In: *Proceedings of the 16th Conference on Behavior Representation in Modeling and Simulation*. 51-60. 07-BRIMS-014. Orlando, FL: U. of Central Florida.

dTank Updated: Exploring Moderated Behavior in a Light-weight Synthetic Environment

Frank E. Ritter
Sue E. Kase
Damodar Bhandarkar
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802
frank.ritter@psu.edu, skase@ist.psu.edu, dnb133@psu.edu

Bil Lewis
Computer Science Department, Tufts University
Medford, MA 02155
bil@cs.tufts.edu

Mark A. Cohen
Business Administration, Computer Science, and Information Technology
Lock Haven University
Lock Haven, PA 17745
mcohen@lhup.edu

Keywords: simulation environments, agent architectures, behavior moderators

ABSTRACT: *We provide an update on dTank (Morgan et al., BRIMS 2005), a highly usable adversarial environment. It can be used for examining performance variability in situation awareness and architectural comparisons of competitive agents. First, the new design and implementation details of the updated dTank environment are discussed. In-progress models constructed with several cognitive and agent architectures (Java, Jess, and Soar) are then noted. Next, in the moderated behavior section, we present preliminary analyses of embedded performance delays in reaction to battlefield environmental conditions. Noise factors and variability in delay length at the tank commander level lead to different battle outcomes. Finally, we note some changes that will be required for dTank to better model situation awareness. Light-weight agent-construction environments such as dTank fill an important need for experimentation and prototyping tools that support quick scenario development and behavior implementations in a usable programming environment available to a wider user audience. These types of modeling tools can both raise and answer critical questions concerning agents' awareness of their surroundings and resulting behavior.*

1. Introduction

We present dTank as a highly usable construction tool for studying the effects of behavior variability in a simple simulated battlefield environment and for inter-architectural comparison of models and agents. In this report we examine how it can be used to study the effect of moderators and situation awareness on performance. dTank, a Java-based tool, has been designed to present uniform capabilities to models, agents, and humans. It uses socket communication methods to provide uniform connections to all models.

First, we present the system design of dTank and discuss how this design facilitates parallel comparison of models, agents, and humans. Second, we introduce several dTank models and agents built using different cognitive and agent architectures. Third, we perform an example analysis, examining how variation in battalion performance influences measures such as destroyed and damaged tanks, and successful shots on target. Fourth, and finally, we conclude with a discussion of these comparisons, their implications, and future work with dTank.

2. dTank's Design

dTank was designed with two main criteria, that (a) multiple software models/agents should be able to use a universal interface for connection, and that (b) the human and software players should have parallel capabilities available to them. There were several secondary criteria as well, including that the

system be interesting and easy to use, and that it logged the agents' behavior for later analysis and playback.

The design of dTank was inspired by Tank-Soar, a tank game developed by Mazin As-Sanie and included with Soar distributions. dTank provides very similar functionality, but its implementation has drifted away somewhat from Tank-Soar. Most importantly, dTank was intended to provide a light-weight alternative to modeling programs such as ModSAF (Loral, 1995). It starts to realize the idea of a Java-implemented synthetic environment that Elliman wrote about in a joint review of modeling projects for synthetic environments (Ritter, Shadbolt, Elliman, Young, Gobet, & Baxter, 2003). dTank is available online at http://acs.ist.psu.edu/ projects/dTank/).

2.1 Architecture and Interface

dTank, version 4.0, is a physical world-based simulation environment for armored fighting vehicles (and other fighting units, all called platforms). The simulated battle is fought on a square grid map scaled 0.5 to 10 kilometers (other dimensions are possible).

dTank 4.0 is a major rewrite of dTank 3.0 (Morgan et al., 2005). This new version eliminates a vast amount of the complexity existing in the latter version, while providing a cleaner interface and an enormous increase in performance. In particular, dTank 4.0 includes a dozen WWII tank platforms (e.g., Sherman, Tiger, T-34) that behave realistically. Adding new platforms is relatively simple. Additionally, there are multiple terrain types, methods of computing vehicle damage, and behavior modes.

dTank utilizes a client-server architecture and a socket-based interface. A server is started up as a Java program. The server displays everything, and runs the simulation. When a tank commander connects to the server, it is given a tank on the battlefield, and can then send commands to move the tank. The commanders are given updates every two seconds of what is visible to them on the battlefield. This update rate was chosen because it is the estimated scan rate of Navy pilots determined by another study (Councill, Haynes, & Ritter, 2003), and emphasizes that perception is not instantaneous.

The commanders receive information from the battlefield and then generate commands in the form of text-strings. Converter files must be created on a per-agent/architecture basis. Currently commanders are available in Jess, Soar, and Java.

The battlefield terrain consists of a wide range of features (e.g., grass, woods, roads). The features affect the exploding of fired shells and the movement speed and visual qualities.

A battle is a time-limited action between two opposing battalions, called the Allies and the Axis. Each side has a battalion commander, who is presumably the programmer of the tactical code. In each battle, the battalion commanders are arbitrarily assigned to be either the Allies or the Axis. The Axis is positioned on the west side of the battlefield with the Allies positioned on the east.

Down one level from the battalion commander are the tank commanders. Tank commanders decide the tactics for their tanks. A tank commander is an object that runs in its own thread or process. It receives information from the tank (operating condition, location, what is visible from the turret) and responds by sending commands back to the tank. The tank commander has no knowledge beyond the information gathered by the tank. To support studies examining teamwork, commanders can send radio messages to one another if the programmer wishes to model coordinated actions.

The battalion configurations are typically declared in a configuration file including commander code, the types of tanks being commanded, and the number of tanks of that type. There is no limit on the type and number of tanks participating in a battle. If a display of the battlefield is desired, a window can 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. When the battle ends, statistics about the tanks (e.g., hits taken, hits scored, damage sustained, number of shells fired) are written to a results file.

2.2 The Simulator

The simulator advances the battle time by 0.1 to 10 seconds on each clock tick. A clock tick is typically 10-100 ms. This gives a real-time delay in the main simulation loop that allows the tank commanders to run and the display to repaint. The simulator is strictly physical with no knowledge of displays or commanders. When projectiles are in the air, the seconds/cycle is decreased to enable visibility by human observers.

2.3 The Tanks

The tank types (platforms) have knowledge of their location, speed, heading, operational status, and ammunition remaining. Tanks do not have direct access to information about the environment or other tanks. A Controller object communicates with the tank

commander who in turn tells the tank what to do (e.g., set the throttle to full, turn the turret). Controllers run in their own threads and communicate to the server over a socket. A Mover object moves the tank. The Mover object is responsible for calculating new locations, actual speed, and collisions with other objects.

2.4 The Displays

The server display runs in a separate thread in the server process, with no interaction with the simulation system. The display system is solely for the human observer.

2.5 Scanning the Environment

Every two seconds, the tank's Mover executes an environment scan sending the gathered information to its Controller. A typical scan includes only what a human in the turret (hatch down) can see and the human game display provides—tanks, rocks, hills, and explosions in an arc of 45 degrees around the turret heading. Sight may optionally be reduced by haze and other environmental conditions. Using the environmental information, the tank commander decides what actions to take.

2.6 The Adversaries

The server cycles through a battle, running the battle to conclusion and writing out the results. The Controller 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.

Controller processes can run independently. If the Controller is written in Java, it may run in the server process. To spawn a new process automatically, a new RemoteCommander class is written that will spawn the new process. This is how commanders written in different languages are run.

Figure 1 shows part of the interface that a human user sees when his dTank display has connected to a server. Figure 2 shows a more complete view including several tanks in the server window.

Models can use the state-representation differently, either acting directly on information as it is made available, or building internal state-representations. Models then decide upon an action that affects the environment. This action is sent to dTank in a formatted string held in a buffer for time-synchronization purposes.

Aside from state information, which is sent at regular intervals, software agents are also informed of important events, such as being hit, killed, and various types of damage. This is to allow modelers some flexibility in the behaviors of models in difficult situations.

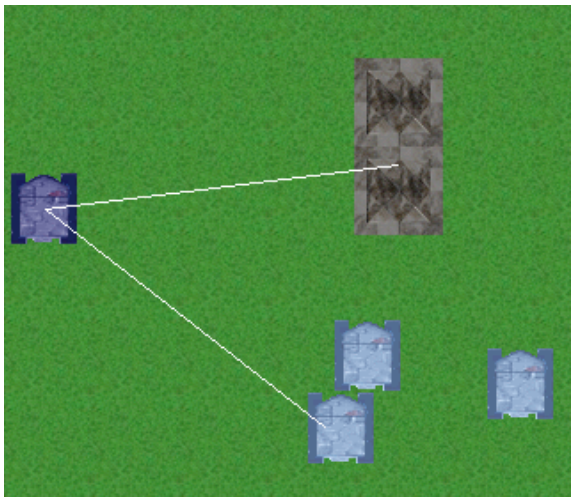

Figure 1: A detailed view of dTank from the commander's interface

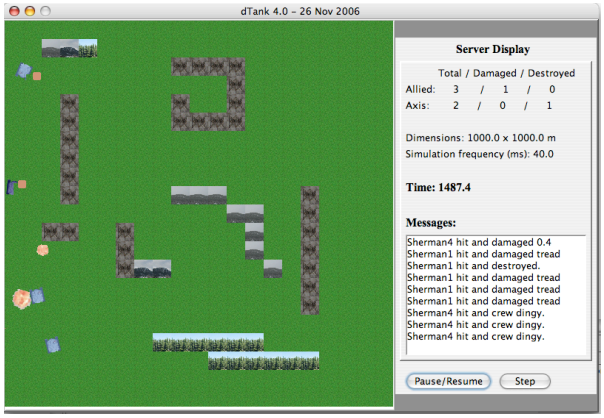

Figure 2: View of the dTank server window

3. dTank Models

In order to understand several different architectures, dTank models have been created in Java, Jess, CAST, and Soar. Work on ACT-R (Anderson, Bothell, Byrne, Douglass, Lebiere, & Qin, 2004) and CoJACK (Norling & Ritter, 2004; Ritter & Norling, 2006) models are in-progress. Previous work has used dTank to compare Soar and CAST (Yen, Yin, Ioerger, Miller, Xu, & Volz, 2001) models to understand how and when to communicate (Sun, Council, Fan, Ritter, & Yen, 2004). We note a few of them here being actively developed and included to illustrate this use of dTank.

Actions in dTank such as moving forward, turning, rotating the turret, and firing can be composed into strategies such as Wandering, Hunting, Chasing, Retreating, and Attacking that are broadly applicable to both models and humans. The most common strategies employed by current dTank models are described next.

Wandering is moving in a random or semi-random fashion across the battlefield while rotating the turret. This strategy initiates when no target is visible and terminates upon acquiring a target. It is composed of a series of turret rotation, move-forward, and turn commands. *Retreating* is moving away from a known target location. It is composed of a series of turn and move-backward commands. *Attacking* is a rapid series of commands that aim and fire the turret. Usually this is composed of an alternating sequence of aim and fire commands until the enemy can no longer be seen or is destroyed. This strategy initiates upon sight of an enemy tank and terminates when no enemy tank is visible.

3.1 Built-in JavaTanks

Several sample Java tank commanders are built into the dTank 4.0 distribution for use in tests and to populate the environment. These commanders parse the information messages from the tanks, and implement simple battlefield tactics.

JavaTanks are capable of executing any of the strategies noted above. For example, the SmartCommander JavaTank wanders the battlefield in search of enemy tanks. When a target is detected the SmartCommander initiates an attack-like strategy by setting the tank heading and aiming the turret towards the target location. For firing purposes, the SmartCommander evaluates the distance between his tank and the target tank. If the SmartCommander is too far away from the target, no fire command is given, and if too close to the target, a retreat strategy is executed. The portion of the SmartCommander code shown below illustrates the use of target distance evaluation and the retreating action.

```
if (distance < 200.0){
  writeDisplayMessage("Too close. Retreat!");
  retreating = true;
  commanderCommunicator.setDesiredHeading
    (afvModel.getHeading());
  commanderCommunicator.setThrottle(-.50);
  continue;
} else
  commanderCommunicator.setThrottle(1.0);
if (distance > 600.0){
  writeDisplayMessage("Too far to fire: " +
  Utility.shorten(distance));
  continue; }
```

In contrast to the SmartCommander, the DumbCommander does not scan the battlefield for information. Instead, the DumbCommander aimlessly wanders the battlefield firing random shots.

3.2 Built-in JessTank

Jess is a rule-based engine and scripting environment developed by Sandia National Laboratories (Friedman-Hill, 2003). 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:

```
(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.

3.3 Built-in SoarTank

Soar defines behavior as movement through a problem space; a high-level organizational tool that can be used to partition knowledge in goal-relevant ways (Lehman, Laird, & Rosenbloom, 1998). A problem space consists of a goal, states, and operators, and is goal directed. When a Soar model in unable to accomplish its current goal, an impasse is generated causing the model to enter a child problem space with the goal of resolving the impasse.

In order to create a Soar dTank commander, Soar had to be hooked up to dTank using the Java to Soar API and the Soar Markup Language (SML). The API makes it possible for Java applications to communicate with the Soar Kernel using SML to define the format of the messages exchanged. Both the API and SML are available as part of the standard Soar distribution.

The connection between Soar and dTank is initiated by creating a Soar Kernel and loading the Soar productions that define the commander's behavior. These productions watch the input link for changes in the environment and respond by placing appropriate actions on the output link.

During each decision cycle, dTank monitors the Soar output link for pending actions such as moving or rotating the turret. If actions do exist, they are performed by the commander. Next, dTank populates working memory with the current state of the environment (as it is seen by the commander) so that the Soar model can react to environmental changes. This cycle continues until the tank is destroyed or the user terminates the battle.

The Soar commander demonstrates the Wander, Retreat, and Attack strategies described in section 3. It was created using the Herbal high level behavior representation language (Cohen, Ritter, & Haynes, 2005; Ritter et al., 2006). These strategies are implemented using a top problem space and one for each of the operators: wander problem space, attack problem space, and retreat problem space.

The goal of the Soar commander's top problem space is to destroy all enemy tanks before its own tank is destroyed. However, the top space relies heavily on the behavior provided by the wander, attack, and retreat sub goals. As the Soar commander operates within the dTank environment, it continually switches between these problem spaces in order to respond to changes in the environment. For example, if there is no enemy spotted and the commander's tank is healthy, the commander enters the wander problem space with the goal of finding an enemy tank. On the other hand, if the commander's tank is damaged, it will enter the retreat problem space with the goal of avoiding enemy fire. The relationships between these four problem spaces are shown in Figure 3.

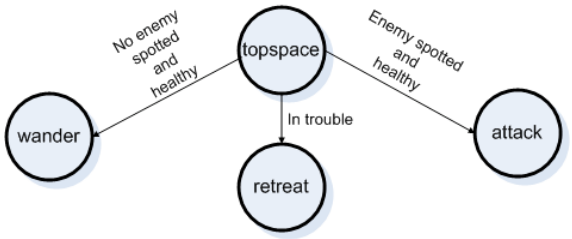

Figure 3: The problem space hierarchy for the Soar/dTank commander

Each problem space in the Soar commander contains operators that help accomplish the main goal. The top space acts as a controller—its operators detect environmental conditions that move the model into one of the remaining three problem spaces. The wander, retreat, and attack problem spaces contain operators that carry out actions that generate wander, attack, or retreat behavior. The operators in each problem space are shown in Figure 4.

The Soar commander described here contains 33 productions and serves as an excellent starting point for creating more complicated dTank commanders in Soar.

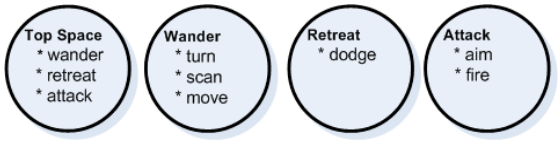

Figure 4: The operators used by each problem space in the Soar/dTank commander

4. Moderator-Influenced Behavior

In many existing synthetic environments, the individual entities in the simulation execute the same task in the same way, ignoring differences between individuals, and even the variability of a given individual over time. Recent thinking recognizes the inherent variability of humans (i.e., differences in cognition and physiology from one individual to the next), and the necessity of modeling this to improve the realism of synthetic forces (e.g., Gluck, Gunzelmann, Gratch, Hudlicka, & Ritter, 2006; Hudlicka & Pfautz, 2002; Silverman, 2004). This variability, even after differences in knowledge are removed, arises both from individual differences, where different abilities can lead to marked differences in behavior, and also from behavior moderators—internal and external factors, typically related to time and environmental conditions, that moderate individual differences.

Endsley (1988) defined Situation Awareness (SA) as "the perception of elements in the environment within a volume of time and space, the comprehension of their meaning and the projection of their status in the near future." Thus, the construct of SA suggests three levels (Matthews, Pleban, Endsley & Strater, 2000). Level 1 involves the perception of the elements of a particular environment. Level 2 involves understanding what those elements mean. Level 3 requires the individual to translate the perception and understanding of the environment into a projection of future events likely to occur within that environment.

To explore the effect of level 1 SA (the perception of elements in the environment) and to demonstrate how the effects of moderators on SA could be explored, a simple experiment was conducted using two dTank commanders derived from the SmartCommander described earlier.

Tank commanders have the capability to gather information from the battlefield environment at specific time intervals. By varying the time delay between environmental scans, individual differences in perception and response time to battlefield events can be simulated. For instance, an extended delay between information gathering tasks and action could mimic a fatigued or stressed commander.

Four sets of dTank battles, 30 battles per set, were staged between two teams, four AlertCommanders versus four DullCommanders. Starting positions, Allies and Axis, were alternated by battle to eliminate any initial positional advantage. A constant battle time limit of 1,000 seconds was used. During the first set of battles a default environmental scan delay of one second was used. For the second set of battles, the environmental scan delay for the DullCommanders was increased to two seconds, while AlertCommanders' delay remained at the default—one second. For the third set of battles the DullCommanders' scanning delay was again extended by one second, totaling three seconds. During the last set of battles, the DullCommanders' scanning delay was again increased by one, totaling four seconds.

Figure 5 shows the number of destroyed tanks for the four sets of battles with increasing environmental scanning delays applied to the DullCommanders. Solid lines with circle markers are the tanks under control of DullCommanders that have been destroyed during the battle. The dashed line with square markers indicates AlertCommanders' destroyed tanks.

The top plot of Figure 5, with no additional scanning delays, shows both DullCommanders and AlertCommanders nearly equal in battle wins based on the number of destroyed tanks. Across the 30 total battles, DullCommanders won 13 battles while AlertCommanders won 14 battles, and three battles were considered a tie with an equal number of tank losses per side. In the second plot of Figure 5 DullCommanders have an additional delay of one second. In this case, AlertCommanders accumulated twice as many wins (16 battles) over DullCommanders (8 battles), however, the number of ties has doubled as well (6 ties). In the third plot, with DullCommanders scanning delayed two additional seconds, Alert-Commanders' performance dropped slightly to 13 battle wins over DullCommanders' 7 battle wins. In this case one third of the total battles are considered a tie in number of destroyed tanks (10 out of 30). The bottom plot in Figure 5 shows that with an additional three-second delay the DullCommanders are nearly obliterated with only two wins out of 30 battles. AlertCommanders accumulated 23 winning battles with 5 battles considered a tie.

Table 1 presents a summary of winning battles—least number of destroyed tanks equals a win. In each battle set the length of the environmental scanning delay applied to each team is shown along with the corresponding wins and ties. The figure and the table together show that the environment and current scenario are noisy, but that there are effects in the expected range, and that these effects are of an interesting size.

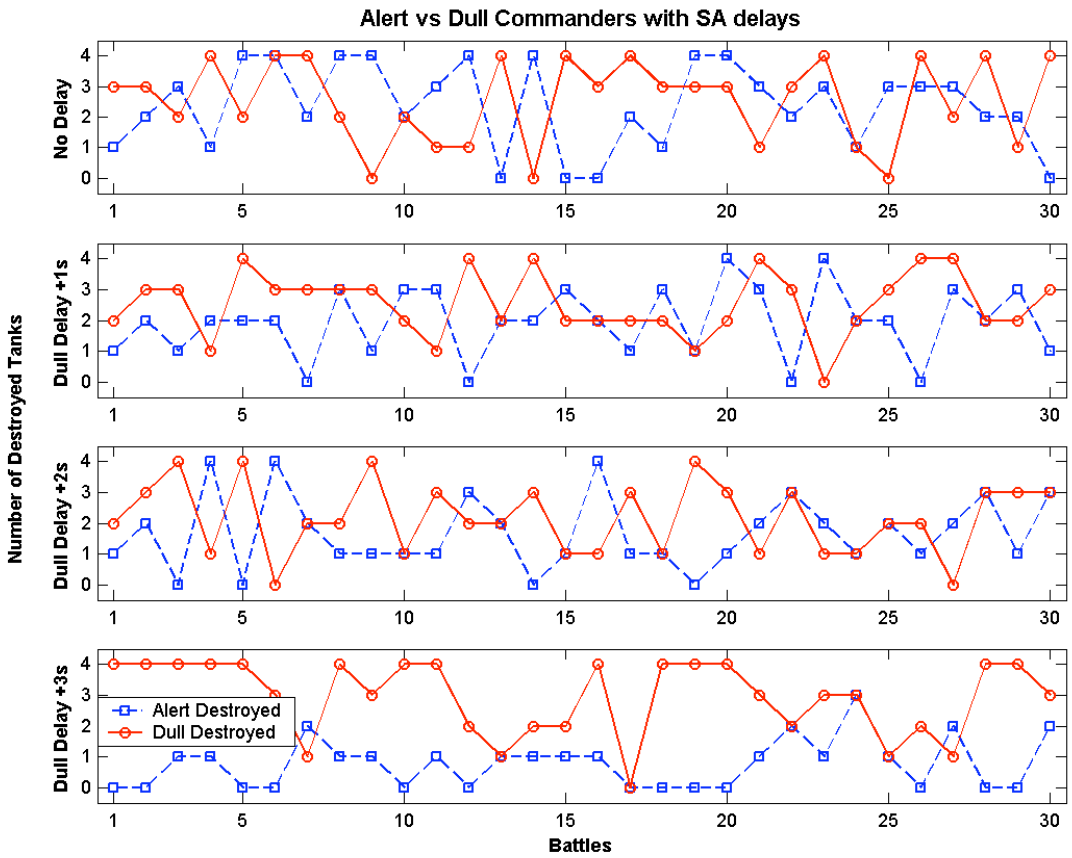

Figure 5: Number of destroyed tanks by delay type for Alert vs. DullCommander teams

Table 1: Number of winning battles by team

Battle Set (30)	Team	Additional Delay (s)	Wins	Ties
1	Alert	0	14	3
	Dull	0	13	
2	Alert	0	16	6
	Dull	1	8	
3	Alert	0	13	10
	Dull	2	7	
4	Alert	0	23	5
	Dull	3	2	

Figure 6 displays the means for the number of tanks destroyed and damaged, and the number of successful shots fired on opposition's tanks across each set of 30 battles. In the top plot, when no additional scanning delays are applied (leftmost on the x-axis) the destruction is approximately equal—the markers overlap. Moving along the x-axis, with additional delays added to DullCommanders, the lines begin to separate into a downward trend for AlertCommanders' destroyed tanks, while the DullCommanders' means stay within the 2 to 3 destroyed tank range. The greatest difference in mean destroyed tanks appears at the rightmost point of the x-axis, with three seconds of delay added to the DullCommanders.

Similarly, in the bottom plot of Figure 6, the mean number of successful shots fired per tank is tracked by increasing amounts of environmental scanning delay added to DullCommanders. As expected, the theoretically equal initial no-delay state (leftmost x position) for both teams is nearly equal empirically. As the additional delays are added to the DullCommanders, their successful shots sharply decrease down to a mean of 2 shots. In contrast, AlertCommanders' successful shots increase to a mean of 6.5 as they win more battles and the Dull-Commanders become more passive in the environment.

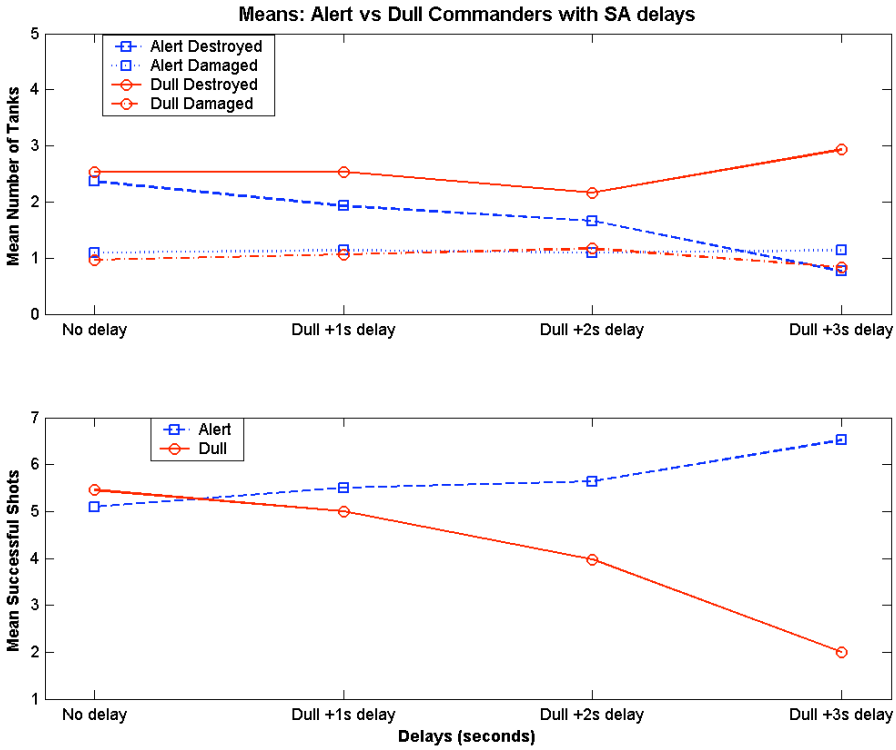

Figure 6: Mean number of tanks destroyed, damaged, and successful shots by delay type per tank for Alert vs. DullCommander teams

Despite the somewhat noisy simulation environment, there is an obvious effect of scanning delay on both the number of tanks destroyed and successful shots fired. The number of damaged DullCommander tanks goes down slightly at the end as they are being destroyed instead. Besides scanning time, tank agents could react to the information gathered from the battlefield in many other ways, for example, implementing different tactical strategies or coordinating battalion behavior via radio messages.

Although preliminary and small in scale this experiment, including code modifications to the tank commanders and executing all 120 battle runs, was completed in a relatively short period of time—approximately two days by a graduate student not particularly familiar with the dTank modeling tool or Java. Clearly, this pace is an advantage over more complex synthetic environments for exploratory research and analysis.

5. Summary and Future Work with dTank

This experiment shows that the dTank environment provides a useful testbed for concepts in this area. Tanks can be created that have at least simple situation awareness, and the effects of changes to SA can be modified with results appearing in performance. Modifying the situation awareness led to significant differences in performance, both anticipated and unanticipated. The pace and rate of these changes (with delays of 1-3 seconds) is both a useful range and a psychologically plausible range. More serious examinations of the effect of response time on situation awareness are possible. dTank thus appears to be able to support studies on the role of behavioral moderators on performance through modifications to situation awareness.

An additional use of dTank is as a training environment for cognitive modelers. Non-expert level programmers find the environment friendly, interesting, and highly usable—it has been used by third year undergraduates at Penn State and Lock Haven Universities in AI and modeling classes. dTank provides tools that support the analysis of scenarios and simulation data.

dTank supports communication between dTank agents, making it possible to create and test theories of social processes such as teamwork. It would also be possible to start to examine how teamwork and moderators and situation awareness interact and influence each other.

Although dTank is a promising and useful modeling environment for behavior representation, there are

many opportunities for improvements. A recent report by Sheppard and Baxter (2006) examining the effects of moderators on behavior suggest that simulations like dTank will have to pass more information to the agents. For example, currently the effects of suppressive fire cannot be modeled because agents only know when they are hit, not when someone near them is hit and makes a noise, or when someone firing at them makes a noise.

6. Acknowledgements

The development of this software was supported by ONR (contract N00014-06-1-0164). This work was also supported by the UK MOD's Analysis, Experimentation and Simulation corporate research programme (Project No: RT/COM/3/006). Comments from Harold Hawkins and Colin Sheppard have influenced this work; comments from three anonymous reviewers have improved this presentation of it.

7. References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review, 111*(4), 1036-1060.

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2005). Herbal: A high-level language and development environment for developing cognitive models in Soar. In *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation,* 133-140. 105-BRIMS-043. Orlando, FL: U. of Central Florida.

Councill, I. G., Haynes, S. R., & Ritter, F. E. (2003). Explaining Soar: Analysis of existing tools and user information requirements. In *Proceedings of the Fifth International Conference on Cognitive Modeling*, 63-68. Bamberg, Germany.

Endsley, M. R. (1988). Design and evaluation for situation awareness enhancement. In *Proceedings of the Human Factors Society 32nd Annual Meeting*, 97-101. Santa Monica, CA.

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.

Gluck, K. A., Gunzelmann, G., Gratch, J., Hudlicka, E., & Ritter, F. E. (2006). Modeling the impact of cognitive moderators on human cognition and performance. In *Proceedings of the 2006 Conference of the Cognitive Science Society,* 2658. Mahwah, NJ: Erlbaum.

Hudlicka, E., & Pfautz, J. (2002). Architecture and representation requirements for modeling effects of behavior moderators. In *Proceedings of the Eleventh Conference on Computer Generated Forces and Behavioral Representation,* 9-20. 02-CGF-085. Orlando, FL: Division of Continuing Education, University of Central Florida.

Lehman, J. F., Laird, J. E., & Rosenbloom, P. S. (1998). A gentle introduction to Soar: An architecture for human cognition. *An invitation to cognitive science*. D. Scarborough & S. Sternberg (eds.). Cambridge, MA: MIT Press.

Loral. (1995). *ModSAF software architecture design and overview document (SADOD)* (No. Report ADST/WDL/TR--95--W003339B). Orlando, FL: Prepared for U.S. Army Simulation, Training and Instrumentation Command (STRICOM).

Matthews, M. D., Pleban, R. J., Endsley, M. R., & Strater, L. D. (2000). Measures of infantry situation awareness for a virtual MOUT environment. In *Proceedings of the Human Performance, Situation Awareness and Automation: User Centered Design for the New Millennium Conference*.

Morgan, G. P., Ritter, F. E., Stevenson, W. E., Schenck, I. N., & Cohen, M. A. (2005). dTank: An environment for architectural comparisons of competitive agents. In *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation,* 133-140. 105-BRIMS-044. Orlando, FL.

Norling, E., & Ritter, F. E. (2004). A parameter set to support psychologically plausible variability in agent-based human modelling. In *The Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS04),* 758-765. New York, NY.

Ritter, F. E., Haynes, S. R., Cohen, M., Howes, A., John, B., Best, B., Lebiere, C., Jones, R. M., Crossman, J., Lewis, R. L., St. Amant, R., McBride, S. P., Urbas, L., Leuchter, S., & Vera, A. (2006). High-level behavior representation languages revisited. In *Proceedings of ICCM - 2006- Seventh International Conference on Cognitive Modeling.,* 404-407. Trieste, Italy: Edizioni Goliardiche.

Ritter, F. E., & Norling, E. (2006). Including human variability in a cognitive architecture to improve team simulation. In R. Sun (Ed.), *Cognition and multi-agent interaction: From cognitive modeling to social simulation* (pp. 417-427). Cambridge, UK: Cambridge University Press.

Ritter, F. E., Shadbolt, N. R., Elliman, D., Young, R. M., Gobet, F., & Baxter, G. D. (2003). *Techniques for modeling human performance in synthetic environments: A supplementary review*. Wright-Patterson Air Force Base, OH: Human Systems Information Analysis Center (HSIAC).

Sheppard, T., & Baxter, J. (2006). The Suitability of CoJACK for Modelling the Effects of Suppressive

Fire (No. RT/COM/3/006/2, Task 5, initial report): QinetiQ, Farnborough.
Silverman, B. G. (2004). Human performance simulation. In J. W. Ness, D. R. Ritzer & V. Tepe (Eds.), *The science and simulation of human performance* (pp. 469-498). Amsterdam: Elsevier.
Sun, S., Councill, I. G., Fan, X., Ritter, F. E., & Yen, J. (2004). Comparing teamwork modeling in an empirical approach. In *Proceedings of the Sixth International Conference on Cognitive Modeling,* 388-389. Mahwah, NJ: Erlbaum.
Yen, J., Yin, J., Ioerger, T. R., Miller, M. S., Xu, D., & Volz, R. A. (2001). CAST: Collaborative agents for simulating teamwork. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01),* 1135-1142. Los Altos, CA: Morgan Kaufmann.

Author Biographies

FRANK RITTER helped found the College of Information Sciences and Technology (IST), an interdisciplinary academic unit at Penn State University, to study how people process information using technology. He works on the development, application, and methodology of cognitive models, particularly as applied to interfaces and emotions. He is an editorial board member of *Human Factors* and *AISB Journal*.

SUE KASE is a research assistant in the Applied Cognitive Science Lab, College of IST. Her research involves implementing behavioral moderators in cognitive architectures and models of teams. Her Ph.D. thesis utilizes cognitive architectures and parallel genetic algorithms in high-performance computing environments to fit human behavior models of arithmetic performance under stressful conditions.

BIL LEWIS teaches Computer Science at Tufts University and does research on Electronic Voting at the MIT Media Lab. His primary work is in debugging backwards in time. (He is the creator of the Omniscient Debugger, www.lambdacs.com/debugger/debugger.html.) He is author of three books on multithreaded programming along with the GNU Emacs Lisp Manual.

DAMODAR BHANDARKAR is a research assistant in the Applied Cognitive Science Lab, College of IST. He is currently completing his Ph.D. in Industrial Engineering, with a focus in Human Factors. His current research involves modeling human performance in complex environments. He helps maintain dTank and the dTank API's, and creates models in various architectures to work with dTank.

MARK COHEN is an assistant professor in the Business Administration, CS and IT Department at Lock Haven University, and a graduate student associated with the Applied Cognitive Science Lab in the College of IST at Penn State. His current research efforts include developing software that simplifies the creation and maintenance of cognitive models.