:|bye
1083247479437 CLIENT rotateTurret|5.9614307724
1083247479765 SERVER INPUT:clear:|bye
1083247479796 SERVER
EVENT:turretRot:5.910121179565796:|bye
1083247480015 CLIENT rotateTurret|5.9614307724
```

1.5 Simple modifications to dTank

One of the important characteristics of dTank is that it provides an easy to use set of routines for the user to extend the simulation to suite one's study. One can easily extend the interface and create new Terrains, and at the same time, one can also extend commanders by creating customized code and writing candidate strategies into them. In this section, we provide you with an overview of modifying such elements as the Terrain, Armored vehicles, and Commanders.

1.5.1 Terrain

The first step to creating a new Terrain type is by adding a new constant in TerrainType.java, and then adding it to the initiation sequence.

Here's a typical example:

```
public static TerrainType ROAD = new TerrainType("ROAD",
"Road.JPEG", "R");

public static void initializeAll(displayFeatureSize) {
      ROAD.initialize(displayFeatureSize);
```

```
            GRASS.initialize(displayFeatureSize);
      }
```

Deciding how the new terrain behaves can be done in an ad-hoc manner. It is recommended that the user begin by adding methods (e.g., isSoggy()) to `TerrainType` and calling them only from the `Mover`. For example in class `MagicAFVMover` one could add:

```
public void move(double seconds) {
      afv.turn(seconds);
      Map map = Battlefield.getMap();
      TerrainType terrain = map.get(afv.getX(), afv.getY());
      double maxSpeed = afv.getMaxSpeed();
      double desiredSpeed = afv.getDesiredSpeedMPS();
      double p = terrain.isSoggy() ? 0.5 : 1.0;
      maxSpeed = maxSpeed * p;
      double newSpeed = (desiredSpeed> maxSpeed)? maxSpeed
:desiredSpeed;
      afv.setSpeedMPS(newSpeed);
      }
```

1.5.2 Armored Vehicle

Extending Armored Vehicles (refered as AFV here) are also fairly straightforward. If all the user needs is a new armored vehicle (or a platoon) that has the same types of capabilities of existing ones, this can be done by simply adding a new constant to AFVTypes.java and do the initializations in TerrainType. In the example below, a Matilda is created with following characteristics.

```
40 mm gun with a 600 m/s muzzle velocity
90 rounds capacity
78 mm front turret armor
No gyro-stabilizer
Can travel at 25 kph on-road (half that off-road, 1/4 in reverse)
Carries 211 liters of fuel and has a range of 80km.
```

This can be generated using the following code.

```
      public static final AFVType MATILDA = new AFVType("Matilda",
      40.0, 600.0, 90, 78.0, false, 25.0, 211.0, 80.0);

      public static void initializeAll(displayFeatureSize) {
            MATILDA.initialize(displayFeatureSize);
            SHERMAN.initialize(displayFeatureSize);
      }
```

In addition, if the user prefers to add some unique features that not included in the standard set of features, one will have to add new methods, and possibly add new instance variables and constructors. For example, if the user wishes to include the turret swivel rate for each tank, one could add a new variable swivelRate to AFVType and have the AFVMover look at it.

These features are also extendible to a number of units that can be simulated into a battle. For example, the method used above can be also used to create a foot soldier that may walk along beside tanks. Consider for example the following example of a foot soldier by assigning capabilities corresponding to an soldier.

```
10 mm gun with a 400 m/s muzzle velocity
15 rounds capacity
8 mm front turret body armor
No gyro-stabilizer
Can travel at 5 kph on-road (half that off-road, 1/4 in reverse)
Carries 2 liter equivalent of fuel and has a range of 100 km.
```

This can be generated using the following code.

```
public static final AFVType SOLDIER = new AFVType("Soldier ",
10.0, 400.0, 15, 8, false, 5.0, 2.0, 100);

public static void initializeAll(displayFeatureSize) {
      SOLDIER.initialize(displayFeatureSize);
}
```

Note that each time, a new armored vehicle is created, one may also want to display it in the simulation using bitmaps.

1.5.3 Commander

One of the most widely extended features in dTank is probably the Commander. Users interested in generating their own commanders have to follow a fairly easy set of procedures. The user who is writing the code to command a tank or a full battalion is not required to worry about the details of creating new tanks or terrain types. The only issue that the user must think of is the logic with which the commander operates with respect to the simulation. To write API's for Soar and Jess commanders, follow instructions in section 2 and 3. For creating Java commander conduct the following steps.

1. Copy the `DumbCommander.java` (or `SmartCommander.java`)
2. Rename the commander to `JoeCommander` (could be any name).
3. Change the File Name to `JoeCommander.java`
4. Rename the Constructurs and Class Names to `JoeCommander`.
5. Modify the `runCommander()` method in the `JoeCommander` class.
    a. This is where your logic for the commander goes.
    b. Add new variables and instantiate them create a user-defined structure for your commander.
6. Make sure that the new commander is in the Commander folder, and extended from the from the `Commander` Class.
7. Recompile and run dTank.

1.5.4. Damage Controller

This is where damage after a hit is determined and the AFV's condition is set. A single damage controller is created for the entire server. The basic algorithm looks at the shell's momentum and figures out how thick the armor must be to fully survive. The armor may be weakened, the radio may be destroyed, etc. A side impact may take out the tread. You get to define this anyway you want. The `ProjectileMover` is called when the shell passes within `Parameters.hitRadius` of the target. When this occurs, the mover decides if there has been a hit and the `damageController` decides what the damage is. Note that a damaged tank will continue to inform its commander of its status and what it can see. If a command comes in that cannot be carried out, the controller just fails to do it. If destroyed, the commander gets one last message telling it that the AFV has been destroyed. The commander's thread then exits.

1.5.5. InformationMessage, CommandMessage

An `InformationMessage` is the message that the `AFVController` sends to the commander. When it is determined that it is message time, the `AFVMover` does a visual scan of the environment and sends this along with the AFV's current status (location, heading, etc.). The message strings are parsed via a set of `Parser` classes. The parsed message is now turned into an object of type `InformationMessage` that has utilize call methods such as `getTurretHeading()` to retrieve corresponding information.

`CommandMessages` are similar, only they carry commands such as `SetThrottle:1.0`. They are parsed and passed to the `AFVController`. The controllers make calls such as `getThrottle()` and set the appropriate variables in the AFV as the controller sees fit. The `MagicController` will set the speed immediately. The `RealisticController` will set the throttle and let the `Mover` increase the speed bit-by-bit.

A non-Java commander would have to implement the parsing within itself. This should be fairly simple.

If you wish to implement a new message element, you will have to write a new parser for it and include it in the initialization sequence. Here's a typical parser that takes one token from the message and sets the throttle speed to `Double.parseDouble(token)`:

```
public class SetThrottleParser extends CommandParser {
  SetThrottleParser() {
        super("SetThrottle", 1);
  }
        public void parse(String[] tokens, int i, CommandMessage message)
{
        message.setThrottle(tokens[i]);
  }
}

and the initialization in CommandParser.java:

public static void initialize() {
  initialize(new SetThrottleParser());
  initialize(new SetHeadingParser());
  }
```

The actual commanders should never see the message strings. It will receive message objects and extract information from them via calls like `getSpeed()`. In the sample Java commanders, incoming information will placed directly into the `AFVModel` that the commander maintains. The commander will then look at the status of the AFV when and as it sees fit.

1.5.6. Unit Tests

There are sets of unit tests that do a modestly reasonable job of covering the major functionality of dTank. Most certainly they should be run after every change to the code base before check in. Most of the tests can be run as a set from `AllTests.main()`. A few create threads and don't currently have proper shutdown methods and must be run by hand. They are included in `AllTests` so you can find them, but they won't actually run as they are commented out.

Testing and running remote commanders requires that the host environment be set up just so. For UNIX and Windows, this means the right `PATH` and `CLASSPATH`.

1.5.7. The Configuration File

You may specify the configuration file on the command line or simply accept the default (`Default.config`). Here's a typical file:

```
BattleName: Tobruk
Repeat: 1
BattleDuration: 500
PhysicalDimensions: 2000.0 2000.0
Mapfile: testMaps/empty10x10.map
#Mapfile: testMaps/test50x50.map
#Mapfile: testMaps/empty50x50.map
#Mapfile: testMaps/battleA_20x20.map
SimulationFrequency: 40
DisplayFrequency: 40
SecondsPerSimulationCycle: 0.4
Mode: Realistic
DamageController: SimpleDamageController
#Mode: Magic
#LogMessages:
LogMessages: /tmp/Default.log
ResultsFile: /tmp/Default.results
#Combattant: TestACommander Mary Tiger noDisplay 1
#Combattant: RemoteSmartCommander Marge Sherman noDisplay 2
#Combattant: SmartCommander Joe Sherman noDisplay 2
#Combattant: SmartCommander Jane Tiger noDisplay 2
Combattant: HumanCommander Jane Tiger display 1
#Combattant: HumanCommander Jane Tiger display 0
#Combattant:  RemoteSmartCommander Jan Sherman display 2 arg0 arg1
Combattant:  SmartCommander John PKW_IV noDisplay  2
#Combattant: SmartCommander John Tiger noDisplay 1
#Combattant: JoesCommander John Sherman noDisplay 1
```

The entries are:

Note: Any line starting with "#" is ignored.

BattleName	is just the name printed out in the results file.
Repeat	is the number of times the set of specified battles are repeated.
BattleDuration	is the number of battle seconds a battle lasts. At the conclusion, the results are output and everything is reset for the next battle.
PhysicalDimensions	is the number of meters of the battlefield. (Must be square.)
Mapfile	is the text file that contains the map.
SimulationFrequency	is the slight misnomer meaning the number of milliseconds of real time between simulation cycles.
DisplayFrequencey	is the number of milliseconds between redisplays.
SecondsPerSimulationCycle	is the number of battle seconds that elapse between each cycle of the simulation system.
Mode	is either Realistic or Magic and determines which movers will be used.
DamageController	is the damage controller to be used.
LogMessages	with no argument logs to the console. With an argument, logs to that file.
ResultsFile	is the file to write the results into.
Combattant	declares that a given battalion commander shall have the specified tanks with the specified commanders. When a battle is begun, the tail of this line will be passed directly to the specified commander, exactly as the commander expects when run from the command line.

Thus, the line (all arguments required):

```
Combattant: SmartCommander Jan Tiger display 2
```

will be equivalent to:

```
% java acsl.dTank.commander.SmartCommander Axis Tiger display 2
```

The selection of Jan as being Allied or Axis is made when the configuration file is read and the battle pairing are set up. Here, only the battalion commander name (Jan) is used. All other references are as either Allied or Axis.

If you wish to run the commanders from the command line (as above), including a line that declares a battalion with zero AFV's will start a battle with no AFVs on one side. You can then start the AFVs from the command line as you wish.

If you decide to change the format of the configuration, make sure to keep the passing of the arguments are identical between the configuration file and the command line.

1.5.8. Typical Results File

The details for each tank below are actually on a single long line. They are shown with the continuation of the line indented.

```
========== Starting dTank4.0 version: 26 Nov 2006 ==========
========== Starting Battle 'Default'    Combattants: Axis: Joe vs.
Allied: Mary ===========
TestACommander Mary Tiger NoDisplay 1
SmartCommander Joe Sherman NoDisplay 1
Allies:     Total:1 Damaged:0 Destroyed:0
Axis: Total:1 Damaged:0 Destroyed:0
Axis: Sherman0 AFV_is:Operational Damage:0.0 Radio:Operational
Tread:Operational Shells_Fired:0 Successful_Shots:0
Allied:     Tiger1 AFV_is:Operational Damage:0.0 Radio:Operational
Tread:Operational Shells_Fired:0 Successful_Shots:0

========== Starting Battle 'Default'    Combattants: Axis: Jane vs.
Allied: Mary ===========
TestACommander Mary Tiger NoDisplay 1
HumanCommander Jane Tiger NoDisplay 1
Allies:     Total:1 Damaged:0 Destroyed:0
Axis: Total:1 Damaged:0 Destroyed:0
Axis: Tiger2 AFV_is:Operational Damage:0.0 Radio:Operational
Tread:Operational Shells_Fired:0 Successful_Shots:0
Allied:     Tiger3 AFV_is:Operational Damage:0.0 Radio:Operational
Tread:Operational Shells_Fired:0 Successful_Shots:0

========== Starting Battle 'Default'    Combattants: Axis: Jane vs.
Allied: Joe ===========
SmartCommander Joe Sherman NoDisplay 1
HumanCommander Jane Tiger NoDisplay 1
Allies:     Total:1 Damaged:0 Destroyed:0
Axis: Total:1 Damaged:0 Destroyed:0 Axis: Tiger4 AFV_is:Operational
Damage:0.0 Radio:Operational Tread:Operational Shells_Fired:0
Successful_Shots:0
Allied:     Sherman5 AFV_is:Operational Damage:0.0 Radio:Operational
Tread:Operational Shells_Fired:0 Successful_Shots:0

========== Battle over ==========
```

1.6 Creating a communication channel to the server

For an agent to join the dTank server, it needs to do the following steps.

1. Connect to the dTank server socket and send the following message: `"agent join"`
2. Server responds with the following: `"join|<<int>>"`, where `<<int>>` is a port number on the server host
3. Close the connection to the server socket and open a new connection to the port number specified in the join message. Before connecting, pause for half a second or so to allow time for the server to create a new listener-socket. Your agent is now connected and you will receive an initialization string over the socket.

1.6.1 Commands to the Server

Commands are sent by the client to the server, so it can update the state of the world. Most commands to the server should incorporate unique command ID's so that acknowledgement `(ACK)` functionality works properly. Where necessary, the command ID will be noted by "<id>" in the following syntax descriptions.

MOVE
The tank moves forward one cell when this command is acknowledged by the server
Format: `moveForward||<id>`

TURN
Turns the tank 90 degrees clockwise or counter-clockwise.
Format: `rotate|<<int>>|<id>`
For this command, <<int>> may be 1 for clockwise or 0 for counter-clockwise.

ATTACK
The tank fires in the direction faced by the turret upon successful acknowledgement of the command by the server.
Format: `attack||<id>`

ROTATE TURRET
Rotates the tank's turret to an angle specified in radians, where 0 is north.
Format: `rotateTurret|<<double>>|`

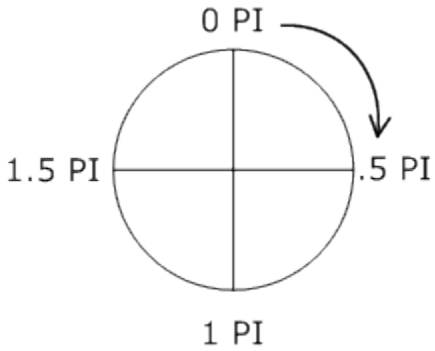


Figure 4. Radian measures.

SCAN
The agent requests a scan for detailed stats on a specified tank within its visual field.
Format: `scan|<<string ?tank-id>>|<id>`

RADIO
The agent communicates to other tanks in the dTank world. There are three message types:

1. Broadcast messages can be used to post the message to all other tanks in the dTank world,
2. Unicast messages are agent-to-agent communications where there is only one-recipient

3. Team messages are sent to all agents on the same team as the sender.

Format: `RADIO|<<string broadcast|team>>|<<string>>,`
Format: `RADIO|unicast|<recipient-id>|<<string>>`

SET TEAM
The agent announces its team allegiance. This command can be sent to the server at any point in the game, but is generally most useful at startup.
Format: `setTeam|<<string>>`

SET NAME
The agent announces its name. This is an optional field used to identify different tanks, rather than by ID number. This command can be sent to the server at any point in the game, but is generally most useful at startup.
Format: `setName|<<string>>`

START LOGGING
This tells the server to start a log for this client. This can be used to create a log for humans or agents.
Format: `startLogging|`

STOP LOGGING
This tells the server to stop logging this client's actions.
Format: `stopLogging|`

1.6.2 Messages from the Server

During game-play, agents will receive un-requested messages from dTank about the actions of visible players and game status.

1.6.3 Initialization

The following initialization string is received when the agent connects to the dTank server for the first time.

Format:

```
InitialSettings:Name|<<string>>:Color|<<string>>:Health|<<int>>:W
eapon|<<int>>:
Shields|<<int>>:X|<<int>>:Y|<<int>>:Orient|<<int>>:TurretRot|<<do
uble>>
```

1.6.4 Server updates

During game-play, your agent will receive various message types from the dTank server.

STATUS

Periodically (about every 2 seconds) each agent will receive a status string that informs it of its current attributes

Format:

```
STATUS:Health|<<int>>:Weapon|<<int>>:Shields|<<int>>:X|<<int>>:Y|
<<int>>: Orient|<<int>>:TurretRot|<<double>>:ShieldsStat|<<string
up|down>>
```

VISUAL

About every 2 seconds each agent is sent a string containing information for all objects that your agent can currently see. This will include all objects in a 100 degree angle centered on the agent's turret orientation. Agents cannot see objects that are obscured by other tanks or stones.

Format:

```
VISUAL:{Tank:ID|<<string>>:X|<<int>>:Y|<<int>>:Color|<<string>>:
ShieldsStat|<<string>>:Orient|<<int>>:TurretRot|<<double>>:}*{Sto
ne:X|<<int>>:Y|<<int>>:}*
```

Please note that the '*' indicates there can be more than one of them in the string sent. Two stones, for example, can be recorded in one message.

Example: Two stones in view.

```
VISUAL:Stone:X|5:Y|5:Stone:X|2:Y|2:
Example 2:  1 tank in vew
VISUAL:TankID|thisTank:X|3:Y|4:Color|Cyan:ShieldStat|Up:Orient|3:
TurretRot|3.14:
```

SCAN

If an agent successfully request a scan of another tank in the game, the server will send a scan message. This gives important and more detailed information about rival tanks that cannot be discovered with the visual message.

Format:

```
SCAN:<<string ?tank-id>>:Health|<<int>>:Weapon|<<int>>:Shields|<<int>>
```

RADIO

If another agent sends a message to your agent, you will receive a radio string.

Format: `RADIO:<<string ?tank-id>>:<<string ?message>>`

ACK

The server will acknowledge all commands sent to dTank as either acceptable (ok) or not acceptable (nok). To use this functionality, your agent interface must send command ids with every command string as described in the output section.

Format: `ACK:<<string ?id>>:<<string ok|nok>>`

INPUT

When your agent's environmental status changes (possible to move, not possible to move, reloading cannon) your agent will be notified. Agents may not move forward or turn while moves are not possible. Likewise, agents may not fire their cannon while reloading.

Format: `INPUT:{<<string blocked|clear>>}|{reloading|<<string yes|no>>}`

EVENT

When your agent is hit, killed, or moves its turret to a new orientation, an event message will be sent.

Format: `EVENT:<<string hit|died>>`

ACTION

When tanks that your agent can see engage in some action, your agent will be notified with an action message.

Format:

```
ACTION:<<string ?tank-id>>:{moveForward|}|{raised-
shields}|{lowered-shields}|{rotate|<<string clock|c-
clock>>}|{rotateTurret|<<double>>}|{fired| <<double>>}
```

2.0 The dTank/Soar API

{Input from Mark}

3.0 The dTank/ Jess API

The dTank/Jess API is a Java library that makes it possible to write intelligent tanks for dTank using Jess. Jess is a rule-based engine and scripting environment. Jess enables the development of software systems with reasoning capabilities developed using knowledge supplied in the form of declarative rules. In contrast to other agents such as Soar, Jess is light-weight. Also, because of its powerful scripting language, Jess provides easy and direct access to a majority of Java API's. Hooking Jess agents to dTank is relatively simple. In most cases, it is as simple as assigning the class path to the Jess executable file. Once the class path is set, dTank can be connected to the Jess reasoning engine using standard import statements. The Jess engine uses the Rete algorithm (Forgy, 1982), and all references to Jess are through the set of Rete interfaces provided by Jess.

Upon initializing Jess in the dTank environment, a common working memory is generated between the dTank environment and the Jess agent; from this point onwards all communications between the two is through this working memory. Jess interfaces support standardized storing and transmission between the agent and environment. This is performed through the use of templates that represent facts in the environment. Fact templates can be defined either in the Jess agent or the dTank code. A sample definition of the tank template in dTank storing facts associated with the tank is:

```
rete.executeCommand("(deftemplate tank (slot name)
      (slot nationality)(slot distance)(slot speed))")
```

Each time the dTank environment is scanned, all environmental features are transmitted back to the common working memory and stored as facts in corresponding templates. The following code demonstrates the storage of the environment facts in the common working memory.

```
String assertFact = "(assert
      (tank " + "
      (name " + tankname + ")" + "
      (nationality \"" + nationality + "\")" + "
      (distance " + dist + ") " + "
      (speed " + speed + ")" ;
rete.executeCommand("(assertFact)");
```

Each addition of a fact triggers the Rete algorithm, which attempts to match rules with the new fact(s). Based on the matched rules, actions are specified for the Jess commander. Writing rules in Jess is based on a Lisp-like notation. For example, a strategy that fires on Axis tanks traveling at a speed greater than 50 mph may bewritten like this:

```
(defrule tankRule
      (tank (name ?nm) (nationality "Axis")
      (distance ?dist)(speed ?spd&:(> ?spd 50)))
 => (fire))
```

Similar rules can be created for any of the commander actions mentioned earlier, such as turning, forward movement, chasing, and rotating. Because of Jess's compatibility with Java,

Jess commands can be executed both from the Jess module or the main Java code. The major advantage of employing the Jess engine is the ease with which reasoning rules and a knowledge base of facts can be stored in a separate module without having to recompile the central dTank program.

3.1 Installation

Installing the interface is fairly easy. Create a directory named dTankJess (any other names are also fine) and copy the dTank.jar file (you do not need to extract the contents of the jar file) into this directory.

Place the following files into this directory:

```
The Jess file.
The Jess API Package.
```

3.1.1 The Jess file

The Jess file, with an extension ".clp" contains declarative rules that will be loaded into the dTank program once the simulation begins. Two important things must be kept in mind while using the Jess file.

File Naming. Care should be taken in naming the jess file. It is possible that the dTank environment is looking for a particular file name when the loading begins. Before placing the file into the root folder, consult the latest dTank manual (this document) for appropriate name of the file. At the time of this writing, the file was named "JessTank.clp".

File Syntax. dTank works with the Jess agent based on a common working memory. This working memory provides a method to store and retrieve facts using templates. To avoid errors, the data format utilized in Jess must correspond with that of the dTank program. Again, the user must consult the latest dTank manual to obtain all required input and output formats so that an understanding is maintained between the two.

There are many ways to create templates in the clip file. One could use the Java API available in Jess and the template could be written as:

```
Deftemplate d = new Deftemplate("tank", " Data base of tanks",
rete);
d.addSlot("name", Funcall.NIL, "STRING");
d.addSlot("nationality", Funcall.NIL, "STRING");
d.addSlot("distance", Funcall.NIL, "FLOAT");
d.addSlot("speed", Funcall.NIL, "FLOAT");
rete.addDeftemplate(d);
```

In contrast, the template could also be written in the native Jess language, as:

```
deftemplate tank (slot name)(slot  nationality)(slot
distance)(slot speed))
```

3.1.2 The Jess API Package

Jess can be licensed for commercial purposes, and is available at no cost for academic use. To obtain Jess, contact Craig Smith at casmith@sandia.gov for prices and commercial licensing terms, or for a research based academic license. A trial download is also available at http://herzberg.ca.sandia.gov/jess/. Upon downloading, an executable package called `Jess.jar` of approximately 410 KB in size will be available.

3.1.3 The Jess/dTank file structure

The root folder for running Jess/ dTank must look like this.

```
dTankJess\
      Jess.Jar
      dTank.jar
      JessTank.clip
```

To run Jess/ dTank, open the command prompt and go to the dTankJess\ folder and type the command:

```
java -cp Jess.jar;dTank.jar dTank
```

The best way to see how Jess/dTank works is to observe facts being created and rules being matched. To see this, turn on the `(watch all)` command in either the jess file or the dTank code. If clip file is used, this command should be written as the first line of code the clip file. A sample output is shown here. In this output, each time the Jess tank spots an enemy tank, it is stored in the working memory. This rule is fired at each scan if an enemy tank is spotted. In the following trace, the three tanks spotted and their attributes are printed out in the command prompt.

```
      Jess File: Working Memory initialized
      MAIN::tankRule: +1+1+t
      MAIN::dist_if_low: +1+1+1+t
       ==> f-0 (MAIN::tank (name "tank0") (nationality "Axis")
(distance 204.8) (speed 250.64))
      ==> Activation: MAIN::tankRule :  f-0
      facts added  new tank
      FIRE 1 MAIN::tankRule f-0
       Jess File: New Tank Added
       ==> f-1 (MAIN::tank (name "tank1") (nationality "Axis")
(distance 200.0) (speed 250.6))
      ==> Activation: MAIN::tankRule :  f-1
      facts added  new tank
      FIRE 1 MAIN::tankRule f-1
       Jess File: New Tank Added
       ==> f-2 (MAIN::tank (name "tank2") (nationality "Axis")
(distance 371.7) (speed 255.6))
```

4.0 References

Ritter, F., Kase, S., Bhandarkar, D., Lewis, B., & Cohen, M. (In Press). dTank Updated: Exploring Moderated Behavior in a Light-weight Synthetic Environment, *Proceedings of the 16th Conference on Behavior Representation in Modeling and Simulation.* Orlando, FL: U. of Central Florida.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19, 17-37.

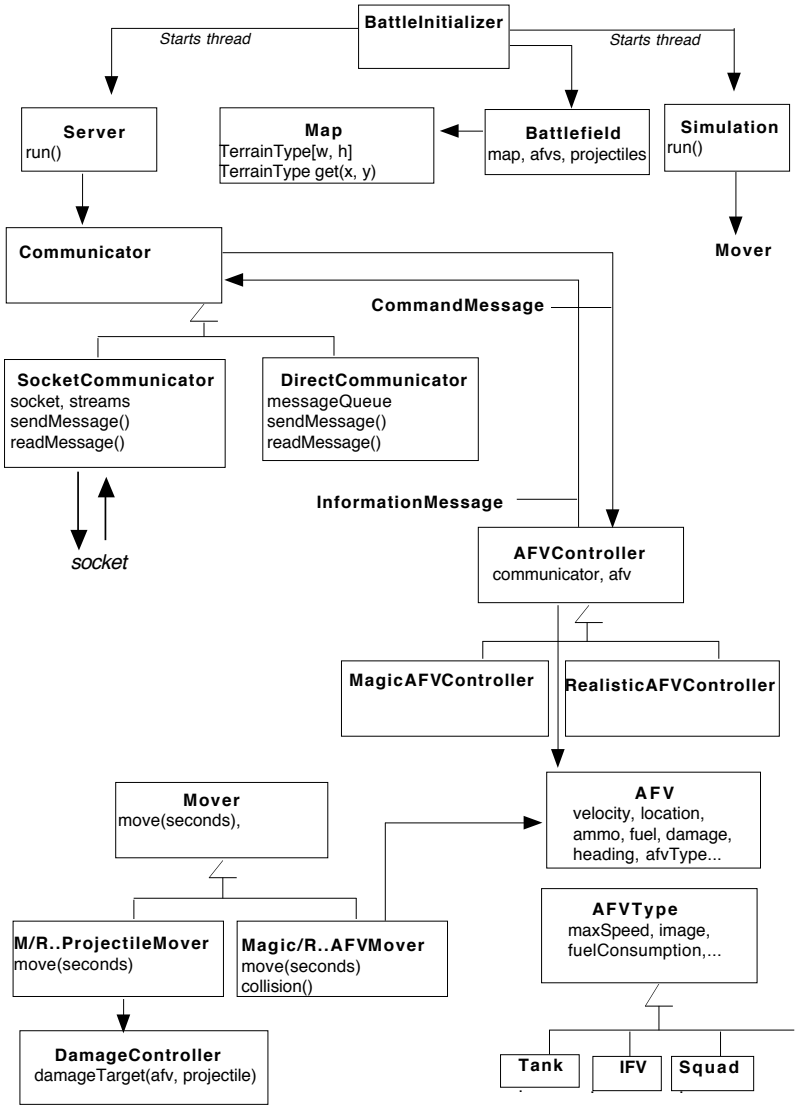
Friedman-Hill, E. (2003). *JESS in action*. Greenwich, CT: Manning Publications.

Appendix I. Design Diagram

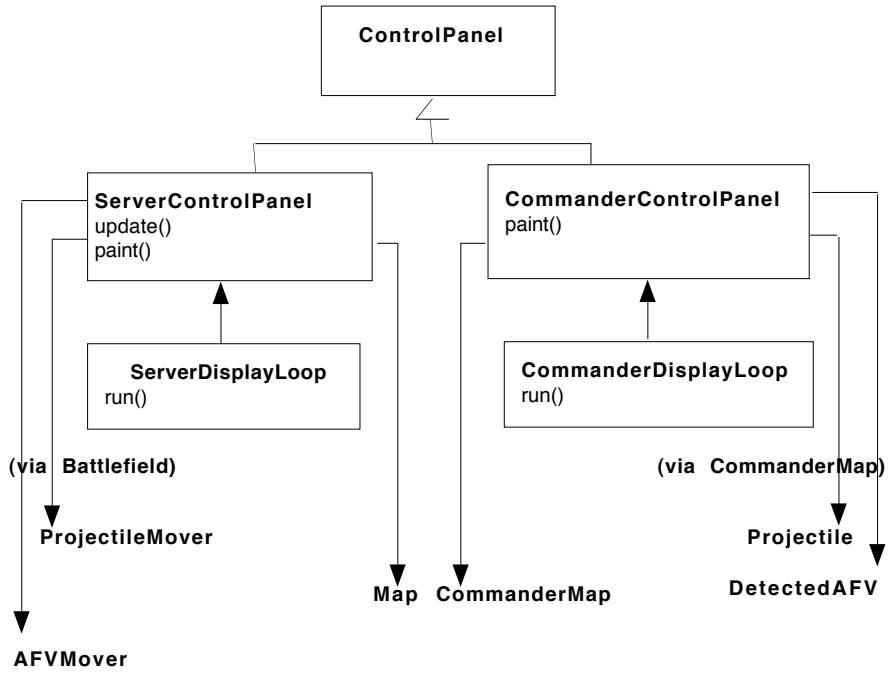
A1. Design Diagrams

The UML diagrams shown in this section provides an overview from a programmer's perspective; these show the major classes involved in dTank and how they are connected to each other.

A1.1 UML diagram for Battle Component of dTank



A1.2 UML diagram for Control Panel Component of dTank



A1.3 UML diagram for Communication component of dTank