Stratagus: An Open-Source Game Engine for Research in Real-Time Strategy Games

Marc J.V. Ponsen[1], Stephen Lee-Urban[1], Héctor Muñoz-Avila[1], David W. Aha[2], Matthew Molineaux[3]

[1]Dept. of Computer Science & Engineering; Lehigh University; Bethlehem, PA; USA; {mjp304, sml3, hem4}@lehigh.edu
[2]Navy Center for Applied Research in AI; Naval Research Laboratory (Code 5515); Washington, DC; USA; aha@aic.nrl.navy.mil
[3]ITT Industries; AES Division; Alexandria, VA 22303 molineau@aic.nrl.navy.mil

Abstract

This paper advocates using Stratagus, an open-source real-time strategy game engine, in artificial intelligence research on real-time strategy games. We also describe its integration with TIELT, a testbed for integrating and evaluating decision systems with game engines. Together they can be used to study approaches for reasoning, representation, and learning in computer games.

1. Introduction

In recent years, AI researchers (e.g., Laird & van Lent, 2001; Guestrin *et al.,* 2003; Buro, 2003; Spronck *et al,.* 2004; Ponsen *et al.,* 2005) have begun focusing on artificial intelligence (AI) challenges presented by complex computer games. Among these, real-time strategy (RTS) games offer a unique set of AI challenges such as planning under uncertainty, learning adversarial strategies, and analyzing partially observable terrain. However, many commercial RTS games are closed-source, complicating their integration with AI decision systems. Therefore, academic practitioners often develop their own gaming engines (e.g., Buro's (2002) ORTS) or turn to open-source alternatives (e.g., FreeCiv, Stratagus).

Stratagus is an appropriate RTS engine for AI research for several reasons: it is open-source, the engine is highly configurable, and it is integrated with TIELT (2005), which can be used to integrate AI decision systems with gaming simulators and help evaluate how well those systems perform on selected tasks. In this paper, we first describe RTS games and the research challenges posed by these environments. In Section 3 we then detail the Stratagus engine, and describe its integration with TIELT in Section 4. In Section 5 we describe an example application of their integration. Finally, we conclude and highlight future work in Section 6.

2. Research Challenges in RTS Games

RTS is a genre of strategy games that usually focuses on military combat. RTS games such as Warcraft™ and Empire Earth™ require the player to control a civilization and use military force to defeat all opposing civilizations that are situated in a virtual battlefield (often called a *map*) in real time. In most RTS games, winning requires efficiently collecting and managing resources, and appropriately distributing these resources over the various game action elements. Typical RTS game actions include constructing buildings, researching new technologies, and waging combat with armies (i.e., consisting of different types of units).

The game AI in RTS manages the decision-making process of computer-controlled opponents. We distinguish between two levels of decision-making in RTS games. The *strategic* level involves abstract decision making considerations. In contrast, the *tactical* level concerns more concrete decisions. The *global AI* in RTS games is responsible for making strategic decisions (e.g., choosing warfare objectives or deciding the size of the force necessary to assault and control an enemy city). The *local AI* works on the game action level (e.g., give individual commands to train soldiers, and give them orders to move to specific coordinates near the enemy base) and is responsible for achieving the objectives defined at the tactical level.

Unlike the high performance in some classic board games where AI players are among the best in the world (Schaeffer, 2001), their performance in RTS games is comparatively poor (Buro, 2003) because designing strong AI in these complex environments is a challenging task. RTS games include only partially observable environments that contain adversaries who modify the game state asynchronously, and whose decision models are unknown, thereby making it infeasible to obtain complete information on the current game situation. In addition, RTS games include an enormous number of possible game actions that can be executed at any given time, and some of their effects on the game state are uncertain. Also, to successfully play an RTS game, players must make their decisions in real-time (i.e., under severe time constraints) and execute multiple orders in a short time span. These properties of RTS games make them a challenging domain for AI research.

Because of their decision making complexity, previous research with RTS games often resorted to simpler tasks (Guestrin *et al*., 2003), decision systems were applied offline (e.g., the evolutionary algorithm developed by Ponsen and Spronck (2004)), or they employed an

Figure 1: Screenshots of two Stratagus games. The left shows Wargus, a clone of Blizzard's Warcraft II™ game, and the right shows Magnant. The Wargus game is situated in a fantasy world where players either control armies of humans or 'orcs'. Magnant combines an RTS game with a trading card game. In Magnant, players can collect, exchange, and trade cards that give them unique abilities.

abstraction of the state and decision space (e.g., Madeira *et al.*, 2004; Ponsen *et al.*, 2005; Aha *et al*., 2005).

When employing online decision-making in RTS games, it is computationally impractical for an AI decision system to plan on the game action level when addressing more comprehensive tasks (e.g., making all strategic and tactical decisions for a civilization). Decision systems can potentially plan more quickly and efficiently when employing appropriate abstractions for state and decision space. However, current AI systems lag behind human abilities to abstract, generalize, and plan in complex domains. Although some consider abstraction to be the essence of intelligence (Brooks, 1991), comparatively little research has focused on abstraction in the context of complex computer games. However, Ponsen and Spronck (2004) developed a lattice for representing and relating abstract states in RTS games. They identified states with the set of buildings a player possesses. Buildings are essential elements in many RTS games because these typically allow a player to train armies and acquire new technologies (which are relevant game actions in RTS games such as Warcraft™). Thus, they determine what tactics can be executed during that game state. Similarly, Ponsen et al. (2005) and Aha et al. (2005) used an abstraction of the decision space by searching in the space of compound tactics (i.e., fine-tuned combination of game actions) that were acquired offline with an evolutionary algorithm.

Another limitation of commercial game AI is its inability to learn and adapt quickly to changing players and situations. In recent years, the topic of learning in computer games has grown in interest and several such studies have focused on RTS. For instance, Guestrin et al. (2003) applied relational Markov decision process models for some limited combat scenarios (e.g., 3x3 combat). In contrast, by using abstract state and decision spaces Ponsen and Spronck (2004) and Ponsen et al. (2005) applied *dynamic scripting* (Spronck *et al*., 2004) to learn to win complete games against a static opponent. Aha et al. (2005) relaxed the assumption of a fixed adversary by extending the state representation and recording game performance measures from selecting specific tactics in these extended states. This permitted them to evaluate a case-based tactic selector (CaT) designed to win against random opponents.

3. Stratagus: An Open-Source RTS Engine

Stratagus is a cross-platform, open-source gaming engine for building RTS games. It supports both single player (i.e., playing against a computer opponent) and multi player games (i.e., playing over the internet or LAN). Stratagus is implemented primarily in C. A precompiled version can be downloaded freely at the Stratagus website (http://stratagus.sourceforge.net). The latest source code of this actively maintained engine can be downloaded from a CVS (Concurrent Versioning System) repository.

Games that use the Stratagus engine are implemented in scripts using the LUA scripting language (www.lua.org). These games can also be found on the Stratagus website. Popular games for Stratagus include Wargus (a clone of the popular RTS game Warcraft II™, illustrated in Figure 1), Battle of Survival (a futuristic real-time strategy game), and Magnant (a trading card game situated in an RTS environment, also illustrated in Figure 1). These games all share the same API provided by the game engine.

Stratagus includes some useful features for AI research:

- *Configurable*: This highly configurable engine can be used to create RTS games with varying features. Therefore, games can be tailored to specific AI tasks (e.g., learning to win complete games, winning local battles, resource management, pathfinding).
- *Games*: Stratagus already includes several operational games (e.g., see Figure 1). Because all games share the same API, a decision system can be evaluated in multiple domains.
- *Modifiable*: Although Stratagus is fairly mature code, changing existing code in the engine can be a daunting task, in particular for novice C programmers. Fortunately, LUA allows AI researchers to modify the game AI without having to change the engine code. LUA employs many familiar programming paradigms from 'common' programming languages such as C (e.g., variables, statements, functions), but in a simpler fashion. Also, LUA is well documented; many detailed tutorials and example snippets can be found on-line (e.g., see http://lua-users.org/wiki/).
- *Fast mode*: Stratagus includes a fast forward mode where graphics are partially turned off, resulting in fast games. This is particularly useful for expediting experiments.
- *Statistics*: During and after Stratagus games, numerous game related data (e.g., time elapsed before winning, the number of killed units, the number of units lost) are available, and are particularly useful for constructing a performance measure used by machine learning algorithms.
- *Recordable*: Games can be recorded and replayed. Recorded games can be used for training AI systems to perform tasks such as plan recognition and strategic planning.
- *Map Editor*: Stratagus includes a straightforward (random) map editor. This facility can be used to vary the initial state when evaluating the utility of AI problem solving systems.

Another important feature of Stratagus is that it is integrated with TIELT. We describe TIELT, its integration with Stratagus, and an example application in Sections 4 and 5.

4. TIELT Integration

Integrating AI systems with most closed-source commercial gaming simulators such as Stratagus can be a difficult or even impossible task. Also, the resulting interface may not be reusable for similar integrations that a researcher may want to develop. In this section we describe an integration of Stratagus with TIELT (**T**estbed for **I**ntegrating and **E**valuating **L**earning **T**echniques) (TIELT, 2005; Aha & Molineaux, 2004). TIELT is a freely available tool that facilitates the integration of decision systems and simulators. To date, its focus has been on supporting the integration of machine learning systems and complex gaming simulators.

Decision systems integrated with TIELT can, among other tasks, take the role of the human player in a Stratagus game (i.e., they can play a complete game in place of the human player, or assist a human player with a particular task). TIELT-integrated decision systems, in Buro's (2003) terminology, can be viewed as "AI plug-ins". For example, they can assist the human player with managing resources or take control of the human player's fleet. These decision systems can also be used as advisors. For example, based on TIELT's observations of the opponent's army, its decision system could advise the human player to attack with an army of x footmen, y archers, and z catapults. TIELT could prompt the human player to determine whether its decision system should execute this order. If the user agrees, it will then execute the order (i.e., train the army and send it into battle).

4.1 TIELT Knowledge Bases

TIELT's integration with Stratagus requires constructing or reusing/adapting five knowledge bases. We describe each of them briefly here.

The *Game Model* is a (usually partial) declarative representation of the game. TIELT's Game Model for this integration includes operators for key game tasks like building armies, researching technology advances, and attacking. Other operators obtain information about the game (e.g., the player's score). It also contains information about key events such as when a building or unit is completed. This information is kept updated by the Game Interface Model.

The *Game Interface Model* and *Decision System Interface Model* define the format and content of messages passed between TIELT, the selected game engine, and the selected decision system. Future research using Stratagus can reuse these models. We will describe an example decision system in Section 5. Two-way communication with the Game Interface Model is achieved over a TCP/IP connection and includes both actions and sensors (see Table 1). Actions are commands send from TIELT to Stratagus, and are used to either control the game AI or change settings in the game engine. In contrast, sensors are received by TIELT from Stratagus and give information on the game state or current engine settings.

Currently, TIELT's game actions interface with the global AI in Stratagus. By sending strategic commands to the Stratagus engine, TIELT does not have to plan on the local AI level. The latter is currently hard-coded in the engine and is automatically triggered through global AI actions. For example, suppose the AI wants to launch an attack on an enemy. This can be achieved by first specifying the number and type of the units belonging to

Table 1: Description of the most important, currently available actions (sent from TIELT to Stratagus) and sensors (sent from Stratagus to TIELT).

Game AI Actions	
AiAttackWithForce (forceID) *e.g., AiAttackWithForce(1)*	Command the AI to attack an enemy with all units belonging to a predefined force.
AiCheckForce (ForceID) *e.g., AiCheckForce(1)*	Check if a force is ready.
AiForce (ForceID, {force}) *e.g., AiForce(1,{"unit-grunt", 3})*	Define a force: determine the type and number of units that belong to it.
AiForceRole (forceID, role) *e.g.,AiForceRole(1,"defend")*	Define the role of a force: Assign it either an defensive or offensive role.
AiNeed (unitType) *e.g., AiNeed("unit-footmen")*	Command the AI to train or build a unit of a specific unit type (e.g., request the training of a soldier).
AiMoveTo (forceID, location) *e.g., AiMoveTo(1,200,200)*	Command an army to move to a specific location on the map.
AiSendMessage (message) *e.g., AiSendMessage ("Some text")*	Send a message that will be displayed in the game console.
AiResearch (researchType) *e.g., AiResearch("upgrade-sword")*	Command the AI to pursue a specific research advancement.
AiUpgradeTo (unitType) *e.g.,AiUpgradeTo("upgrade-ranger")*	Command the AI to upgrade a specific unit.
Miscellaneous Actions	
GameCycle ()	Request the current game cycle.
GetPlayerData (playerID, info) *e.g., GetPlayerData(1,"Score")*	Request player information (e.g., get the player score or the number of units or buildings controlled by this player).
QuitGame ()	Quit a Stratagus game.
SetFastForwardCycle(cycle) *e.g., SetFastFowardCycle(200000)*	Set the fast forward cycle (only for single player games).
UseStrategy(strategyID) *e.g., UseStrategy(2)*	Set the strategy for a specific opponent.
Sensors	
CONNECT ()	Informs TIELT that a TCP connection has been established with Stratagus.
FORCE (ForceID) *e.g., FORCE(1)*	Informs TIELT that all units for a specific army are combat-ready .
GAMECYCLE (gameCycle) *e.g., GameCycle(20000)*	Sent in response to the GameCycle action. It returns the current game cycle.
GAMEOVER (result, TieltScore, endCycle) *e.g.,GAMEOVER("won",1000,500)*	Informs TIELT that the game has ended. It reports information on the result of the game.
INFO (playerID, message) *e.g., INFO(1,"New Peasant ready")*	Informs TIELT about game events (e.g., 'under attack', 'building or unit is complete', or 'a weapon upgrade has been acquired').
PLAYERDATA (playerID, info, content) *e.g., PLAYERDATA (1,TotalKills, 10)*	Sent in response to the GetPlayerData action. It returns information for a specific player.
SELECTOPPONENT (playerID) *e.g., SELECTOPPONENT(1)*	Used during single player game setup, it allows TIELT to select the strategy for this opponent in this specific game.

the attack force and then ordering it to attack the enemy. Local AI tasks such as training the individual units, target selection, and pathfinding to an enemy base is currently hard-coded in the engine. Another typical global AI command is to construct a particular building. Deciding the best place to construct the building and deciding which worker will be assigned to the task is left to the engine.

Next, the Agent *Description* includes an executable task structure that distinguishes the responsibilities of the decision system from those of game engine components. The Agent Description links a decision system to the abstractions of the Game Model. TIELT retrieves instructions from a decision system and executes them using operators. When events recognized by the Game Model occur, it notifies the decision system, using information from the Decision System Interface Model to communicate. For example, when a building is finished, Stratagus sends a message to TIELT. The Game Interface Model interprets this, and fires a Game Model event. The Agent Description, listening for this event, notifies the decision system and asks for further instructions. This Agent Description would need to be rewritten to work with a different decision system (if it targets a different performance task), but the abstractions available from the Game Model simplify creating this knowledge base.

Finally, the *Experiment Methodology* encodes how the user wants to test the decision system on selected game engine tasks. For example, it defines the number of runs, when to stop the game, and resets the decision system's memory when experiments begin. Also, it records game data for post-experiment analysis. It also permits a user to repeat experiments overnight, and record any data passed from Stratagus to the decision system, or vice versa.

4.2 Single and Multiplayer Games

Stratagus includes both single player and multiplayer (i.e., network) games. The communication between Stratagus and TIELT for both game types is illustrated in Figure 2.

In a single player game, the 'TIELT AI' will be pitted against one or more static or adaptive (Ponsen and Spronck, 2004) opponent AIs controlled by scripts, which are lists of game actions that are executed sequentially (Tozour, 2002). In single player games, it is possible to run games in a fast forward mode. This speeds up games enormously (e.g., a typical battle between two civilizations on a relatively small map finishes in less than 3 minutes, whereas in normal speed a game can take up to 30 minutes).

A multiplayer game, which is played by multiple players over the Internet or a LAN, can be used to play with multiple different decision systems for TIELT and/or human players. Multiplayer games cannot be run in fast forward mode due to network synchronization issues.

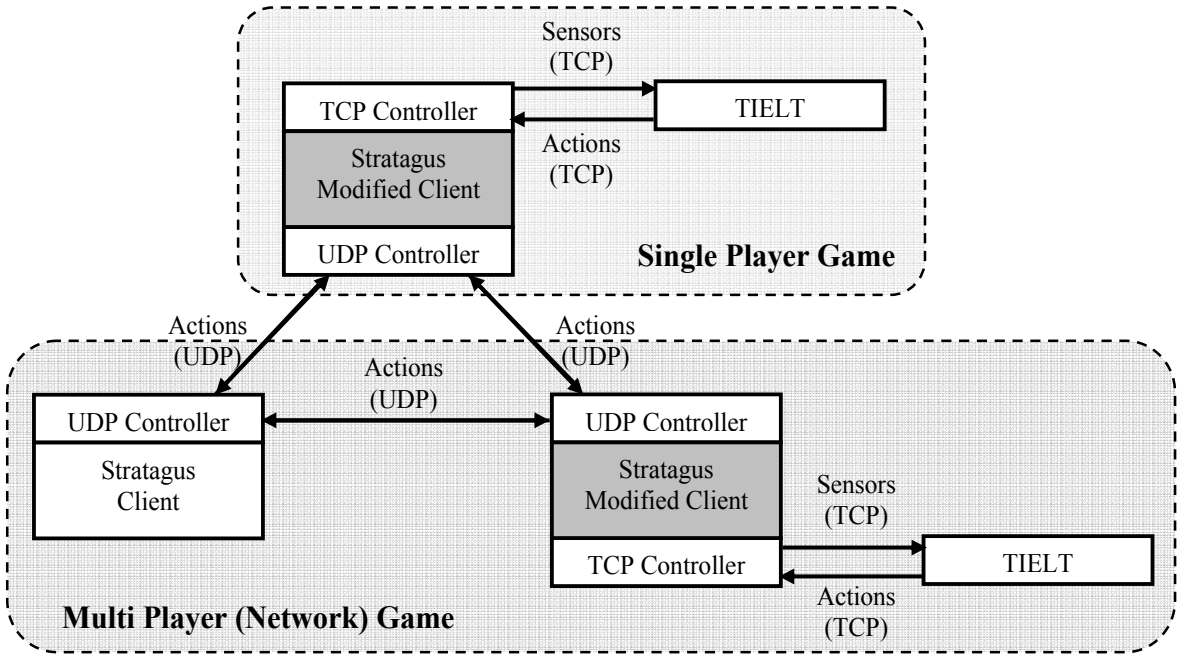

Figure 2: Schematic view of the communication between Stratagus clients, modified clients (i.e., those with TIELT integration), and TIELT.

5. Example Application

In this section, we describe an example application of our Stratagus-TIELT integration. We implemented a decision system, named *SR*, that employs a 'soldier's rush' strategy in the Wargus domain, and integrated it with TIELT. This strategy attempts to overwhelm the opponent with cheap military units in an early state of the game. In Wargus, our soldier's rush implementation requires the AI to first train (i.e., build) a small defensive force, then build several barracks followed by a blacksmith. After constructing a blacksmith, the AI will upgrade its soldiers by researching better weapons and shields. This strategy will then continuously attack the enemy with large groups of soldiers.

In the initial state, the AI starts with a town hall and barracks. SR will first command the AI to train a small defensive force. This command is divided into two parts. First, an army will be trained with the Game AI action *AiForce*. Next, this army will be assigned a defensive role using the action *AiForceRole*. SR, through TIELT, can command Stratagus to construct several barracks and a blacksmith using the action *AiNeed*. The planned research advancements (better weapons and shields) can be achieved using the action *AiResearch*. However, these research advancements are only applicable when a blacksmith has been fully constructed. Therefore, before triggering these actions, TIELT will first wait for an INFO sensor informing it that a blacksmith has been constructed. When the soldiers are upgraded, SR can order Stratagus to train an army consisting of *x* soldiers that will be used to attack the enemy. Again, the force is defined using the action *AiForce*. Before sending the army into battle, SR will first wait until the army is complete. SR periodically triggers the action *AiCheckForce*. If the desired *x* number of soldiers has been trained, Stratagus will send a FORCE sensor (in response to the *AiCheckForce* action). After receiving the FORCE sensor, SR can launch the attack with the Game AI action *AiAttackWithForce*. The actions for the training and use of an offensive army can be repeated by SR to continuously launch attacks on the enemy.

After a game has finished, Stratagus will send a GAMEOVER sensor message. This sensor informs TIELT and SR on the result of the game (e.g., a win or loss). In return, TIELT can collect more detailed game-related data by sending the *GameCycle* and/or *GetPlayerData* actions. The former returns the current game cycle (in this case, the time it took before the game was won by either player) with the GAMECYCLE sensor, while the latter returns player information (e.g., player score, units killed by player, units lost by player) with the PLAYERDATA sensor.

The SR example decision system, Stratagus' necessary knowledge base files for TIELT, and the modified Stratagus engine can be downloaded at http://nrlsat.ittid.com and http://www.cse.lehigh.edu.

Note to reviewers: these files are currently not available, but they will be made available prior to the workshop (if the paper is accepted).

6. Conclusions and Future Work

Currently, a TIELT-integrated decision system can control the game AI in Stratagus in both single and multiplayer games. This integration was used for experiments in single player games (Aha *et al.*, 2005). The integrated decision system has complete control over the global AI in Stratagus games through a set of TIELT actions that interface with the game engine's API. It also receives feedback from the game engine through a set of TIELT sensors.

In our future work we will improve the control a TIELT-integrated decision system has over the AI by adding more game actions. For example, we will also allow TIELT to interface with the local AI. However, we expect that the API will always be subject to change because different decision systems require different APIs. Because Stratagus is open-source, researchers can change or extend the API so that it meets their demands. We will also extend the sensors and provide TIELT with more detailed information on the game state. Greater interaction between human Stratagus players and TIELT-integrated decision systems can be aided by the addition of a TIELT specific "toolbox" panel to the games. Such a toolbox would visually allow the human player to have a dialog with TIELT (e.g., accepting the advice of the TIELT interfaced decision system to attack a particular target, or asking the system to find an ideal path for an army through varied terrain). Finally, in addition to the example application described in Section 5, we will design and evaluate other decision systems with TIELT, and explore issues such as on-line learning, collaborative decision systems, and methods for automatically learning domain knowledge.

Acknowledgements

This research was sponsored by a grant from DARPA and the Naval Research Laboratory. Many thanks to the Stratagus developers for their support.

References

Aha, D.W., & Molineaux, M. (2004). Integrating learning in interactive gaming simulators. In D. Fu & J. Orkin (Eds.) *Challenges in Game AI: Papers of the AAAI'04 Workshop* (Technical Report WS-04-04). San José, CA: AAAI Press.

Aha, D.W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. *To appear in Proceedings of the Sixth International Conference on Case-Based Reasoning.* Chicago, IL: Springer.

Brooks, R.A. (1991). Intelligence without representation. *Artificial Intelligence*, **47**, 139-159.

Buro, M. (2002). ORTS: A hack-free RTS game environment. *Proceedings of the International Computers and Games Conference.* Edmonton, Canada: Springer.

Buro, M. (2003). Real-time strategy games: A new AI research challenge. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence* (pp. 1534-1535). Acapulco, Mexico: Morgan Kaufmann.

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. *Proceedings of the Eighteenth International Joint Conference in Artificial Intelligence* (pp. 1003-1010). Acapulco, Mexico: Morgan Kaufmann.

Laird, J.E., & van Lent, M. (2001). Interactive computer games: Human-level AI's killer application. *AI Magazine*, **22**(2), 15-25.

Madeira, C., Corruble, V. Ramalho, G., & Ratich B. (2004). Bootstrapping the learning process for the semi-automated design of challenging game AI. In D. Fu & J. Orkin (Eds.) *Challenges in Game AI: Papers of the AAAI'04 Workshop* (Technical Report WS-04-04). San José, CA: AAAI Press.

Ponsen, M., Muñoz-Avila, H., Spronck P., & Aha D.W (2005). Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning. To appear in *Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence*. Pittsburgh, PA: Morgan Kaufmann.

Ponsen, M., & Spronck, P. (2004). Improving adaptive game AI with evolutionary learning. *Computer Games: Artificial Intelligence, Design and Education* (pp. 389-396). Reading, UK: University of Wolverhampton Press.

Schaeffer, J. (2001). A gamut of games. *AI Magazine*, **22**(3), 29–46.

Spronck, P., Sprinkhuizen-Kuyper, I., & Postma, E. (2004). Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation,* **3**(1), 45–53.

TIELT (2005). Testbed for integrating and evaluating learning techniques. [http://nrlsat.ittid.com]

Tozour, P. (2002). The perils of AI scripting. In S. Rabin (Ed.) *AI Game Programming Wisdom*. Hingham, MA: Charles River Media.