# Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games

(http://home.earthlink.net/~dwaha/research/meetings/ijcai05-rrlcgw)

David W. Aha, Héctor Muñoz-Avila, & Michael van Lent (Eds.)

Edinburgh, Scotland
31 July 2005

## Workshop Committee



Aha, D.W., Muñoz-Avila, H., & van Lent, M. (Eds.) (2005). *Reasoning, Representation, and Learning in Computer Games*: *Proceedings of the IJCAI Workshop* (Technical Report AIC-05). Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

# Preface

These proceedings contain the papers presented at the Workshop on *Reasoning, Representation, and Learning in Computer Games* held at the 2005 International Joint Conference on Artificial Intelligence (IJCAI'05) in Edinburgh, Scotland on 31 July 2005.

Our objective for holding this workshop was to encourage the study, development, integration, and evaluation of AI techniques on tasks from complex games. These challenging performance tasks are characterized by huge search spaces, uncertainty, opportunities for coordination/teaming, and (frequently) multi-agent adversarial conditions. We wanted to foster a dialogue among researchers in a variety of AI disciplines who seek to develop and test their theories on comprehensive intelligent agents that can function competently in virtual gaming worlds. We expected that this workshop would yield an understanding of (1) state-of-the-art approaches for performing well in complex gaming environments and (2) research issues that require additional attention.

Games-related research extends back to the origins of artificial intelligence, which includes Turing's proposed imitation game and Arthur Samuel's work on checkers. Several notable achievements have been attained for the games of checkers, reversi, Scrabble, backgammon, and chess, among several others. Several AI journals are devoted to this topic, such as the *International Computer Games Association Journal*, the *International Journal of Intelligent Games and Simulation*, and the *Journal of Game Development*. Similarly, many conferences are likewise devoted to this topic, including the *International Conference on Computers and Games*, the *European Conference on Simulation and AI in Computer Games*, and the new *Conference on Artificial Intelligence and Interactive Digital Entertainment.* Naturally, IJCAI has also hosted workshops on AI research and games, including *Entertainment and AI/Alife* (IJCAI'95), *Using Games as an Experimental Testbed for AI Research* (IJCAI'97), and *RoboCup* (IJCAI'97, IJCAI'99).

In contrast to previous IJCAI workshops on AI and games, this one has a relatively broad scope; it was not focused on a specific sub-topic (e.g., testbeds), game or game genre, or AI reasoning paradigm. Rather, we focused on topics of general interest to AI researchers (i.e., reasoning, representation, learning) to which many different types of AI approaches could apply. Thus, this workshop provided an opportunity to share and learn from a wide variety of research perspectives, which is not feasible for meetings held at conferences on AI sub-disciplines.

Therefore, our agenda was quite broad. Our invited speakers included Ian Davis, who discussed applied research at Mad Doc Software, and Michael Genesereth/Nat Love, who reported on the first annual General Game Playing Competition (at AAAI'05) and their future plans. In addition to sessions of presentations and discussion periods on the workshop's three themes (i.e., reasoning, representation, and learning), our fourth session focused on AI architectures. We also held a sizable poster session (i.e., perhaps we should have scheduled this as a 2-day event) and a wrap-up panel that generated visions for future research and development, including feasible and productive suggestions for dissertation topics.

The Workshop Committee did a great job in providing suggestions and informative reviews for the submissions; thank you! Thanks also to Carlos Guestrin, ICCBR'05 Workshop Chair, for his assistance in helping us to hold and schedule this workshop. Finally, thanks to all the participants; we hope you found this to be useful!

David W. Aha, Héctor Muñoz-Avila, & Michael van Lent
Edinburgh, Scotland
31 July 2005

# Table of Contents

# Hazard: A Framework Towards Connecting Artificial Intelligence and Robotics

**Peter J. Andersson**
Department of Computer and Information Science, Linköping university
petan@ida.liu.se

## Abstract

The gaming industry has started to look for solutions in the Artificial intelligence (AI) research community and work has begun with common standards for integration. At the same time, few robotic systems in development use already developed AI frameworks and technologies. In this article, we present the development and evaluation of the Hazard framework that has been used to rapidly create simulations for development of cognitive systems. Implementations include for example a dialogue system that transparently can connect to either an Unmanned Aerial Vehicle (UAV) or a simulated counterpart. Hazard is found suitable for developing simulations supporting high-level AI development and we identify and propose a solution to the factors that make the framework unsuitable for lower level robotic specific tasks such as event/chronicle recognition.

## 1 Introduction

When developing or testing techniques for artificial intelligence, it is common to use a simulation. The simulation should as completely as possible simulate the environment that the AI will encounter when deployed live. Sometimes, it is the only environment in which the AI will be deployed (as is the case with computer games). As there exist many more types of environments than AI techniques, there is a need to reuse existing implementations of AI techniques with new environments. AI frameworks often come bundled with an environment to test the AI technique against and the environments are often not as easy to modify as a developer would want, nor is it easy to evaluate the framework in new environments without developing time-consuming middleware. In the case of robotics, there is an extended development period to develop low-level control programs. The AI then developed is often tailored to the low-level control programs and it is hard, if at all possible to reuse.

The Player-Stage project [Gerkey *et al.*, 2003] develops low-level drivers for robotic architectures and gives the developer an interface that can either be used together with the stand-alone simulator to evaluate the configuration or to control the actual robotic hardware. This kind of interface encourages focus on the high-level control of the robot, but since there are no wrappers to high-level AI frameworks, it does not encourage reuse of existing AI techniques. By developing a high-level interface between Player-Stage and AI frameworks, we will also allow AI researchers to take advantage of the Player-Stage project.

The Robocup initiative [Kitano *et al.*, 1997] uses both actual robotic hardware and simulation in competition. Yet, there exists no common interface for using simulation league AIs with robotic league robots. This can mean that the simulation interface is unintuitive for actual robotics, or that AIs developed with the simulation are not usable with actual robots. In either case it is a problem worth investigating.

The WITAS Unmanned Aerial Vehicle project [Doherty *et al.*, 2000] uses several simulators in their research, both for hardware-in-the-loop simulation of the helicopter hardware and for development of dialogue interaction with an actual UAV. A middleware translating actions and events from WITAS protocol to other protocols would allow experimentation with for example SOAR [Laird *et al.*, 1987] as a high-level decision system. It would also allow the application of developed agent architecture to other environments and problems.

By creating a middleware framework that can mediate between low-level drivers and high-level decision system, we hope to be able to alleviate the problems for AI researchers presented above and inspire researchers in both artificial intelligence and robotics to reuse existing implementations. Robotic researchers can reuse AI techniques that exist and AI researchers can test their AI implementation in off-the-shelf simulators before building an expensive robotic system. It would also allow separate research groups to work with low-level control and high-level decision issues.

As a step towards proposing an interface and middleware for connecting AI and robotics, we have designed and implemented a framework for developing agents and environments where we focus on the distinction between agent and environment. The design iterations and various implementations with the framework have allowed us to gain valuable experience in designing both interface and framework. The work is presented here together with an evaluation of the strengths, weaknesses and limitations.

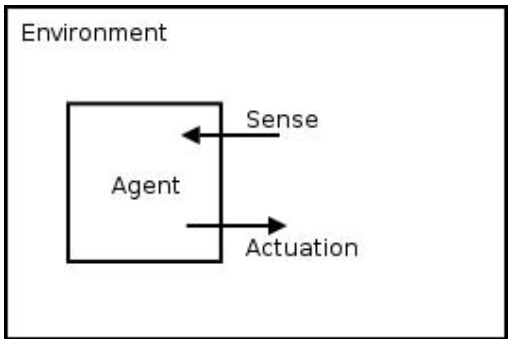


Figure 1: Agents and Environments.

## 2 Interface

The framework is designed around action theory and based on the agent-environment concept as found in [Russel and Norvig, 1995], see figure 1. The framework is thus divided into three parts; interface layer, agent module and environment module. The agent uses actuators to communicate its intention to the environment, the environment delivers the sensor impressions back to the agent. In this interface the actuation is handled by actions. Each action and sensor has an owning agent to which all data is passed. The interface encourages multi-threaded applications where the agent decision loop is in one thread and the environment is in another. This design has been helpful when building continuous, asynchronous environments such as simulations of robotic vehicles.

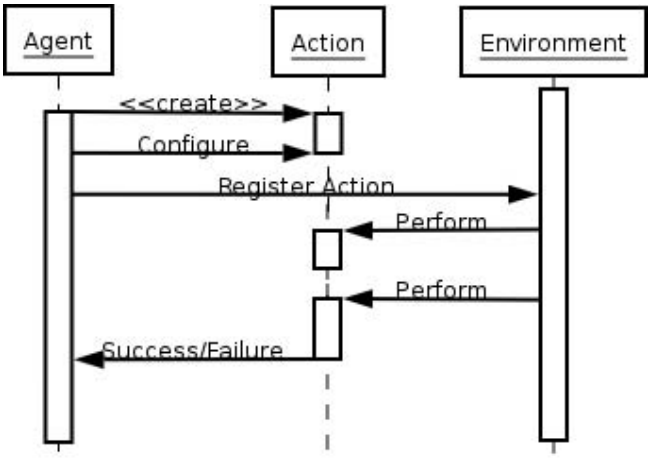


Figure 2: How actions are used.

Actions
: When an agent has decided on using a certain action, it configures the action and notifies the environment that it wants to "register" the action. If the action is applicable in the current state of the environment, it is accepted and execution starts. The action contains its own executable code and is thus defined in the environment module. Actions are bundled with their own event recognition and execution monitoring. At each execution interval, they update the state of their owner in the environment. If applicable, they can send "checkpoint" reports back to the owner. When the action has reached its end criteria, it sends a "success" message. If it fails during execution, it sends a "fail" message. If it is an action to which success/fail has no meaning (for actions without an explicit goal), the action can send a "stop" message. All these messages are implemented as callback methods in the agent interface. The implementation is visualized in the sequence diagram in figure 2. Since actions are implemented in the environment, the agent must have knowledge of the environment to be able to use the actions. Actions are considered to be executed in continuous time, can be concurrent and are either discrete or durative regarding the end criteria.

Sensors
: Sensors are permanent non-invasive actions that can be registered in the environment. Sensors contain their own executable code and should not modify the environment. A sensor analyzes the environment at a preset time-interval and stores the extracted information. When it has new data, the sensor can notify the owning agent via a callback routine. The agent can then fetch the new sensor data from the sensor via the sensor interface. What kind of data that is passed between sensor and agent is not defined by the interface but implemented on a case by case basis.

## 3 Framework

Together with the interface a framework has been designed. The framework has been named Hazard and helps developers to rapidly create new environments and agents. It can also be used to design wrappers for other environments or agents, thus allowing other implementations using Hazard to be quickly modified to use that agent implementation or environment. The design has been developed iteratively with several simulations as reference implementations. The experience from developing each simulator has given a simpler, more robust and better designed framework ready for the next project.

The goal of the framework has been both to allow development of new environments and agents, and to use it as a middleware between already existing frameworks. The design focus has been on forcing developers to make important design decisions at an early stage of the project and to leave simpler issues in the hands of the framework.

### 3.1 Environments

An overview of the environment module can be found in figure 3 and described in more detail below.

Environment
: An environment in Hazard has the task of keeping track of Maps and execution of actions and sensors. The environment can contain several maps, dividing computation, and gives a more scalable structure.

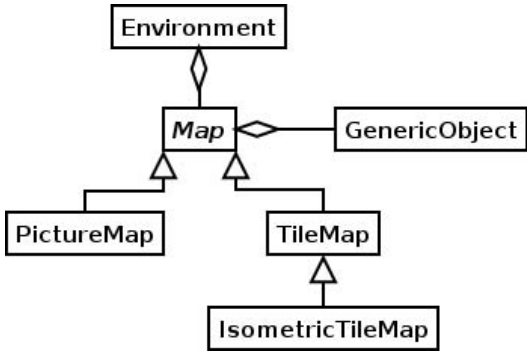


Figure 3: General structure of the environment module.

Map
: A map in Hazard can contain a number of objects and agents, where agents are a subset of the objects. The framework has a number of default types of maps that can be used by developers. Each of these maps is data-driven and can be inherited or used as is.

GenericObject
: GenericObject is the template implementation of all objects in the environment. For most objects, the GenericObject class can be used as is. For specific needs, the class can be inherited to create unique objects specific to a certain environment.

The environment module also implements a basic graphics engine that can handle two dimensional graphics.

### 3.2 Agents

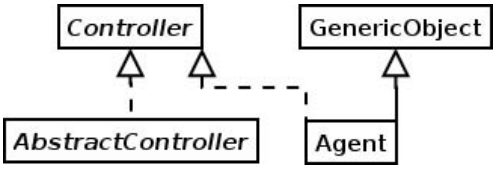


Figure 4: General structure of the agent module.

Agents are controlled by either AI or a user. To generalize, both cases implement the interface "Controller". A controller can control one or more objects in the environment, effectively turning them into agents. Controllers implement methods for registering/unregistering actions and sensors in the environment. They implement the callbacks for actions and sensors, but have otherwise no limitations. A GenericObject that implements the Controller interface is called an agent. The Agent module also contains classes and methods for class reuse, action and sensor design and abstract classes for different types of controllers.

## 4 Iterations of Design and Implementation

The strength of a design proposal may be measured in term of its development history. Designs that have undergone several iterations under different conditions are much more mature than designs without practical grounding. The Hazard framework was derived from the Haphazard Role-Playing Game project. The derivation was redesigned and reimplemented to form the first version of the framework. The Haphazard project was redesigned to use the new framework and experience from that work was used to implement the traffic simulator. The framework was redesigned again, and was used to implement a development support simulator for a dialogue system, leading to the present Hazard framework. The Haphazard Online Role-Playing Game and the Simulator for Support of Dialogue System Development are currently using the latest version of the Hazard framework while the Subsumption Simulator is using an older version.

### 4.1 Haphazard - An Online Role-Playing Game

The Haphazard Role-Playing Game [Andersson and Beskid, 2003] started as an open-source project for creating a simple online role-playing game with focus on AI implementations. It was from this game project that the first version of Hazard was extracted. The framework was then redesigned and Haphazard was reimplemented to use the new framework. Haphazard has the most complex environment of all implementations up to date, but only rudimentary AI. The Haphazard project was the most prominent project when designing the framework for environments.

#### Environment

The environment in Haphazard is a grid-based model using tiles for visualisation. Objects in the world are either static or dynamic. Static objects include houses, trees and other scenery. Dynamic objects are the objects that can be manipulated by the player. Haphazard includes a novel inventory system, environment specific enhancements to the Hazard framework that include minor agent interaction and a dynamic environment system. Players can roam between different maps which allows for a distributed environment. The environment is visualized by an isometric tile-based graphics engine. The environment is data-driven. Both environment and graphics can be changed at run-time.

#### Agents

The main focus is on the agent that allows the user to interact with the environment. It implements a graphical user interface which allows the user to have direct control of the agent. The user interface implements movement control, skill management and inventory management. The agent does not have complete vision of the world, but has a small field of vision that is represented by grayed out tiles in the user interface. The agent does not make use of sensors and has complete access to the environment. The available actions for each agent are Move, Pack, Drop, Equip, Eat, Fight, Say, Pull, Push and Stop. Sensors used in the AI-implementation are Closest Object, Global Position and Closest Item. Some small AI agents have been implemented, for example roving barbarians that hunt the player on sight. All agents are data-driven and can be added or removed from the environment at run-time.

### 4.2 Simulator for Evaluating the Subsumption Architecture

An implementation of the subsumption architecture [Brooks, 1985] was introduced to the agent part of the framework as part of the work towards evaluation the subsumption architecture for use in obstacle avoidance systems for cars [Woltjer and McGee, 2004]. This implementation allowed us to evaluate the agent part of the framework and enhance it. The subsumption architecture was integrated with an editor which allowed the user to change the subsumption network during simulation runs to experiment with new behaviours.

**Environment**

The subsumption agent was used in two environments. The architecture was developed and tested in the Haphazard environment, but mainly used in a traffic simulator. The traffic simulator used a straight road without any static obstacles. It used an approaching car as a dynamic obstacle and the user's input was merged with the subsumption architecture to get user driven obstacle avoidance.

**Agents**

The user controlled a car with acceleration and turning and the agent merged the user's input with sensor data in a subsumption network before it was actuated. The sensor data was a vector to the closest object within a preset distance, possible actions were accelerate, break, steer left, steer right. Another agent was also implemented, a car travelling with constant speed in the opposite direction.

### 4.3 Simulator for Dialogue System Development Support

Within the WITAS Unmanned Aerial Vehicle (UAV) project [Doherty *et al.*, 2000], the Hazard framework was used in implementing a simulator that supports development of a dialogue system for command and control of one or more UAVs. The new simulator replaced an old simulator that had been in development by several persons totalling approximately one man year. The implementation achieved about the same functionality (more in visualization, less in camera control) but better integration by one person in three weeks.

**Environment**

The environment consists of a three-dimensional map containing roads and buildings. The roads consist of several segments, intersections and dead ends. All roads have a name. Buildings are visualized as polygons and have a height. All buildings also have a name, color, material and one or more windows which can be observed by sensors. The environment is fully data-driven and new environments using these types of objects can easily be constructed.

**Agents**

There exist three types of agents:

WITAS UAV
: The WITAS UAV agent mainly consists of a socket interface which can receive commands from the dialog system, execute these and report their progress. The agent has a camera sensor and can detect cars and buildings that come within the camera's field of vision. A WITAS UAV can take off, land, ascend, descend, hover, fly to positions/buildings/heights, follow cars and follow roads to accomplish it's mission. A camera sensor detects buildings and cars within a preset distance. It is an interactive agent that can build plans and follow the commands of a user.

COMETS UAV
: The COMETS UAV agent was implemented as an experiment together with the COMETS project [Ollero *et al.*, 2004]. It implements an interface to the COMETS Multi-Level Executive [Gancet *et al.*, 2005] and uses the same set of actions and sensors as the WITAS UAV agent. Since the WITAS project and the COMETS project deal with approximately the same environment and agents, the interface could reuse most of the WITAS UAV implementation. The integration of the COMETS UAV into the simulation was made in about 12 man hours.

Car
: Cars drive along roads with a set speed. They can either drive around randomly or follow a preset route, but can't detect or interact with other Car agents. Cars are completely autonomous agents. They use only one action, "drive", and no sensors.

All agents are data-driven and can be added to the environment before the start up. They cannot be added during run-time (at present). The simulation is transparent and can be fully or partly substituted by a connection to a real world UAV. The visualization is 2D, but current work is extending both camera view and world view to three dimensions.

## 5 Evaluation

The evaluation of the framework is based on our own experience in developing simulations. It is focused on development and how suitable the framework is for use as middleware and as a framework for development of high-level and low-level AI for robotic applications.

### 5.1 Strengths

Clear design guidelines
: The framework gives clear design guidelines due to its structure. The flow for developing is generally:

  - What type of environment (discrete-continuous) shall I use?
  - What important objects exist in the environment?
  - What actions are available?
  - What information do agents need that is not available through success/fail information for actions? (i.e. what sensors are needed?)
  - What agents do we need?

Rapid development
: The different implementations have shown that the framework rapidly gives a working system. The only comparison that has been done on development time is with the replacement of the Simulator for Dialogue System Development Support. The implementation using

the Hazard framework cut the development time radically.

Scalability
: The framework is very scalable in the form of developing new agents or objects for an environment. It has not been tested how scalable it is in regard to the number of existing objects/agents or actions/sensors in an environment.

### 5.2 Weaknesses

Agents are tightly linked to the environment
: Since the framework was extracted from an integrated system, the agents are tightly linked to the environment and can have complete knowledge of the world without using sensors. This is a big drawback as it allows developers to "cheat" and use information that shouldn't be available to the AI implementation. The agent module should be completely separate from the environment.

Agent to agent interaction
: Since the design does not distinguish between success/fail messages for an action and sensor data, it is hard to develop agent to agent interactions. A solution for this problem could be to remove the success/fail notion from the environment side of the action and let the agent side sensor interpreter decide when an action has been successful or failed. This solution would also allow research into event/chronicle recognition.

New developers
: The system is well documented, but lacks examples and tutorials. This makes it harder for new developers to use the framework.

### 5.3 Limitations

Mid-level functionality
: Since the framework only supports high-level actions and is confusing on the part of sensor data, mid-level functionality is not intuitively supported. By mid-level functionality is meant functionality such as event/chronicle recognition, symbol grounding and similar robotic tasks. This is a disadvantage if the system is used for developing AI techniques for robotic systems since a developer can easily "cheat" with constructed events instead of having to identify them from sensor data.

Pre-defined set of actions
: Since the actions are pre-defined in the environment, both with regard to execution and evaluation of the execution (success/fail), an agent cannot learn new actions or interpret the result in new ways. Also, since the actions can be of different level of abstraction, it is hard to combine actions concurrently.

## 6 Related Work

Currently, the International Game Developers Association (IGDA) is pushing towards AI standard interfaces for computer games and is in the process of publishing the first drafts of a proposed standard. These standards are geared towards enabling game AI developers to reuse existing AI middleware and to concentrate on higher level AI tasks. IGDA is working on interfaces for common AI tasks and has currently working groups on interface standards for world interfacing, path planning, steering, finite state machines, rule-based systems and goal-oriented action planning. The work presented here is closest to the work on world interfacing, but since the draft was not available at the time of writing, it was impossible to compare.

The Testbed for Integrating and Evaluating Learning Techniques (TIELT) [Aha and Molineaux, 2004] is a free software tool that can be used to integrate AI systems with (e.g., real-time) gaming simulators, and to evaluate how well those systems learn on selected simulation tasks. TIELT is used as a configurable middleware between gaming simulators and learning systems and can probably be used as a middleware between general environments and agent frameworks with some modification. The middleware philosophy of TIELT differs from our implementation, TIELT sits as a black box between environment and agent while Hazard is only meant as a transparent interface without any real functionality, except if the framework is used on either side. The black box functionality would hide event/chronical recognition, symbol grounding, etc. . . in a robotic system. This means that special care has to be taken if the system is to be used with actual robotics.

The System for Parallel Agent Discrete Event Simulator (SPADES) [Riley and Riley, 2003] is a simulation framework with approximately the same concept as the Hazard framework with regards to the separation between agents and environments. It focuses on repeatability of simulations and uses a concept of Software-in-the-Loop. Software-in-the-Loop in SPADES measures the actual time an agent executes. Using this technology, it can give a large number of agents the same time-share by slowing down the simulation without sacrificing simulation detail and repeatability. SPADES is a discrete event simulator and uses a sense-think-act loop for the agents which limits its deliberation time to the time between receiving events until it has decided what to do. This limitation is minimized by allowing agents to tell the simulation that it wants to receive a *time notification* which works as a sense event. Our framework on the other hand sacrifices repeatability and agent timesharing to obtain a continuous, asynchronous time model which is more inline with robotic architectures than a discrete model. The agent developer can then decide to translate into a discrete time model or keep the continuous one.

There is also research on modules and frameworks that can be used in conjunction with a set of interfaces for cognitive systems, in particular DyKnow [Heinz and Doherty, 2004], a framework to facilitate event and chronicle recognition in a robotic architecture.

## 7 Conclusions and Future Work

The iterative development of framework and interfaces has enabled us to gain valuable experience in designing interfaces that are adequate for both robotic systems and simulated environments without sacrificing detail or ease of use. Our goal

is to develop an architecture for connecting AI and robotics with the following characteristics:

- Rapid environment/agent development
- Connects agents and environments
- Able to reuse both existing agents and environments
- Capable of acting as middleware between existing frameworks
- Usable in research of both simulations and actual robotic systems

Hazard is a mature system which has undergone several design iterations. It allows rapid development and reuse of agents and environments. It contains intuitive interfaces between agent and environment and can be used as middleware between existing frameworks. But Hazard has been found unsuitable for development of AI or simulations for actual robotic systems due to its inherent limitation in event recognition and actuator control. To be usable in a robotic AI implementation, the interfaces need to be layered to allow both for high-level AI frameworks and middle-level event and chronical recognition on the agent side. The framework for agents and environments also need to be structured in layers to support both discrete event and continuous time simulations with action/event and actuator/sensor interfaces.

Currently, work has been started on a new generation of interfaces and framework. This work is called CAIRo (Connecting AI to Robotics) and a first implementation with the framework has already been done. The development will focus on first validating the new framework with the current implementations and then continue the validation with middleware functionality and implementations of robotic systems.

## 8 Acknowledgements

This research work was funded by the Knut and Alice Wallenberg Foundation, Sweden, and by the Swedish National Graduate School in Computer Science (CUGS).

## References

[Aha and Molineaux, 2004] D.W. Aha and M Molineaux. Integrating learning in interactive gaming simulators. In D. Fu and J. Orkin, editors, *Challenges of Game AI: Proceedings of the AAAI'04 Workshop (Technical Report WS-04-04)*, San Jose, CA, 2004. AAAI Press.

[Andersson and Beskid, 2003] Peter J. Andersson and Lucian Cristian Beskid. The haphazard game project. http://haphazard.sf.net, 2003.

[Brooks, 1985] R. A. Brooks. A robust layered control system for a mobile robot. Memo 864, MIT AI Lab, September 1985.

[Doherty *et al.*, 2000] P. Doherty, G. Granlund, G. Krzysztof, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The witas unmanned aerial vehicle project. In *ECAI-00*, Berlin, Germany, 2000.

[Gancet *et al.*, 2005] Jérémi Gancet, Gautier Hattenberger, Rachid Alami, and Simon Lacroix. An approach to decision in multi-uav systems: architecture and algorithms. In *Proceedings of the ICRA-2005 Workshop on Cooperative Robotics*, 2005.

[Gerkey *et al.*, 2003] Brian Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, June 2003.

[Heinz and Doherty, 2004] Fredrik Heinz and Patrick Doherty. Dyknow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems*, 15(1), 2004.

[Kitano *et al.*, 1997] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 340–347, New York, 5–8, 1997. ACM Press.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3):1–64, 1987.

[Ollero *et al.*, 2004] Aníbal Ollero, Günter Hommel, Jeremi Gancet, Luis-Gonzalo Gutierrez, D.X. Viegas, Per-Erik Forssén, and M.A. González. Comets: A multiple heterogeneous uav system. In *Proceedings of the 2004 IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR 2004)*, Bonn (Germany), May 2004.

[Riley and Riley, 2003] Patrick F. Riley and George F. Riley. Spades - a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sanchez, D. Ferrin, and D.J. Morrice, editors, *Proceedings of the 2003 Winter Simulation Conference*, pages 817–825, 2003.

[Russel and Norvig, 1995] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Woltjer and McGee, 2004] R. Woltjer and K. McGee. Subsumption architecture as a framework to address human machine function allocation. Presented at SimSafe, http://130.243.99.7/pph/pph0220/simsafe/dok/ simsafe05.pdf, 2004.

# Extending Reinforcement Learning to Provide Dynamic Game Balancing

**Gustavo Andrade**
**Geber Ramalho**
**Hugo Santana**
Universidade Federal de Pernambuco
Centro de Informática
Cx. Postal 7851, 50732-970, Recife, Brazil
{gda,glr,hps}@cin.ufpe.br

**Vincent Corruble**
Université Paris 6
Laboratoire d'Informatique de Paris VI
Bôite 169 - 4 Place Jussieu
75252 PARIS CEDEX 05
Vincent.Corruble@lip6.fr

## Abstract

A recognized major concern for the game developers' community is to provide mechanisms to dynamically balance the difficulty level of the games in order to keep the user interested in playing. This work presents an innovative use of reinforcement learning to build intelligent agents that adapt their behavior in order to provide dynamic game balancing. The idea is to couple learning with an action selection mechanism which depends on the evaluation of the current user's skills. To validate our approach, we applied it to a real-time fighting game, obtaining good results, as the adaptive agent is able to quickly play at the same level as opponents with different skills.

## 1 Introduction

In computer games, the issue of providing a good level of challenge for the user is referred to as *game balancing*. It is widely recognized as a key feature of successful games [Falstein, 2004]. Balancing a game consists in changing parameters, scenarios and behaviors in order to avoid the extremes of getting the player frustrated because the game is too hard or becoming bored because the game is too easy [Koster, 2004]. The idea is too keep the user interested in playing the game from the beginning to the end. Unfortunately, setting a few pre-defined and static difficulty levels (e.g. beginner, intermediate and advanced) is not sufficient. In fact, not only should the classification of users' skill levels be fine-grained, but the game difficulty should also follow the players' personal evolutions, as they make progress via learning, or as they regress (for instance, after a long period without playing the game).

In order to deal with the problem, we avoided the drawbacks of directly learning to play at the same level as the user. Instead, we coupled two mechanisms. First, we build agents that are capable of learning optimal strategies, using Reinforcement Learning (RL). These agents are trained offline to exhibit a good initial performance level and keep learning during the game. Second, we provide the agents with an adaptive capability through an innovative action selection mechanism dependent on the difficulty the human player is currently facing. We validated our approach empirically, applying it to a real-time two-opponent fighting game named Knock'Em [Andrade *et al.*, 2004], whose functionalities are similar to those of successful commercial games, such as Capcom Street Fighter and Midway Mortal Kombat.

In the next section we introduce the problems in providing dynamic game balancing. Then, we briefly present some RL concepts and discuss their application to games. In Section 4 and 5, respectively, we introduce our approach and show its application to a specific game. Finally, we present some conclusions and ongoing work.

## 2 Dynamic Game Balancing

Dynamic game balancing is a process which must satisfy at least three basic requirements. First, the game must, as quickly as possible, identify and adapt itself to the human player's initial level, which can vary widely from novices to experts. Second, the game must track as close and as fast as possible the evolutions and regressions in the player's performance. Third, in adapting itself, the behavior of the game must remain believable, since the user is not meant to perceive that the computer is somehow facilitating things (e.g. by decreasing the agents' physical attributes or executing clearly random and inefficient actions).

There are many different approaches to address dynamic game balancing. In all cases, it is necessary to measure, implicitly or explicitly, the difficulty the user is facing at a given moment. This measure can be performed by a heuristic function, which some authors [Demasi and Cruz, 2002] call a "challenge function". This function is supposed to map a given game state into a value that specifies how easy or difficult the game feels to the user at that specific moment. Examples of heuristics used are: the rate of successful shots or hits, the numbers of pieces which have been won and lost, life point evolution, time to complete a task, or any metric used to calculate a game score.

Hunicke and Chapman's approach [2004] controls the game environment settings in order to make challenges easier or harder. For example, if the game is too hard, the player gets more weapons, recovers life points faster or faces fewer opponents. Although this approach is effective, its application is constrained to game genres where such particular manipulations are possible. This approach could not be used, for instance, in board games, where the players share the same features.

Another approach to dynamic game balancing is to modify the behavior of the Non-Player Characters (NPCs), characters controlled by the computer and usually modeled as intelligent agents. A traditional implementation of such an agent's intelligence is to use behavior rules, defined during game development using domain-specific knowledge. A typical rule in a fighting game would state "*punch opponent if he is reachable, chase him, otherwise*". Besides the fact that it is time-consuming and error-prone to manually write rule bases, adaptive behavior can hardly be obtained with this approach. Extending such an approach to include opponent modeling can be made through dynamic scripting [Spronck *et al.*, 2004], which assigns to each rule a probability of being picked. Rule weights are dynamically updated throughout the game, reflecting the success or failure rate of each rule. The use of this technique to game balancing can be made by not choosing the best rule, but the closest one to the user level. However, as game complexity increases, this technique requires a lot of rules, which are hard to build and maintain. Moreover, the performance of the agent becomes limited by the best designed rule, which can not be good enough for very skilled users.

A natural approach to address the problem is to use machine learning [Langley, 1997]. Demasi and Cruz [2002] built intelligent agents employing genetic algorithm techniques to keep alive those agents that best fit the user level. Online coevolution is used in order to speed up the learning process. Online coevolution uses pre-defined models (agents with good genetic features) as parents in the genetic operations, so that the evolution is biased by them. These models are constructed by offline training or by hand, when the agent genetic encoding is simple enough. This is an innovative approach, indeed. However, it shows some limitations when considering the requirements stated before. Using pre-defined models, the agent's learning becomes restricted by these models, jeopardizing the application of the technique for very skilled users or users with uncommon behavior. As these users do not have a model to speed up learning, it takes a long time until the agents reaches the user level. Furthermore, this approach works only to increase the agent's performance level. If the player's skill regresses, the agent cannot regress also. This limitation compels the agent to always start the evolution from the easiest level. While this can be a good strategy when the player is a beginner, it can be bothering for skilled players, since they will probably need to wait a lot for the agents' evolution.

# 3 Reinforcement Learning in Games

## 3.1 Background

Reinforcement Learning (RL) is often characterized as the problem of "learning what to do (how to map situations into actions) so as to maximize a numerical reward signal" [Sutton and Barto, 1998]. This framework is often defined in terms of the Markov Decision Processes (MDP) formalism, in which we have an agent that sequentially makes decisions in an environment: it perceives the current state *s*, from a finite set *S*, chooses an action *a*, from a finite set *A*, reaches a new state *s'* and receives an immediate reward signal *r(s,a)*. The information encoded in *s* should satisfy the Markov Property, that is, it should summarize all present and past sensations in such a way that all relevant information is retained. Implicitly, the reward signal *r(s,a)* determines the agent's objective: it is the feedback which guides the desired behavior.

The main goal is to maximize a long-term performance criterion, called return, which represents the expected value of future rewards. The agent then tries to learn an *optimal policy* $\pi^*$ which maximizes the expected return. A policy is a function $\pi(s) \rightarrow a$ that maps state perceptions into actions. Another concept in this formalism is the *action-value function*, $Q^{\pi}(s,a)$, defined as the expected return when starting from state *s*, performing action *a*, and then following $\pi$ thereafter. If the agent can learn the optimal action-value function $Q^*(s,a)$, an optimal policy can be constructed greedily: for each state *s*, the best action *a* is the one that maximizes $Q^*(s,a)$.

A traditional algorithm for solving MDPs is Q-Learning [Sutton and Barto, 1998]. It consists in iteratively computing the values for the action-value function, using the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma.V(s') - Q(s,a)]$$

in which $V(s') = \max_a Q(s',a)$, $\alpha$ is the learning rate and $\gamma$ is a discount factor, which represents the relative importance of future against immediate rewards.

There are some particular characteristics of RL which makes it attractive to applications like computer games. First, different from other kinds of machine learning techniques, it does not require any previous example of good behavior, being able to find optimal strategies only through trial and error, thus reducing the development effort necessary to build the AI of the game. Furthermore,

it can be applied offline, as a pre-processing step during the development of the game, and then be continuously improved online after its release [Manslow, 2003].

RL has been successfully used in board games, like backgammon [Tesauro, 1994] and checkers [Samuel, 1967], as well as in other domains such as robotic soccer [Merke and Riedmiller, 2001]. The work presented here differs from these mainly in two different aspects. First, many of these applications are turn-based games. We deal in this work with real-time, dynamic games, which lead in general to more complex state representations, and the need to address time processing issues. Second, while these works are basically interested in making the computer beat any opponent, our goal is to have the computer always adequately challenge his opponent, whether or not he/she is playing optimally.

### 3.2 Directly Learning to Play at the User Level

The problem of dynamically changing the game level could be addressed with RL by carefully choosing the reward so that the agent learns to act in the same level of user skill. When the game is too easy or too hard a negative reward is given to the agent, otherwise the feedback is a positive reward.

This approach has the clear benefit that the mathematical model of learning actually corresponds to the goal of the agent. However, this approach has a lot of disadvantages, with respect to the requirements stated in Section 2. First, using this approach, the agent will not be able immediately to fight against expert players, since it would need to learn first. Second, this learning approach may lead to non-believable behaviors. For instance, in a fight game such as Knock'em, the agent can learn that after hitting the user hard, it must be static and use no defense, letting the user hit him back, so as the overall game score remains balanced.

## 4 Challenge-Sensitive Action Selection

Given the difficulties in directly learning to play at the user level, an alternative is to face dynamic game balancing as two separate problems: learning (building agents that can learn optimal strategies) and adapting (providing action selection mechanisms for providing game balance, possibly using sub-optimal actions).

Our approach faces both problems with reinforcement learning. Due to the requirement of being immediately able to play at the human player level, including expert ones, at the beginning of the game, offline training is needed to bootstrap the learning process. This can be done by letting the agent play against itself (self-learning) [Kaelbling *et al.*, 1996], or other pre-programmed agents [Madeira *et al.*, 2004]. Then, online learning is used to adapt continually this initially built-in intelligence to the specific human opponent, in order to discover the most adapted strategy to play against him or her.

Concerning dynamic game balancing, the idea is to find the adequate policy for choosing actions that provide a good game balance, i.e., actions that keep agent and human player at approximately the same performance level. In our approach, according to the difficulty the player is facing, we propose that the agent choose actions with high or low expected return. For a given situation, if the game level is too hard, the agent does not choose the optimal action (the one with highest action value), but chooses progressively sub-optimal actions until its performance is as good as the player's. This entails choosing the second best action, the third one, and so on, until it reaches the player's level. Similarly, if the game level becomes too easy, it will choose actions whose values are higher, possibly until it reaches the optimal policy. In this sense, our idea of adaptation shares the same principles of the one proposed by Spronck *et al.* [2004], although the techniques used are different and the works have been developed in parallel.

In this challenge-sensitive action selection mechanism, the agent periodically evaluates if it is at the same level of the player, through the challenge function, and according to this result, maintains or changes its performance level. The agent does not change its level until the next cycle. This evaluation cycle is strongly tied to the particular environment, in which the agent acts, depending, in particular, on the delay of the rewards. If the cycle is too short, the agent can exhibit a random behavior; if the cycle is too long, it will not match the player evolution (or regression) fast enough.

Our approach uses the order relation naturally defined over the actions in a given state by the action-value function, which is automatically built during the learning process. As these values estimate the individual quality of each possible action, it is possible to have a strong and fast control on the agent behavior and consequently on the game level.

It is important to notice that this technique changes only the action selection procedure, while the agent keeps learning during the entire game. It is also worthwhile to stress that this action selection mechanism coupled with a successful offline learning phase (during the bootstrap phase), can allow the agent to be fast enough to play at the user level at the beginning of the game, no matter how experienced he or she is.

## 5 Case Study

### 5.1 Game Description

As a case study, the approach was applied to Knock'Em [Andrade *et al.*, 2004], a real-time fighting game where

two players face each other inside a bullring. The main objective of the game is to beat the opponent. A fight ends when the life points of one player (initially, 100 points) reach zero, or after 1min30secs of fighting, whatever comes first. The winner is the fighter which has the highest remaining life at the end. The environment is a bidimensional arena in which horizontal moves are free and vertical moves are possible through jumps. The possible attack actions are to punch (strong or fast), to kick (strong or fast), and to launch fireballs. Punches and kicks can also be performed in the air, during a jump. The defensive actions are blocking or crouching. While crouching, it is also possible for a fighter to punch and kick the opponent. If the fighter has enough spiritual energy (called mana), fireballs can be launched. Mana is reduced after a fireball launch and continuously refilled over time at a fixed rate.

## 5.2 Learning to Fight

The fighters' artificial intelligence is implemented as a reinforcement learning task. As such, it is necessary to code the agents' perceptions, possible actions and reward signal. We used a straightforward tabular implementation of Q-learning, since, as we will show, although simple, this approach can provide very good results. The state representation (agent perceptions) is represented by the following tuple:

$$S = (S_{agent}, S_{opponent}, D, M_{agent}, M_{opponent}, F)$$

$\mathbf{S_{agent}}$ stands for agent state (stopped, jumping, or crouching), and represents the ones in which it can effectively make decisions, i.e., change its state. $\mathbf{S_{opponent}}$ stands for opponent state (stopped, jumping, crouching, attacking, jump attacking, crouch attacking, and blocking). **D** represents opponent distance (near, medium distance and far away). **M** stands for agent or opponent mana (sufficient or insufficient to launch one fireball). Finally, **F** stands for enemies' fireballs configuration (near, medium distance, far away, and no existence).

The agent's actions are the ones possible to all fighters: punching and kicking (strong or fast), coming close, running away, jumping, jumping to approximate, jumping to escape, launching fireball, blocking, crouching and standing still.

The reinforcement signal is based on the difference of life caused by the action (life taken out from opponent minus life lost by the agent). As a result, the agent reward is always in the range [-100, 100]. Negative rewards mean bad performance, because the agent lost more life than was taken from the opponent, while positive rewards are the desired agent's learning objective.

The RL agent's initial strategy is built through offline learning against other agents. We used state-machine, random and another RL agent (self-learning) as trainers. The 3 agents trained with different opponents fought then against each other in a series of 30 fights. The resulting mean of life differences after each fight showed that the agents trained against the random and the learning agents obtained the best performances. Since the difference between these two latter agents is not significant, in the next section experiments, the agent that learned against a random agent is considered as the starting point for the online RL agent. In all RL agents, the learning rate was fixed at 0.50 and the reward discount rate at 0.90, both for offline and online learning. This high learning rate is used because, as the opponent can be a human player with dynamic behavior, the agent needs to quickly improve its policy against him or her.

## 5.3 Fighting at the User's Level

The action selection mechanism proposed was implemented and evaluated in Knock'em. The challenge function used is based on the life difference during the fights. In order to stimulate the player, the function is designed so that the agent tries to act better than him or her. Therefore, we empirically stated the following heuristic challenge function: when the agent's life is smaller than the player's life, the game level is easy; when their life difference is smaller than 10% of the total life (100), the game level is medium; otherwise, it is difficult.

$$f = \begin{cases} easy, if \quad L(agent) < L(player) \\ medium, if \quad L(agent) - L(player) < 10 \\ difficult, \ otherwise \end{cases}$$

The evaluation cycle used is 100 game cycles (or game loops). This value was empirically set to be long enough so that the agent can get sufficient data about the opponent before evaluating him or her, and short enough so that the agent quickly adapt to the player's behavior.

The implementation of the proposed action selection mechanism is summarized as follows. The agent starts the game acting at its medium performance level. As in Knock'em there are thirteen possible actions and so thirteen possible levels in our adaptive agent, it starts at the sixth level. During the game, the agent chooses the action which is the $6^{th}$ highest value at the action-value function. After the evaluation cycle (100 game cycles), it evaluates the player through the previous challenge function. If the level is easy, the agent regresses to the $7^{th}$ level; if it is difficult, it progresses to the $5^{th}$ level; otherwise, it remains on the $6^{th}$ level. As there are approximately 10 evaluations through a single fight, the agent can advance to the best performance level or regress to the worst one in just the first fight.

## 5.4 Experimental Results

Since it is too expensive and complex to run experiments with human players, we decided to initially validate our

adaptation approach comparing the performance of two distinct agents: a traditional reinforcement learning (playing as well as possible), and the adaptive agent (implementing the proposed approach). Both agents were previously trained against a random agent. The evaluation scenario consists of a series of fights against different opponents, simulating the diversity of human players strategies: a state-machine (SM, static behavior), a random (RD, unforeseeable behavior) and a trained traditional RL agent (TRL, intelligent and with learning skills).

Each agent being evaluated plays 30 fights against each opponent. The performance measure is based on the final life difference in each fight. Positive values represent victories of the evaluated agent, whereas negative ones represent defeats.

In each evaluation, testing hypotheses (p-value significance tests) are provided to check whether the mean of the differences of life in each set is different from zero at a significance level of 5%. If the mean is significantly different from zero, then one of the agents is better than the other; otherwise, the agents must be playing at the same level.

Figure 1 shows the traditional RL (TRL) agent performance against each of the other 3 agents. The positive values of the white and black lines show that the agent can beat a random and a state-machine opponent. The gray line shows that two TRL fighters have a similar performance while fighting against each other. Table 1 summarizes these data and shows that the TRL is significantly better than a SM and a RD agent, but play at the same level of other TRL agent.

Figure 2 illustrates the adaptive RL agent performance. Although this agent has the same capabilities as traditional RL, because their learning algorithms and their initial knowledge are the same, the adaptive mechanism forces the agent to act at the same level as the opponent. The average performance shows that most of the fights end with a small difference of life, meaning that both fighters had similar performance. Table 2 confirms these results showing that the adaptive agent is equal to the three opponent agents.

The adaptive RL agent (ARL) obtains similar performance against different opponents because it can interleave easy and hard actions, balancing the game level. Figure 3 shows a histogram with the agent actions frequency. The thirteen x-values correspond to all actions the agent can perform. The leftmost action is to the one with the highest value at the action-value function, while the rightmost is the one with lowest value. This way, the leftmost is the action which the agent believes to be the strongest one in each state and the rightmost is the action it believes to be the lighter one. The high frequency of powerful actions against the TRL agent shows that the ARL agent needed to play at an advanced performance level. The frequency of actions against the SM and RD agents shows that the ARL played in an easier level, around the 9th and 11th one.

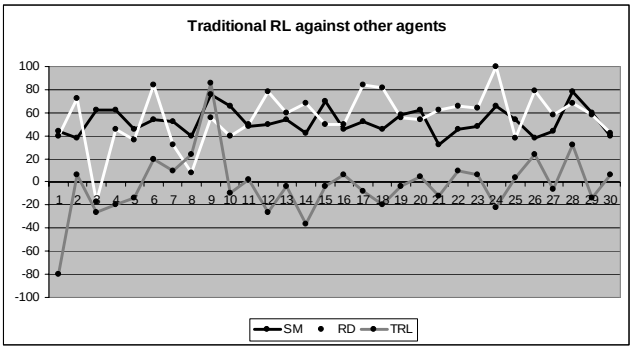


Figure 1: Traditional RL agent performance (SM = State Machine, RD = Random, TRL = traditional RL).

Table 1: Traditional RL performance analysis

| | Mean | Std. deviation | p-value | Difference is significant? |
|---|---|---|---|---|
| **SM** | 52,47 | 11,54 | 0,00 | Yes |
| **RD** | 55,47 | 23,35 | 0,00 | Yes |
| **TRL** | -2,17 | 27,12 | 0,66 | No |

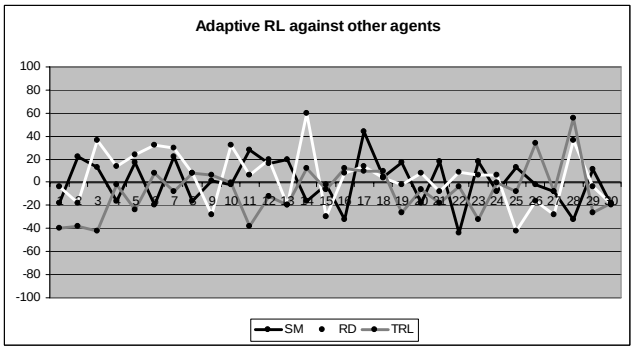


Figure 2: Adaptive RL agent performance (SM = State Machine, RD = Random, TRL = traditional RL).

Table 2: Adaptive RL performance analysis

| | Mean | Std. deviation | p-value | Difference is significant? |
|---|---|---|---|---|
| **SM** | 0,37 | 20,67 | 0,92 | No |
| **RD** | 4,47 | 23,47 | 0,30 | No |
| **TRL** | -7,33 | 21,98 | 0,07 | No |

## 6 Conclusions

In this paper, we presented agents that can analyze their own learned knowledge in order to choose actions which are just good enough to be challenging for the human opponent, whatever his or her level. These agents dynamically adapt their behavior while learning in order

to keep the game level adapted to the current user skills, a key problem in computer games.

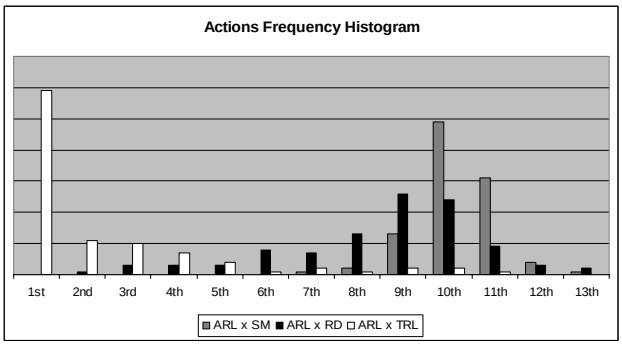


Figure 3: Histogram for the adaptive agent

The results obtained in a real-time fighting game indicate the effectiveness of our approach. In fact, although the adaptive agent could easily beat their opponents, it plays close to their level, interleaving wins and defeats. However, the proposed approach can be used in different game genres [Manslow, 2003]. It is only necessary to formulate the game as a reinforcement learning task, using as challenge function a heuristic based in the game score. Moreover, the concepts presented in this paper can be used with different learning techniques, as long as an order relation for the actions can be defined in each game state. Finally, the need for user adaptation is also a major concern in applications such as computer aided instruction [Beck *et al.*, 1997]. In general, the present work is useful when the skill or knowledge of the user must be continuously assessed to guide the choice of the adequate challenges (e.g. missions, exercises, questions, etc.) to be proposed.

We are now running systematic experiments with around 50 human players, who are playing against different types of agents, including our adaptive one, to check which one is really entertaining. The first results are encouraging, but the study is ongoing.

## References


[Andrade *et al.*, 2004] Gustavo Andrade, Hugo Santana, André Furtado, André Leitão, and Geber Ramalho. Online Adaptation of Computer Games Agents: A Reinforcement Learning Approach. In *Proceedings of the 3rd Brazilian Workshop on Computer Games and Digital Entertainment*, Curitiba, 2004.

[Beck *et al.*, 1997] Joseph Beck., Mia Stern, and Beverly Woolf. Using the Student Model to Control Problem Difficulty. In *Proceedings of Sixth International Conference on User Modeling*, pages 277-288, Vienna, 1997.

[Demais and Cruz, 2002] Pedro Demasi and Adriano Cruz. Online Coevolution for Action Games. In *Proceedings of the 3rd International Conference on Intelligent Games and Simulation*, pages 113-120, London, 2002.

[Falstein, 2004] Noah Falstein. The Flow Channel. *Game Developer Magazine*, May Issue, 2004.

[Hunicke and Chapman, 2004] Robin Hunicke and Vernell Chapman. AI for Dynamic Difficulty Adjustment in Games. *Challenges in Game Artificial Intelligence AAAI Workshop*, pages 91-96, San Jose, 2004.

[Kaelbling *et al.*, 1996] Leslie Kaelbling, Michael Littman, and Andrew Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, pages 4:237-285, 1996.

[Koster, 2004] Raph Koster. *Theory of Fun for Game Design*, Paraglyph Press, Phoenix, 2004.

[Langley, 1997] Pat Langley. Machine Learning for Adaptive User Interfaces. *Kunstiche Intellugenz*, pages 53-62, 1997.

[Madeira *et al.*, 2004] Charles Madeira, Vincent Corruble, Geber Ramalho and Bohdana Ratitch. Bootstrapping the Learning Process for the Semi-automated Design of a Challenging Game AI. *Challenges in Game Artificial Intelligence AAAI Workshop*, pp. 72-76, San Jose, 2004.

[Manslow, 2003] John Manslow. Using Reinforcement Learning to Solve AI Control Problems. In *Steve Rabin, editor, AI Game Programming Wisdom 2*, Charles River Media, Hingham, MA, 2003.

[Merke and Riedmiller, 2001] Artur Merke and Martin Riedmiller. Karlsruhe Brainstormers - A Reinforcement Learning Approach to Robotic Soccer. RoboCup 2001. *RoboCup-2000: Robot Soccer World Cup IV*, pages 435-440, London, 2001.

[Samuel, 1967] A.L. Samuel. Some studies in machine learning using the game of checkers II-Recent progress. *IBM Journal on Research and Development*, pages 11:601-617, 1967.

[Spronck *et al.*, 2004] Pieter Spronck, Ida Sprinkhuizen-Kuyper and Eric Postma. Difficulty Scaling of Game AI. In *Proceedings of the 5th International Conference on Intelligent Games and Simulation*, pages 33-37, Belgium, 2004

[Sutton and Barto, 1998] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Massachusetts, 1998.

[Tesauro, 1994] Garry Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, pages 6(2): 215-219, The MIT Press, Massachusetts, 1994.

# Best-Response Learning of Team Behaviour in Quake III

**Sander Bakkes, Pieter Spronck** and **Eric Postma**
Universiteit Maastricht
Institute for Knowledge and Agent Technology (IKAT)
P.O. Box 616, NL-6200 MD Maastricht, The Netherlands
{s.bakkes, p.spronck, postma}@cs.unimaas.nl

## Abstract

This paper proposes a mechanism for learning a best-response strategy to improve opponent intelligence in team-oriented commercial computer games. The mechanism, called TEAM2, is an extension of the TEAM mechanism for team-oriented adaptive behaviour explored in [Bakkes *et al.*, 2004] and focusses on the exploitation of relevant gameplay experience. We compare the performance of the TEAM2 mechanism with that of the original TEAM mechanism in simulation studies. The results show the TEAM2 mechanism to be better able to learn team behaviour. We argue that the application as an online learning mechanism is hampered by occasional very long learning times due to an improper balance between exploitation and exploration. We conclude that TEAM2 improves opponent behaviour in team-oriented games and that for online learning the balance between exploitation and exploration is of main importance.

## 1 Introduction

In recent years, commercial computer game developers have emphasised the importance of high-quality game opponent behaviour. *Online learning* techniques may be used to significantly improve the quality of game opponents by endowing them with the capability of adaptive behaviour (i.e., artificial creativity and self-correction). However, to our knowledge online learning has never been used in an actual commercial computer game (henceforth called 'game'). In earlier work [Bakkes *et al.*, 2004], we have proposed a mechanism named TEAM (Team-oriented Evolutionary Adaptability Mechanism) for team-oriented learning in games. Our experiments revealed TEAM to be applicable to commercial computer games (such as Quake-like team-games). Unfortunately, the applicability is limited due to the large variation in the time needed to learn the appropriate tactics.

This paper describes our attempts to improve the efficiency of the TEAM mechanism using *implicit opponent models* [van den Herik *et al.*, 2005]. We propose an extension of TEAM called TEAM2. The TEAM2 mechanism employs a data store of a limited history of results of tactical team behaviour, which constitutes an implicit opponent model, on which a best-response strategy [Carmel and Markovitch, 1997] is formulated. We will ague that *best-response learning* of team-oriented behaviour can be applied in games. We investigate to what extent it is suitable for online learning.

The outline of this paper is as follows. Section 2 discusses team-oriented behaviour (team AI) in general, and the application of adaptive team AI in games in particular. The TEAM2 best-response learning mechanism is discussed in section 3. In section 4, an experiment to test the performance of the mechanism is discussed. Section 5 reports our findings, and section 6 concludes and indicates future work.

## 2 Adaptive Team AI in Commercial Computer Games

We defined adaptive team AI as the behaviour of a team of adaptive agents that competes with other teams within a game environment [Bakkes *et al.*, 2004]. Adaptive team AI consists of four components: (1) the individual agent AI, (2) a means of communication, (3) team organisation, and (4) an adaptive mechanism.

Figure 1: Screenshot of the game QUAKE III. An agent fires at a game opponent.

The first three components are required for agents to establish team cohesion, and for team-oriented behaviour to emerge. The fourth component is crucial for improving the quality of the team during gameplay. The next sub-sections discuss a mechanism for adaptive team AI, and its performance.

### 2.1 The Team-oriented Evolutionary Adaptability Mechanism (TEAM)

The observation that humans players prefer to play against other humans over players against artificial opponents [van Rijswijck, 2003], led us to design the Team-oriented Evolutionary Adaptability Mechanism (TEAM). TEAM is an online evolutionary learning technique designed to adapt the team AI of Quake-like games. TEAM assumes that the behaviour of a team in a game is defined by a small number of parameters, specified per game state. A specific instance of team behaviour is defined by values for each of the parameters, for each of the states. TEAM is defined as having the following six properties: 1) state-based evolution, 2) state-based chromosome encoding, 3) state-transition-based fitness function, 4) fitness propagation, 5) elitist selection, and 6) manually-designed initialisation [Bakkes *et al.*, 2004].

For evolving successful behaviour, typical evolutionary learning techniques need thousands of trials (or more). Therefore, at first glance such techniques seem unsuitable for the task of online learning. Laird [2000] is skeptical about the possibilities offered by online evolutionary learning in games. He states that, while evolutionary algorithms may be applied to tune parameters, they are "grossly inadequate when it comes to creating synthetic characters with complex behaviours automatically from scratch". In contrast, the results achieved with the TEAM mechanism in the game QUAKE III show that it is certainly possible to use online evolutionary learning in games.

### 2.2 Enhancing the Performance of TEAM

Spronck [2005] defines four requirements for qualitatively acceptable performance were defined: speed, robustness, effectiveness, and efficiency. For the present study, the requirement of efficiency is of main relevance. Efficiency is defined as the learning time of the mechanism. In adaptive team AI, efficiency depends on the number of learning trials needed to adopt effective behaviour. Applied to the QUAKE III capture-the-flag (CTF) team game, the TEAM mechanism requires about 2 hours of real-time play to significantly outperform the opponent. Since QUAKE III matches take on average half an hour, the TEAM mechanism lacks efficiency to enable successful online learning in games such as QUAKE III.

When one aims for efficient adaptation of opponent behaviour in games, the practical use of evolutionary online learning is doubtful [Spronck, 2005]. Therefore, the design of TEAM needs to be enhanced with a different approach to learning team-oriented behaviour. The enhanced design, named TEAM2, is discussed next.

## 3 Best-Response Learning of Team-oriented Behaviour

The design of TEAM2, aimed at efficiently adapting opponent behaviour, is based on a best-response learning approach (instead of evolutionary learning)[1]. This section discusses the properties of the enhanced design: (1) a symbiotic learning concept, (2) learning a best-response team strategy, (3) a state-transition-based fitness function, and (4) a scaled roulette-wheel selection. The popular QUAKE III CTF game [van Waveren and Rothkrantz, 2001], is used for illustrative purposes.

### 3.1 Symbiotic Learning

Symbiotic learning is a concept for learning adaptive behaviour for *a team as a whole* (rather than learning adaptive behaviour for each individual). The TEAM mechanism successfully applied the concept for the purpose of adapting opponent behaviour in team-oriented games. The onset of the design of TEAM was the observation that the game state of team-oriented games can typically be represented as a finite state machine (FSM). By applying an instance of an adaptive mechanism to each state of the FSM, one is able to learn relatively uncomplicated team-oriented behaviour for the specific state. Cooperatively, from all instances of the applied adaptive mechanism, relatively complex team-oriented behaviour emerges in a computationally fast fashion. The concept of symbiotic learning is illustrated in figure 2. The figure exemplifies how instances of an adaptive mechanism cooperatively learn team-oriented behaviour, which is defined as the combination of the local optima for the states (in this example there are four states).

An instance of the adaptive mechanism automatically generates and selects the best team-configuration for the specific state. A team-configuration is defined by a small number of parameters which represent team behaviour (e.g. one team-configuration can represent an offensive tactic, whereas another team-configuration can represent a defensive tactic).

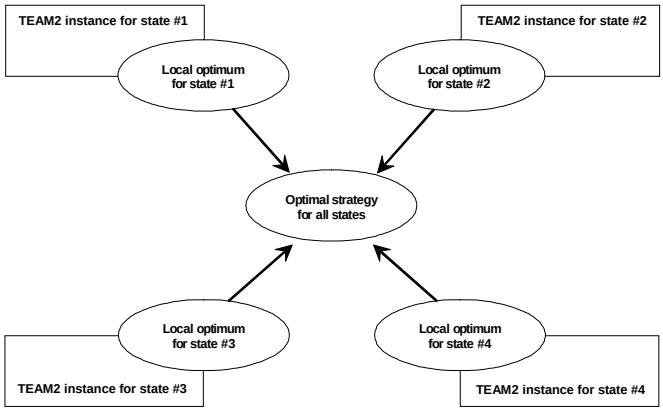


Figure 2: Symbiotic learning.

[1]Since TEAM2 is not inspired by evolutionary algorithms, we let the reader imagine that the letter 'E' is an abbreviation for 'Exploitative' (instead of 'Evolutionary').

### 3.2 Learning a Best-Response Team Strategy

Adaptation to the opponent takes place via an implicit opponent model, which is built and updated when the team game is in progress. Per state of the game, the sampled data merely concerns the specific state and represents all possible team-configurations for the state. The implicit opponent model consists of historic data of results per team-configuration per state. An example of the structure of an implicit opponent model is given in table 1. In the example, the team-configuration represents the role division of a team with four members. Each of which has either an offensive, a defensive or an roaming role. The history can anything from a store of fitness values, to a complex data-structure.

| *Team configuration* | *History* | *Fitness* |
|---|---|---|
| (0,0,4) | [0.1,0.6,...,0.5] | 0.546 |
| (0,1,3) | [0.3,0.1,...,0.2] | 0.189 |
| ⋮ | ⋮ | ⋮ |
| (4,0,0) | [0.8,0.6,...,0.9] | 0.853 |

Table 1: Example of an implicit opponent model for a specific state of the QUAKE III capture-the-flag game.

On this basis, a best-response strategy is formulated when the game transits from one state to another. For reasons of efficiency and relevance, only recent historic data are used for the learning process.

### 3.3 State-transition-based Fitness Function

The TEAM2 mechanism uses a fitness function based on state transitions. Beneficial state transitions reward the tactic that caused the state transition, while detrimental state transitions penalise it. To state transitions that directly lead to scoring (or losing) a point, the fitness function gives a reward (or penalty) of $4$. Whereas to the other state transitions, the fitness function gives a reward (or penalty) of $1$. This ratio is empirically decided by the experimenters. In figure 3, an example of annotations on the FSM of the QUAKE III CTF game is given.

Usually, judgement whether a state transition is beneficial or detrimental cannot be given immediately after the transition; it must be delayed until sufficient game-observations are gathered. For instance, if a state transition happens from a state that is neutral for the team to a state that is good for the team, the transition seems beneficial. However, if this is immediately followed by a second transition to a state that is bad for the team, the first transition cannot be considered beneficial, since it may have been the primary cause for the second transition.

### 3.4 Scaled Roulette-Wheel Selection

The best-response learning mechanism selects the preferred team-configuration by implementing a roulette wheel method [Nolfi and Floreano, 2000], where each slot of the roulette wheel corresponds to a team-configuration in the state-specific solution space, and the size of the slot is proportional to the obtained fitness-value of the team-configuration. The selection mechanism quadratically scales the fitness values to select the higher-ranking team-configurations more often,

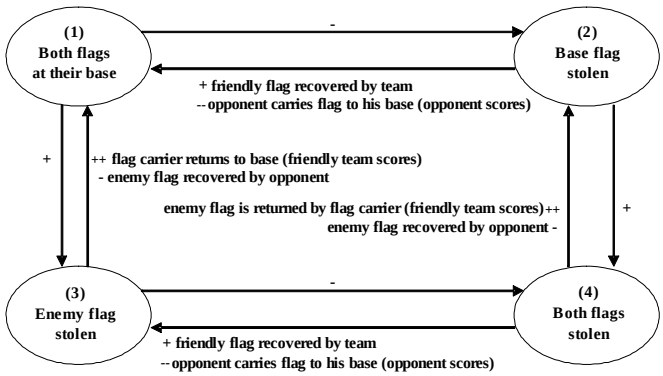


Figure 3: Annotated finite state machine of QUAKE III CTF. Highly beneficial and beneficial transitions are denoted with "$++$" and "$+$" respectively, whereas detrimental and highly detrimental state transitions are denoted with "$-$" and "$--$" respectively.

acknowledging that game opponent behaviour must be non-degrading. In acknowledgement of the inherent randomness of a game environment, the selection mechanism protects against selecting inferior top-ranking team-configurations.

## 4 Experimental Study of the TEAM2 Mechanism

To assess the efficiency of the TEAM2 mechanism, we incorporated it in the QUAKE III CTF game. We performed an experiment in which an adaptive team (controlled by TEAM2) is pitted against a non-adaptive team (controlled by the QUAKE III team AI). In the experiment, the TEAM2 mechanism adapts the tactical behaviour of a team to the opponent. A tactic consists of a small number of parameters which represent the offensive and defensive division of roles of agents that operate in the game.

The inherent randomness in the QUAKE III environment requires the learning mechanism to be able to successfully adapt to significant behavioural changes of the opponent. Both teams consist of four agents with identical individual agent AI, identical means of communication and an identical team organisation. They only differ in the control mechanism employed (adaptive or non-adaptive).

### 4.1 Experimental Setup

An experimental run consists of two teams playing QUAKE III CTF until the game is interrupted by the experimenter. On average, the game is interrupted after two hours of gameplay, since the original TEAM mechanism typically requires two hours to learn successful behaviour, whereas the TEAM2 mechanism should perform more efficiently. We performed $20$ experimental runs with the TEAM2 mechanism. The results obtained will be compared to those obtained with the TEAM mechanism ($15$ runs, see [Bakkes *et al.*, 2004]).

### 4.2 Performance Evaluation

To quantify the performance of the TEAM2 mechanism, we determine the so-called turning point for each experimental run. The turning point is defined as the time step at which the adaptive team takes the lead without being surpassed by the non-adaptive team during the remaining time steps.

We defined two performance indicators to evaluate the efficiency of TEAM2: the median turning point and the mean turning point. Both indicators are compared to those obtained with the TEAM mechanism. The choice for two indicators is motivated by the observation that the amount of variance influences the performance of the mechanism [Bakkes *et al.*, 2004].

To investigate the variance of the experimental results, we defined an outlier as an experimental run which needed more than 91 time steps to acquire the turning point (the equivalent of two hours).

### 4.3 Results

In table 2 an overview of the experimental results of the TEAM2 experiment is given. It should be noted that in two tests, the run was prematurely interrupted without a turning point being reached. We incorporated these two test as having a turning as high as the highest outlier, which is 358. Interim results indicate that, should the runs be not prematurely interrupted, their turning points would have been no more than half of this value.

The median turning point acquired is 38, which is significantly lower that the median turning point of the TEAM mechanism, which is 54. The mean turning point acquired with TEAM2, however, is significantly higher than the mean turning point acquired with the TEAM mechanism (102 and 71, respectively). The percentage of outliers in the total number of tests is about equal. However, the range of the outliers has significantly increased for TEAM2.

To illustrate the course of an experimental run, we plotted the performance for a typical run in figure 4. The performance is expressed in terms of the lead of the adaptive team, which is defined as the score of the adaptive team minus the score of the non-adaptive team. The graph shows that, initially, the adaptive team attains a lead of approximately zero. At the turning point (labeled 38 in figure 4), the adaptive team takes the lead over the non-adaptive team. Additionally, the graph reveals that the adaptive team outperforms the non-adaptive team without any significant degradation in its performance.

| | | TEAM | TEAM2 |
|---|---|---|---|
| *# Experiments* | *Total* | 15 | 20 |
| | *Outliers* | 4 | 6 |
| | *Outliers in %* | 27% | 30% |
| *Mean* | | 71.33 | 102.20 |
| *Std. Deviation* | | 44.78 | 125.29 |
| *Std. Error of Mean* | | 11.56 | 28.02 |
| *Median* | | 54 | 38 |
| *Range* | | 138 | 356 |
| *Minimum* | | 20 | 2 |
| *Maximum* | | 158 | 358 |

Table 2: Summary of experimental results. With TEAM2 the median turning point is significantly lower, yet, outliers have a negative effect on the mean turning point.

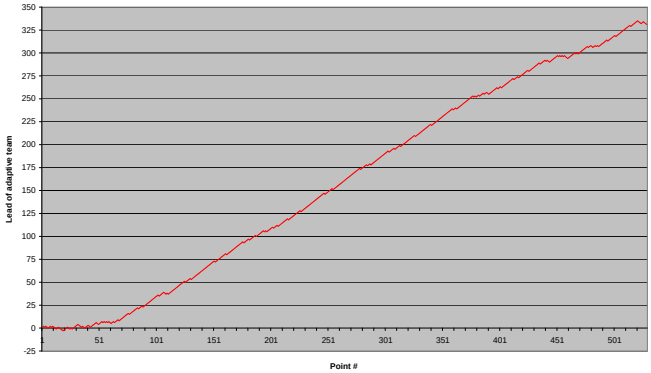


Figure 4: Illustration of typical experimental results obtained with the TEAM2 mechanism. The graph shows the lead of the adaptive team over the non-adaptive team as a function of the number of scored points.

### 4.4 Evaluation of the Results

The experimental results show that TEAM2 is able to successfully adapt game opponent behaviour in an highly non-deterministic environment, as it challenged and defeated the fine-tuned QUAKE III team AI.

The results listed in table 1 show that the TEAM2 mechanism outperforms the TEAM mechanism in terms of the median turning point. However, the mean turning point is larger for TEAM2 than for TEAM, which is explained by the increased range of the outliers. The median turning point indicates that the TEAM2 best-response learning mechanism is more efficient than the TEAM online evolutionary learning mechanism, as the adaptation to successful behaviour progresses more swiftly than before; expressed in time only 48 minutes are required (as compared to 69 minutes).

Therefore, we may draw the conclusion that the TEAM2 mechanism exceeds the applicability of the TEAM mechanism for the purpose of learning in games. The qualitative acceptability of the performance is discussed next.

## 5 Discussion

Our experimental results show that the TEAM2 mechanism succeeded in enhancing the learning performance of the TEAM mechanism with regard to its median, but not mean, efficiency. In sub-section 5.1 we give a comparison of the learned behaviour of both mechanisms. Sub-section 5.2 discusses the task of online learning in a commercial computer game environment with regard to the observed outliers.

### 5.1 Comparison of the Behaviour Learned by TEAM and TEAM2

In the original TEAM experiment we observed that the adaptive team would learn so-called "rush" tactics. Rush tactics aim at quickly obtaining offensive field supremacy. We noted that the QUAKE III team AI, as is was designed by the QUAKE III developers, uses only moderate tactics in all states, and therefore, it is not able to counter *any* field supremacy.

The TEAM2 mechanism is inclined to learn rush tactics as well. Notably, the experiment showed that if the adaptive team uses tactics that are slightly more offensive than the non-adaptive team, it is already able to significantly outperform the opponent. Besides the fact that the QUAKE III team AI cannot adapt to superior player tactics (whereas an adaptive mechanism can), it is not sufficiently fine-tuned; for it implements an obvious and easily detectable local-optimum.

### 5.2 Exploitation versus Exploration

In our experimental results we noticed that the exploitative TEAM2 mechanism obtained a significant difference between the relatively low median and relatively high mean performance, whereas the original, less exploitative, TEAM mechanism obtained a moderate difference between the median and mean performance. This difference is illustrated in figure 5. It reveals that the exploitative TEAM2 mechanism obtained a significant difference between the relatively low median and relatively high mean performance, whereas the original, less exploitative, TEAM mechanism obtained a moderate difference between the median and mean performance.

An analysis of the phenomenon revealed that it is due to a well-known dilemma in machine learning [Carmel and Markovitch, 1997]: the exploitation versus exploration dilemma. This dilemma entails that a learning mechanism requires the exploration of derived results to yield successful behaviour in the future, whereas at the same time the mechanism needs to directly exploit the derived results to yield successful behaviour in the present. Acknowledging the need for an enhanced efficiency, the emphasis of the TEAM2 mechanism lies on exploiting the data represented in a small amount of samples.

In the highly non-deterministic QUAKE III environment, a long run of fitness values may occur that, due to chance, is not representative for the quality of the tactic employed. Obviously, this problem results from the emphasis on exploiting the small samples taken from the distribution of all states. To increase the number of samples, an exploration mechanism can be added. The TEAM online evolutionary learning mechanism employed such an exploration mechanism with a fitness propagation technique, which led to loss of efficiency. We tested several exploration mechanisms in TEAM2, which we found also led to loss of efficiency. However, since it is impossible to rule out chance runs completely, an online learning mechanism must be balanced between an exploitative and explorative emphasis.

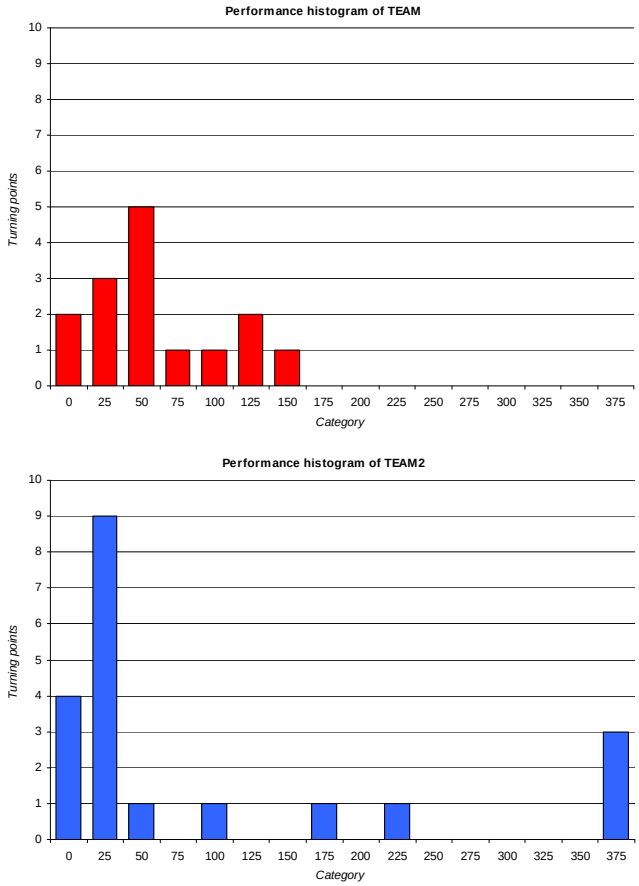


Figure 5: Histograms of the results of both the TEAM2 and TEAM experiment. The graphs show the number of turning points as a function of the value of the turning point, grouped by a category value of 25.

## 6 Conclusions and Future Work

The TEAM2 mechanism was proposed as an enhancement to the novel Tactics Evolutionary Adaptability Mechanism (TEAM), designed to impose adaptive behaviour on opponents in team-oriented games. The original TEAM mechanism is capable of unsupervised and intelligent adaptation to the environment, yet, its efficiency is modest. From the experimental results of the best-response learning experiment, we drew the conclusion that the TEAM2 best-response learning mechanism succeeded in enhancing the median, but not mean, learning performance. This reveals that in the current experimental setup the exploitation and exploration are not sufficiently well balanced to allow efficient and effective online learning in an actual game. As the TEAM2 mechanism is easily able to defeat a non-adaptive opponent, we may therefore conclude that the mechanism is suitable for online learning in an actual game if, and only if, a balance between exploitation and exploration is found for that specific game. Moreover, the TEAM2 mechanism can be used during game development practice to automatically validate and produce

AI that is not limited by a designer's vision.

Future research should investigate how an effective balance between exploitation of historic data and exploration of alternatives can be achieved. We propose to create a data store of gameplay experiences relevant to decision making processes, and use it to build an opponent model. Thereupon, game AI can either predict the effect of actions it is about to execute, or explore a more creative course of action.

## Acknowledgements

This research was funded by a grant from the Netherlands Organization for Scientific Research (NWO grant No 612.066.406).

## References

[Bakkes *et al.*, 2004] Sander Bakkes, Pieter Spronck, and Eric Postma. TEAM: The Team-oriented Evolutionary Adaptability Mechanism. In Matthias Rauterberg, editor, *Entertainment Computing - ICEC 2004*, volume 3166 of *Lecture Notes in Computer Science*, pages 273–282. Springer-Verlag, September 2004.

[Carmel and Markovitch, 1997] David Carmel and Shaul Markovitch. Exploration and adaptation in multiagent systems: A model-based approach. In *Proceedings of The Fifteenth International Joint Conference for Artificial Intelligence*, pages 606–611, Nagoya, Japan, 1997.

[Laird, 2000] John E. Laird. Bridging the gap between developers & researchers. *Game Developers Magazine*, Vol 8, August 2000.

[Nolfi and Floreano, 2000] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*. MIT Press, 2000. ISBN 0-262-14070-5.

[Spronck, 2005] Pieter Spronck. *Adaptive Game AI*. PhD thesis, SIKS Dissertation Series No. 2005-06, Universiteit Maastricht (IKAT), The Netherlands, 2005.

[van den Herik *et al.*, 2005] Jaap van den Herik, Jeroen Donkers, and Pieter Spronck. Opponent modelling and commercial games. Universiteit Maastricht (IKAT), The Netherlands, 2005.

[van Rijswijck, 2003] Jack van Rijswijck. Learning goals in sports games. Department of Computing Science, University of Alberta, Canada, 2003.

[van Waveren and Rothkrantz, 2001] Jean-Paul van Waveren and Leon Rothkrantz. Artificial player for Quake III Arena. In Norman Gough Quasim Mehdi and David Al-Dabass, editors, *Proceedings of the 2nd International Conference on Intelligent Games and Simulation GAME-ON 2001*, pages 48–55. SCS Europe Bvba., 2001.

# OASIS: An Open AI Standard Interface Specification to Support Reasoning, Representation and Learning in Computer Games

Clemens N. Berndt, Ian Watson & Hans Guesgen
*University of Auckland*
*Dept. of Computer Science*
*New Zealand*
*clemens.berndt@gmail.com, {ian, hans}@cs.auckland.ac.nz*

## Abstract

*Representing knowledge in computer games in such a way that reasoning about the knowledge and learning new knowledge, whilst integrating easily with the game is a complex task. Once the task is achieved for one game, it has to be tackled again from scratch for another game, since there are no standards for interfacing an AI engine with a computer game. In this paper, we propose an Open AI Standard Interface Specification (OASIS) that is aimed at helping the integration of AI and computer games.*

## 1. Introduction

Simulations with larger numbers of human participants have been shown to be useful in studying and creating human-level AI for complex and dynamic environments [Jones, et al. 1999]. The premises of interactive computer games as a comparable platform for AI research have been discussed and explored in several papers [Laird, & Duchi, 2000; Laird & van Lent, 2001]. A specific example would be *Flight Gears*, a game similar to Microsoft's Flight Simulator, that has been used for research into agents for piloting autonomous aircraft [Summers, et al. 2002].

If interactive computer games represent a great research opportunity why is it that we still see so comparatively little research being conducted with commercial grade games? Why is most AI research confined to games of the FPS genre and the mostly less complex open-source games? We believe that the single most important reason for this phenomenon is the absence of an open standard interface specification for AI in interactive computer games.

## 2. Standard Interfaces

Standard interfaces allow a piece of software to expose functionality through a common communication model that is shared amongst different implementations with equivalent or related functionality. The advantage of a common communication model is that other software may request similar services from different programs without being aware of a vendor specific implementation. The usefulness of standard interfaces has been widely acknowledged and found widespread application in many computing disciplines and especially within the software engineering community.

Successful examples of open standard interfaces in the industry are plentiful. They include TCP/IP, DOTNET CLI, XML Web Services, CORBA, SQL, ODBC, OpenGL, DirectX, the Java VM specifications and many others. We suggest that applying the same principle to AI in computer games would significantly reduce the effort involved in interfacing AI tools with different games. In the absence of a common communication model for interacting with the virtual worlds of computer games, AI researchers have to concern themselves with implementation specifics of every game they would like to interface with. This usually entails a significant amount of work especially with closed source commercial games that do not expose a proprietary mod interface.

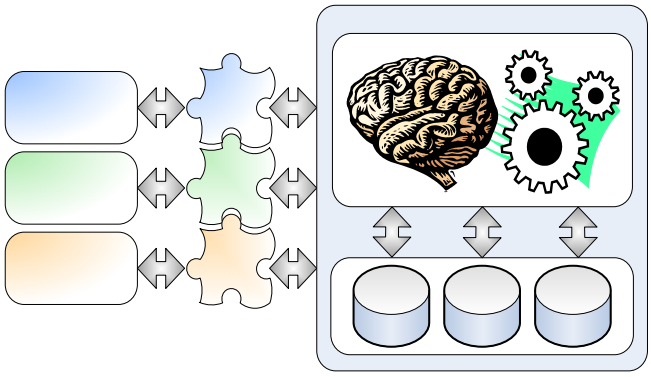

**Fig. 1 Non-Standard Game AI Interfaces**

Games in the FPS segment have been a leader in implementing proprietary mod interfaces to encourage third parties to develop mods (i.e. modification or extensions) to their games. These interfaces significantly reduce the effort required to build custom extensions to the original game engine. As a result of this FPS games have been used as a platform for AI research [Laird, 2000; Khoo & Zubek, 2002; Gordon & Logan, 2004]. Application of similar interfaces to real time strategy games has been suggested by some researchers [van Lent, et al. 2004]. Others have suggested game engine interfaces for supporting particular

areas of AI research such as machine learning [Aha & Molineaux, 2004].

However, proprietary mod interfaces, whilst having the potential to significantly reduce effort when working with a particular game, do not provide AI developers with the benefits associated with an open framework of standard interfaces. An AI mod created for one game will still have to be fitted with an additional interface module to be able to support another game (Fig. 1). This makes it difficult for AI researchers to validate their work across different computer games.

Rather than implementing non-standard interfaces for each and every computer game in the market, we believe it would be useful to create a set of open standard interface specification that are applicable to computer games of all genres. In addition an open standard interface specification for game AI would also have the potential of commercial success as it could provide a means of both reducing AI development costs by acting a guideline and boosting game popularity through third party add-ons while allowing intellectual property to be protected.

In brief, we are pursuing the following goals:

Simplicity: The interface specification should be simple, yet powerful and flexible enough to cover the various aspects of AI associated with computer games.

Extensibility: Modularity should be a core requirement, so that further functionality can easily be added to the framework as necessary.

Encapsulation: The interface specification should consist of layers that provide access to the game engine with an increasing degree of abstraction.

Cohesion: There should be a clear separation of function and logic. Each component of the framework should either play a functional (e.g. symbol mapping) or a logical role (e.g. plan generation), but not both.

## 3. Related Work

Researchers active in different areas of AI have long realised the importance of developing and employing standards that alleviate some of the difficulties of interfacing their research work with its area of application. Even though work in this area has led to advances in providing standardised tools for AI researchers and developers little work has been done in the area of standard interfaces. One of the main reasons for this is probably the heterogeneous nature of AI research. Computer games, however, represent a comparatively homogenous area of application and thus may see a more profound impact from standard interface specifications.

Past standardisation efforts in the area of AI can be roughly grouped into two categories:

1. work on standard communication formats and,
2. the development of standard AI architectures.

The development of standard communication formats is occupied primarily with the standardisation of expressive representational formats that enable AI systems and tools to flexibly interchange information. Work in this category encompasses standards such as KIF [Genesereth & Fikes, 1992] and PPDL [McDermott, et al., 1998]. We will refer to efforts in this category as *information centric standards*. Although information centric standards play a crucial part in the communication model of a standard interface specification they by themselves are not a replacement for such a framework. In addition, most of the information centric standardisation work in the past, whilst being well suited for most areas of AI, does not meet the performance requirements of computer games.

The second category of work comprises the creation of *architecture standards* for AI tools. Successful examples of such standard architectures are SOAR [Tambe, et al., 1995] and more recently TIELT, the Testbed for Integrating and Evaluating Learning Techniques [Aha, & Molineaux, 2004]. Architecture standards are similar to standard interface specifications in the sense that they involve similar issues and principles and both represent service centric standards. As such architectures like TIELT signify a cornerstone in reducing the burden on AI researchers to evaluate their AI methods against multiple areas of application through the interface provided by the TIELT architecture.

However, standard architectures cannot achieve the same degree of flexibility and interoperability as an open standard interface specification. The ultimate difference between something like TIELT and a standard interface specification is that a standard architecture functions as middle-ware. As such it is not directly part of the application providing the actual service, but acts as translation layer. Therefore it in turn must interface with each and every game engine that it is capable of supporting, just like an AI engine had to previously be interfaced with every game that it should be used with. This solution in its very nature only pushes the responsibilities of creating the actual interface to a different component – the underlying issue, however, remains unsolved.

The need for both researchers and developers to address the discussed issues with the present state of game AI and their current solutions has been indicated by the recent emergence of the game developer community's own efforts. These efforts, organised by the IDGA AI Special Interest Group through the game developer conference round table attempt to make some progress on AI interface standards for computer games. The IDGA AI SIG has established a committee with members from both industry and academia to accelerate this process. However, at the time of writing these efforts were still at a conceptual level and had not yet resulted in any experimental results.

# 4. OASIS Architecture Design

## 4.1 OASIS Concepts and Overview

Standard interfaces become powerful only when they are widely implemented by industry and other non-commercial projects. OpenGL and DirectX would be conceptually interesting, but fairly useless standard interface frameworks, if video card developers had not implemented them in practically all 3D acceleration hardware on the market. The OSI networking model on the other hand is an example of an academic conceptual pipe-dream. Viewed purely from an interface point of view, the OSI networking model is an incredibly flexible design that was in its original specification already capable of delivering much of the functionality that is nowadays being patched on to the TCP/IP networking model. However, the OSI networking model remains a teaching tool because it is too complicated to be practicable. Firstly, the process of arriving at some consensus was overly time-consuming because it involved a very large international task force with members from both academia and industry and attempted to address too many issues at once. Secondly, when the standard was finally released, it was prohibitively expensive for hardware manufacturers to build OSI compliant devices, especially in the low budget market segments. In comparison the TCP/IP model was developed by a much smaller group of people, is much simpler, and although failing to address several problems, it is the most dominant networking standard today [Forouzan, 2000].

Despite the shortcomings of TCP/IP, there is an easy explanation for its success; TCP/IP is simple, yet modular and extensible. The focus of TCP/IP is on necessity and efficiency rather than abundance of features. This is what makes it a successful standard interface. Thus we believe a standard interface for AI in games should be modular, extensible and simple while still fulfilling all core requirements. An Open AI Standard Interface Specification (OASIS) should feature a layered model that offers different levels of encapsulation at various layers, allowing interfacing AI modules to choose a mix between performance and ease of implementation adequate for the task at hand. Thus the lower layers of OASIS should provide access to the raw information exposed by the game engine, leaving the onus of processing to the AI module, while higher layers should offer knowledge level [Newell, 1982] services and information, freeing the AI developer from re-implementing common AI engine functionality.

We suggest there be a small number of layers in the OASIS framework. Each layer ought to be highly cohesive; that is, every layer, by itself, should have as few responsibilities as possible besides its core functionality and be completely independent of layers above it, thus allowing layer based compliance with the OASIS framework. This permits game developers to implement the OASIS specifications only up to a certain layer.

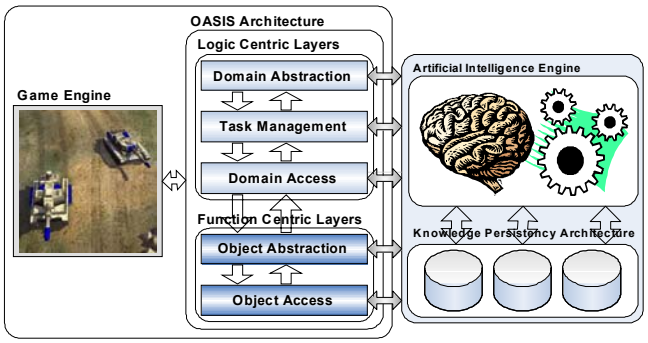


**Fig. 2 OSASIS Architecture**

This is especially useful when some higher layer functionality is either unnecessary due to simplicity of the game or because resource constraints imposed by computationally intensive games would not permit the use of layers with greater performance penalties without seriously impacting playability. Such implementation flexibility reduces both financial and time pressure on developers to comply with all specifications of the OASIS framework. Since modularity is a core feature of OASIS, compliance can be developed incrementally. This is not only good software engineering practice, but would allow game developers to provide patches after a game's release to add further OASIS compliance.

As a prototype design for the OASIS framework we have conceived a simple five layer architecture comprising:

1. an *object access* layer for direct manipulation of the game engine,
2. an *object abstraction* layer to hide runtime details from higher layers,
3. a *domain access* layer to expose the game domain in a form accessible to reasoning tools,
4. a *task management* layer providing goal arbitration and planning services, and
5. a *domain abstraction* layer that hides the complexity of the underlying game engine domain from more generic AI tools.

The bottom two layers of the architecture (i.e., 1 & 2) are function centric; that is, they are concerned mainly with the runtime specifics of single objects implemented in the game engine. In contrast the top three layers of the OASIS architecture (i.e., 3, 4 & 5) would be knowledge centric and hence would be concerned with manipulation of the domain at the knowledge level and are not directly interacting with single run-time objects. Note, that different from middle-ware architectures such as TIELT or SOAR, the OASIS framework is actually a set of specifications rather then a piece of software. The actual implementation details of the OASIS architecture should not matter as long as the interface specifications are complied with. This design makes the AI engine of a game a separate and readily interchangeable component.

The following sections discuss the suggested

functionality and responsibilities for each of the OASIS layers and their respective components depicted in Figure 2. All of this represents our initial ideas on how the OASIS architecture design could be structured and what features it might need to posses and should be considered neither final nor complete.

### 4.2 Object Access Layer

The access layer directly exposes objects defined in the game engine that may be manipulated by an interfacing AI engine. Objects exposed by the object access layer include everything from the tangible parts of the game environment such as an infantry unit to more abstract components such as the game state. For every object, the access layer specifies properties, operations and events that may be used to interact with the corresponding object in the game engine.

At the object access layer speed should be the main concern. Here the metadata should define information not observable from the signature of the object's operations and events such as preconditions, post conditions, extended effects and duration in terms of low-level descriptors. While this is computationally efficient processing is required before the information provided at this layer can be used to establish the semantics of the objects. In order to not impair performance each object would be a lightweight wrapper around its counterpart in the game engine, simply passing on the received messages with little or no intermediate processing (Fig. 2).

### 4.3 Object Abstraction Layer

The object abstraction layer provides framing of the resources provided by the object access layer into more readily usable structures. The function of the object abstraction layer is three fold, it manages all aspects of object assemblies, it orchestrates objects and assemblies to perform tasks and it compiles metadata from the data access layer into object semantics that define the logical relations between both objects in the game world exposed by the object access layer and object assemblies derived from those objects.

Object assemblies are essentially groupings of game objects with additional properties and functions that allow viewing and manipulating the underlying objects as a single unit. These groupings should be allowed to be very flexible for example it should be possible for an interfacing AI engine to define all objects of a specific type as an object assembly. Object assemblies themselves should in turn permit aggregation thus providing for recursive hierarchies of object assemblies. After creating a new assembly, the interfacing AI engine might then specify additional properties and operations that are not defined by the underlying objects thus making the game engine programmable without requiring access to the source code, which often represents a problem with commercial games. Since the behaviour and execution steps of user created properties and operations need to be explicitly specified some kind of high-level programming language must be part of this layer's protocol suite.

Another function of this layer is compiling the object metadata retrieved from the lower layer into logical relations between objects that are directly usable for example by an execution monitor to verify the progress of a plan and recognise its failure or success. These object semantics should also cover any object assemblies created by the user. This might necessitate the specification of metadata for object assemblies by the user if the metadata of the assemblies' components is insufficient to automatically derive the semantics of the assembly.

Lastly, the object abstraction layer is responsible for object orchestration. This means that it verifies the validity of execution of operations for both objects and assemblies and informs higher layers of invalid requests. It also deals with any runtime concurrency issues and processes events received from the object abstraction layer into a semantic format that may be used by higher layers for reasoning. This should effectively insulate the function centric from the logic centric layers of the OASIS framework.

The protocol suit required for communication with this layer would probably need to be more diverse than that of the object access layer. There are two main issues that need to be addressed. Firstly, fast access to the functions of the object access layer to allow for manipulating objects and assemblies. Secondly, capabilities for creating and programming of object assemblies. Although the focus of protocols at this layer should be to provide more abstraction, speed and lightweight remain a core requirement.

### 4.4 Domain Access Layer

The domain access layer provides a high-level abstraction of the game engine. This includes task execution management and domain description services. Task execution management is concerned with the execution of the logical steps of a plan specified in some expressive standard high level format. The task execution manager functions much like an execution monitor for planners. It translates the high level logical steps of a plan into an instruction format understood by the object abstraction layer, negotiates conflicts, monitors the execution results and informs higher layers of irresolvable conflicts and illegal instructions. The steps it executes may either manipulate objects within the domain (e.g. move tank X behind group of trees Y) or the domain description itself by creating or manipulating object assemblies in the object abstraction layer (e.g. add average unit life time property to infantry type assembly). Concurrency issues between competing plans executed in parallel need to be also managed at this layer. In order to reduce overhead this should occur as transparent as possible only making the AI engine aware of conflicts that are irresolvable.

The domain description component of this layer addresses two separate issues. First, it describes the semantics and mechanics of the domain created by the game engine in a standard high level knowledge representation. Second, it is directly usable by planners and other AI reasoning tools. The domain description provided should

include both native game objects and user created object assemblies. The other task of the domain description is to communicate to any interfacing AI engine the current state of the objects defined in the game world and any changes thereof.

The protocols used to communicate with this layer are fairly high level in terms of the information content they portrait. Optimally, the domain access layer should be able to support different formats for specifying plans to the task execution manager, so that AI engines using different types of AI tools may directly interface with this layer. In terms of protocols the domain description component is probably the most complex to address in this layer since it should allow a variety of AI tools to be able to directly interface with it. The domain description needs to be probably communicated in a variety of standards such as the planning domain description language developed for the 1998/2000 international planning competitions [McDermott, et al. 1998]. One of the major challenges posed by the protocol suite at this layer is to minimize the number of standards that have to be supported by default without limiting the nature of the AI tools interfacing to this layer. This could potentially be achieved by providing support for certain popular standards, while making the protocol suit pluggable and allowing third parties to create their own plug-ins to communicate with this layer. However, the feasibility of such an approach would need to be studied.

### 4.4 Task Management Layer

The domain abstraction layer, unlike all of the other layers, would not primarily serve the purpose of hiding the complexity of the lower layers from the layers above, but rather the provision of services that form an extension to the functionality of the domain access layer. Therefore some functions of the domain abstraction layer will not require the services provided at this layer. Thus in some cases this layer would be transparent to the top layer and simply pass through requests to the domain access layer without any further processing. Overall this layer should provide planning related services such as, plan generation, heuristic definition and goal management.

The plan generation capability of this layer is probably the single most important service offered here. It provides planning capabilities to the top layer as well as AI engines that do not posses the required planning capabilities to interact directly with the domain access layer. The plan generation component of the task management layer outputs a plan that is optimised using any heuristics given by the user and achieves the specified goals. This output plan is fed to the task execution management component in the layer below for processing and execution. The plan generation should be implemented very modular allowing third parties to create pluggable extensions to this functionality to adjoin different planning architectures to the OASIS framework that might not have been part of it originally. This would have two effects. First, this would enable AI researchers to verify, test and benchmark new planning architectures using OASIS. Second, it would provide an easy way to complement the set of the OASIS planners should there be shortcomings for certain kind of domains without needing to release a new version of the OASIS specifications.

Heuristic definition and goal management complement this planning capability. They allow AI engines to specify goals to be achieved and heuristics to be honoured by the planning component. The AI engine should be able to specify these in terms of symbols from the domain description provided by the domain access layer. The user should be permitted to prioritise goals and mark them as either hard goals that must be attained or soft goals that may be compromised. A planner in this layer should be allowed to re-shuffle the order of soft goals as long it does not increase the overall risk of failure. Any heuristics supplied by the AI engine are then applied to create a plan that will satisfy all hard goals and as many soft goals as possible.

Communication at this layer should use high level protocols describing both heuristics and goals in terms of the symbols found in the domain description at the layer below so that the planner does not need to support any additional mapping capabilities and may operate pretty much on the raw input provided. Excluding mapping and transformation capabilities from the task management layer will most definitely have a positive impact on performance.

### 4.5 Domain Abstraction Layer

The domain abstraction layer represents the top of the OASIS layer hierarchy and hence provides the greatest degree of abstraction from the implementation details of the game engine and the lower layers of the OASIS framework. High level functions such as domain model adaptation services, the domain ontology and task management services will be rooted at this layer. The main aim of this layer is to provide a knowledge level access point for AI reasoning tools that are either very limited in their low level capabilities or highly generic in their application. The interfaces provided by the domain abstraction layer and its components are not primarily geared towards speed, but much more towards interoperability and high level problem representation and resolution.

The domain model adaptation service provided here plays an important role in bridging the gap to generic reasoning tools and agents that are designed to handle certain tasks within a particular problem domain such as choosing what unit to add next to a production queue. Such problem description is very generic and will occur in slightly different variants in many games, especially in the real time strategy genre. Domain model adaptation will allow symbols of the domain defined by the game engine to be mapped to semantically equivalent symbols of the agent's domain model. In this way the agent can continue to reason in the confines of his own generic view of the world and is at the same time able to communicate with the game engine using expressions built from its own set of domain symbols. In order to facilitate this translation the domain model adaptation module would have rely on the ontology services provided by this layer and might in certain cases

require the interfacing AI engine to explicitly specify certain mappings. The domain model adaptation component is probably going to be by far the most complex and least understood component in the entire OASIS architecture. This is because domain model adaptation is still mainly a research topic although there are a few successful practical applications [Guarino et al. 1997].

The purpose of the ontology component of this layer is to provide a semantically correct and complete ontology of the symbols found in the domain description of the underlying game. Although fairly straight forward this could prove time intensive for developers to implement because it almost certainly requires human input to create a useful and comprehensive ontology for the game being built. Creating a standardised ontology for similar games and genres will be a key to successful implementation of this feature.

The second major service is task management. This involves facilitating the specification of very high-level tasks in terms of the elements contained in the ontology exposed at this layer and their completion using the functions provided by lower layers. Such task might in terms of logic resemble assertions like *"(capture red flag OR kill all enemy units) AND minimize casualties)*". The task management component would have to take such a task description and transform it into a set of goals and heuristics that may then be passed on to the task management layer. In order to extract goals, heuristics, priorities, etc. from the high-level task description, the interfacing AI engine would be required to flag the description's components. The task management component should also be responsible for tracking task progress and inform the AI engine of completion or failure. Concurrency issues of any kind and nature arising from competing tasks being executed in parallel should be handled by the lower layers.

## 5. Conclusion

Obviously there is still much uncertainty about the exact details of the OASIS framework and there are many issues that this paper has left unsolved. In the future it would probably be valuable to survey, document and analyse in detail the requirements of both game developers and AI researchers to form the basis of a formal requirements analysis. This would provide a better understanding of the problems being addressed and support a better set of design specifications for the OASIS framework. In the immediate future we will take advantage of the modularity and extensibility requirement of OASIS and implement a vertical prototype as a proof of concept. During this process we will also explore the usefulness and feasibility of some of the proposals made by the Artificial Intelligence Interface Standards Committee (AIISC) of the IDGA AI SIG. Potentially, a small number of diversified vertical prototypes might help us gain a more accurate understanding of the requirements for the framework that could form the stepping stone for further work in this area. We would also seek input and comments from other researchers and developers working in this area.

## References


Aha, D. and Molineaux, M. 2004, Integrating Learning in Interactive Gaming Simulators, Challenges in Game Artificial Intelligence – Papers from the AAAI Workshop Technical Report WS-04-04

Forouzan, B. 2000, Data Communications and Networking 2nd Edition, McGraw-Hill

Gordon, E. and Logan, B. 2004, Game Over: You have been beaten by a GRUE, Challenges in Game Artificial Intelligence – Papers from the AAAI Workshop Technical Report WS-04-04

Guarino N., et al. 1997, Logical Modelling of Product Knowledge: Towards a Well-Founded Semantics for STEP, In Proceedings of European Conference on Product Data Technology

International Game Developers Association – Special Interest Group on AI (IDGA – AI SIG), http://www.igda.org/ai/

Jones, R., et al. 1999, Automated Intelligent Pilots for Combat Flight Simulation, AI Magazine

Khoo, A. and Zubek R. 2002, Applying Inexpensive AI Techniques to Computer Games, In Proceedings of IEEE Intelligent Systems

Laird J. 2000, It knows what you're going to do: Adding anticipation to a quakebot, Artificial Intelligence and Interactive Entertainment – Papers from the AAAI Workshop Technical Report SS-00-02

Laird, J. and Duchi, J. 2000, Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot, In Proceedings of AAAI Fall Symposium: Simulating Human Agents

Laird, J. and van Lent, M. 2001, Human Level AI's Killer Application: Interactive Computer Games, AI Magazine Volume 2 – MIT Press

McDermott, D., et al. 1998, PDDL - The Planning Domain Definition Language, Yale Center for Computational Vision and Control - Technical Report CVC TR-98-003/DCS TR-1165

Newell, A. 1982, The Knowledge Level. Artificial Intelligence, 18 (1)

Summers, P., et al. 2002, Determination of Planetary Meteorology from Aerobot Flight Sensors, In Proceedings of 7th ESA Workshop on Advanced Space Technologies for Robotics and Automation

Tambe, M., et al. 1995, Intelligent Agents for Interactive Simulation Environments, AI Magazine

van Lent, M., et al. 2004, A Tactical and Strategic AI Interface for Real-Time Strategy Games, Challenges in Game Artificial Intelligence – Papers from the AAAI Workshop Technical Report WS-04-04

Genesereth, M. and Fikes, R. 1992**,** Knowledge Interchange Format, Version 3.0 Reference Manual*,* Technical Report Logic-92-1 – Computer Science Department Stanford University

# Colored Trails: A Formalism for Investigating Decision-making in Strategic Environments

**Ya'akov Gal** and **Barbara J. Grosz**
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138

**Sarit Kraus**
Dept. of Computer Science
Bar Ilan University
Ramat Gan 52900
Israel

**Avi Pfeffer** and **Stuart Shieber**
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138

## Abstract

Colored Trails is a research testbed for analyzing decision-making strategies of individuals or of teams. It enables the development and testing of strategies for automated agents that operate in groups that include people as well as computer agents. The testbed is based on a conceptually simple but highly expressive game in which players, working individually or in teams, make decisions about how to deploy their resources to achieve their individual or team goals. The complexity of the setting may be increased along several dimensions by varying the system parameters. The game has direct analogues to real-world task settings, making it likely that results obtained using Colored Trails will transfer to other domains. We describe several studies carried out using the formalism, which investigated the effect of different social settings on the negotiation strategies of both people and computer agents. Using machine learning, results from some of these studies were used to train computer agents. These agents outperformed other computer agents that used traditional game theoretic reasoning to guide their behavior, showing that CT provides a better basis for the design of computer agents in these types of settings.

## 1 Introduction

As heterogeneous group activities of computer systems and people become more prevalent, it is important to understand the decision-making strategies people deploy when interacting with computer systems. Colored Trails (CT) is a test-bed for investigating the types of decision-making that arise in task settings where the key interactions are among goals (of individuals or of groups), tasks required to accomplish those goals, and resources. The CT architecture allows games to be played by groups comprising people, computer agents, or heterogeneous mixes of people and computers. The purpose of the CT framework is to enable to design, learn and evaluate players' decision-making behavior as well as group dynamics in settings of varying complexity.

The rules of CT are simple, abstracting from particular task domains, thus enabling investigators to focus on decision-making rather than the specification of domain knowledge. In this respect CT is similar to the games developed in behavioral economics [1]. However, unlike behavioral economics games, CT provides a clear analog to multi-agent task settings, can represent games that are larger in size, and provides situational contexts and interaction histories in which to make decisions.

The CT environment allows a wide range of games to be defined. Games may be made simple or complex along several dimensions including the number of players and size of the game; information about the environment available to different players; information about individual agents available publicly to all players, to subgroups, or only privately; the scoring rules for agents; the types of communication possible among agents; and the negotiation protocol between agents.

At the heart of CT is the ability of players to communicate with each other, enabling them to commit to and retract bargaining proposals and to exchange resources. The conditions of these exchanges, group dynamics and players' behavior towards others are some aspects that can be investigated in these types of settings.

### 1.1 Rules of the Game

CT is played by two or more players on a rectangular board of colored squares with a set of chips in colors chosen from the same palette as the squares. For each game of CT, any number of squares may be designated as the goal. Each player's piece is located initially in one of the non-goal squares, and each player is given a set of colored chips. A piece may be moved into an adjacent square, but only if the player turns in a chip of the same color as the square.

Players are removed from the game if they reach the goal state or have been dormant for a given number of moves, as specified by a game parameter. When all players have been removed, the game is declared over and each player's score is computed. The scoring function of a CT game can depend on the following criteria: the position of a player on the board; the number of chips the player possesses; the number of moves made by the player throughout the game; the score of other players in the game. It is possible to vary the extent to which the scoring function depends on any of these parameters.

The game controller makes available to each player a list of suggested paths to the goal that are displayed in a panel

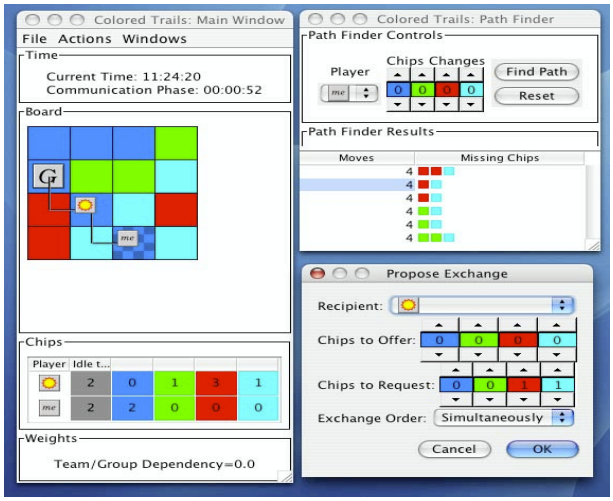


Figure 1: A snapshot of a two-player game

on the screen. These paths are optimized for a given chip distribution and player, as queried by the player, such that they represent the best route given a player's objectives. The ability to access this information is contingent on a player's ability to view the board and chips, as specified by the game parameters.

A snapshot of a two-player game is presented in Figure 1. Here, the Main Window panel shows the board game, player icons and the goal state, as well as the chip distribution for the players. In this game, both *me* and *sun* players lack chips to get to the goal. The *me* player has queried the Path Finder panel and has chosen a path, outlined on the board, for which it lacks a red and cyan chip. It is about to ask the *sun* player for these chips, using the Propose Exchange panel.

Players in CT negotiate with each other during specified communication phases. Each message has a list of fields containing the information embedded in the message. Messages may be of the following types.

1. Propose an exchange.
2. Commit to a proposal.
3. Retract a proposal (i.e., a previous commitment).
4. Request/suggest a path to the goal.
5. Send chips to a player.

Note that messages (1) through (4) pass information between players while message (5) transfers chips between players. By setting the game parameters, agreements reached during the communication phase may or may not be binding. For example, a player whose offer was accepted by another player may need to initiate the sending of chips should it[1] wish to fulfill its commitment.

### 1.2 Analogy with Task Domains

There is a correspondence between CT play and the planning and execution of tasks by a group of agents. Colors correspond to agent capabilities and skills required by tasks: an agent's possession of a chip of a certain color corresponds to having a skill available for use at a time; not all agents get all colors much as different agents have different capabilities and availability. Paths through the board correspond to performing complex tasks, the constituents of which are individual tasks requiring the skills of the corresponding color. Various kinds of requirements on goals and paths followed correspond to different types of group activities and collaborative tasks. For instance, the degree to which an agent's score depends on the performance of other agents may be used to distinguish collaborative teamwork from situations in which group members act in other ways. Also, requiring only that a certain number of agents get to a goal square might correspond to the need for agents to allocate tasks to subgroups or form coalitions to accomplish the actions in a recipe for a collaborative task. Varying the amount of the board an agent can "see" corresponds to varying information about task recipes or resource requirements.

In addition, the inter-dependence between players can be varied. For example, the scoring function may stipulate a *reward dependence* by having the scores of a player depend in some way on the scores of other agents. Second, a *task dependence* arises whenever players lack the chips they need to reach their goals and must depend on other players supplying those chips.

In this paper, we present several studies that were carried out using the CT framework at Harvard and Bar Ilan Universities. Each study is presented in increasing order of the complexity of the CT setting. All results are statistically significant in the 95% confidence interval range. Three types of players interacted in these studies: people, computer agents designed by the experimenters, and computer agents designed by human subjects.

Section 2 describes how a model of human behavior in one-shot games was devised and evaluated using a machine learning approach in a simple CT setting. Section 3 describes a model for negotiation between computer agents in a setting in which agents were uncertain about others' resources as well as their level of helpfulness. Section 4 outlines a study which investigated the effect of reward dependency on the behavior of people and of agents.

## 2 Learning Social Preferences in One-shot Games

Research in behavioral economics [1] has established that a multitude of sociological factors affect people's behavior when they interact with others. In particular, people have been shown to exhibit preferences for choices that affect others as well as themselves and to vary in the extent to which these factors affect their play. Traditional game-theoretic models cannot naturally capture the diversity of this behavior [5]. In a series of studies [3; 6], we showed that computer agents that explicitly represented social preferences in their model and learned their extent of influence on people's behavior were able to outperform traditional game-theoretic models. In particular, they were able to generalize to new situations in the game as well as to new players.

[1] we use gender neutral pronouns to refer to players in the game, be they computer agents or people

## 2.1 The CT set-up

We used a version of CT in which two players played on 4x4 boards with a palette of 4 colors. Each player had full view of the board as well as the other player's tiles. The distribution of tiles at the onset of the game was designed such that (1) at least one of the players could reach the goal after trading with the other player; (2) it was not the case that both players could reach the goal without trading.

The scoring function was chosen so that while getting to the goal was by far the most important component, if a player couldn't get to the goal it was preferable to get as close to the goal as possible. Furthermore, a player's outcome was determined solely by its own performance.

In each game, one player deemed the *allocator* was allowed to propose an offer for exchange of tiles to the other player, deemed the *deliberator*. The deliberator could either accept or reject the allocator's offer. If the allocator did not make an offer, then both players were left with their initial allocation of tiles. The deliberator was not allowed to counter the allocator's offer with another proposal. The score that each player received if no offer was made was identical to the score each player received if the offer was rejected by the deliberator. We referred to this event as the *no negotiation alternative*. The score that each player received if the offer was accepted by the deliberator was referred to as the *proposed outcome* score. Under the conditions specified above, each game consisted of a one-shot negotiation deal between the two players, and a deliberator's reply to the exchange proposed by the allocator completely determines the final outcome of the game.

## 2.2 Model Construction

Our model predicted whether the deliberator will accept a given proposal, given a CT game. The inputs to the model are $NN_A$ and $NN_D$, the no-negotiation alternative scores for the allocator and deliberator, and $PO_A$ and $PO_D$, the proposed outcome scores for the allocator and deliberator.

To develop the model, we introduced the following features, which represented possible social preferences that might affect the deliberator for a given deal. Each feature was derived using the no-negotiation alternative and proposed outcome scores.

- self interest $PO_D - NN_D$
- social welfare $(PO_D + PO_A) - (NN_D + NN_A)$
- advantageous inequality $PO_D - PO_A$
- fair trade $(PO_D - NN_D) - (PO_A - NN_A)$

Given any proposed exchange $x$, a particular deliberator's utility $u(x)$ is a weighted sum of these features. The weights measure the relative importance of each of the social preferences to the deliberator.

We captured the fact that people make mistakes by implementing a noisy decision function for the deliberator. We defined the probability of acceptance for a particular exchange $x$ by a deliberator as $P(accept \mid x, t) = \frac{1}{1+e^{-u(x)}}$. This probability converges to 1 as the utility from an exchange becomes large and positive, and to 0 as the utility becomes large and negative.

The model assumed that people reason about the same types of social factors, but that individuals weigh them differently. We used a mixture model over types of people, with a probability distribution $P(t)$ over the set of types. Each type $t$ was associated with its own set of social preference weights, defining a utility function $u_t$.

Given that we have a model describing the deliberator's behavior, the next step was to incorporate this model into a computer agent that played with humans. In our framework, the computer agent played the allocator and a human played the deliberator. The strategy of the allocator was to propose the deal that maximized its expected utility. The expected utility was the sum of the allocator's utility of the proposal times the probability the proposal is accepted, plus the allocator's no-negotiation alternative score times the probability the proposal is rejected. We took the expectation of this sum with respect to all of the deliberator utility functions.

To learn the parameters of the model, we estimated the distribution $P(T)$ over deliberator types, and for each type $t \in T$, we estimated the feature weights. We did this by interleaving two optimization procedures, a version of the EM algorithm [2] and the gradient descent technique. We began by placing an arbitrary distribution over deliberator types and setting the feature weights with particular parameter values.

## 2.3 Experimental Setup and Results

A total of 42 subjects participated in the experiment, 32 in the data-collection phase and 10 in the evaluation phase. Each phase was composed of playing a number of rounds of different games. A central server was responsible for matching up the participants at each round and for keeping the total score for each subject in all of the rounds of the experiment. Participants were paid in a manner consistent with the scoring function in the game. For example, a score of 130 points gained in a round earned a $1.30 payment. We kept a running score for each subject, revealed at the end of the experiment.

In the data-collection phase, 16 subjects played consecutive CT games against each other Each subject played 24 CT rounds, making for a total of 192 games played. The initial settings (board layout, tile distribution, goal and starting point positions) were different in each game. For each round of the game, we recorded the board and tile settings, as well as the proposal made by the allocator, and the response of the deliberator. The data obtained was then used to learn a mixture model of human play, which included 2 types with probabilities $(0.36, 0.64)$. The feature weights learned for each type were $(3.00, 5.13, 4.61, 0.46)$ and $(3.13, 4.95, 0.47, 3.30)$ for individual-benefit, aggregate-utility, advantage-of-outcome and advantage-of-trade. According to the learned model, both types assigned high weights for social welfare, while still being competitive; one of the types cares more about advantage-of-outcome, and the other type cares more about advantage-of-trade.

The evaluation study consisted of two groups, each involving 3 human subjects and 3 computer players. At each round, eight concurrent games of CT were played in which members of the same group played each other. One of the human subjects, designated as an allocator, played another human subject, designated as a deliberator; each computer player, des-

ignated as an allocator, played another human subject, designated as a deliberator.

The computer players, only playing allocators, were agents capable of mapping any CT game position to some proposed exchange. Agent $SP$ proposed the exchange with the highest expected utility, according to our learned social preferences model. Agent $NE$ proposed the exchange corresponding to the Nash equilibrium strategy for the allocator. Agent $NB$ proposed the exchange corresponding to the Nash bargaining strategy for the allocator, consisting of the exchange that maximized the product of each player's individual benefit.

The game settings, including board layout, start and goal positions, and initial tile distributions, were the same for all of the games played by members of the same group. Therefore, at each round there were 4 matching CT games being played by the eight members of each group.

The following table presents the results of the evaluation phase for each of the models used in the experiment.

| Model | Total Reward | Proposals Accepted | Proposals Declined | No Offers |
|---|---|---|---|---|
| $SP$ | 2880 | 16 | 5 | 0 |
| $NE$ | 2100 | 13 | 8 | 0 |
| $NB$ | 2400 | 14 | 2 | 5 |

It lists the total monetary reward, the number of proposals accepted, the number of proposals rejected, and the number of times no offer was proposed. The $SP$ agent had achieved a significantly higher utility than the other computer agents. It also had the highest number of accepted proposals, along with the allocations proposed by humans. The performance of $NE$ was the worst of the three. The computer allocator labeled $NE$ always proposed the exchange that corresponded to the allocator's strategy in the (unique) sub-game perfect Nash equilibrium of each CT game. This resulted to offering the best exchange for the allocator, out of the set of all of the exchanges that are not worse off to the deliberator. As a consequence, many of the exchanges proposed by this agent were declined. We hypothesize this was because they were not judged as fair by the human deliberator. This result closely follows the findings of behavioral game theory. The computer allocator labeled $NB$ consistently offered more to the deliberator than the $NE$ player did for the same game, when the board and tile distribution enabled it. Because $NB$ tended to offer quite favorable deals to the deliberator, they were accepted more than the other computer players, provided that an offer was made. but its overall reward was less than $SP$.

While we have focused on one particular game for practical reasons, the learned models we used were cast in terms of general social preferences, which did not depend on the specific features of the game and were shown to be exhibited by people in many types of interactions.

# 3 Modeling Agents' Helpfulness in Uncertain Environments

When agents depend on each other to achieve their goals, they need to cooperate in order to succeed, i.e. to perform actions that mutually benefit each other. In open systems, there is no central control for agents' design, and therefore others' willingness to cooperate is unknown. To establish cooperative relationships in such systems, agents must identify those that are helpful and reciprocate their behavior, while staying clear of those that are unhelpful. However, in open environments it is difficult to identify the degree of helpfulness of other agents based solely on their actions. This is further made difficult if agents constantly change their strategies.

In this work [7], we built a model which explicitly represented and reasoned about agents' level of helpfulness. The model characterized helpfulness along two dimensions: cooperation (the tendency to propose mutually beneficial exchanges of resources) and reliability (the tendency to fulfill commitments).

We used a version of CT in which two or four players played on boards of different sizes. Each player had knowledge of the scoring function and full view of the board but could not see the other player's chips. Agreements reached during the communication phase were not binding and thus agents could deceive each other by not fulfilling their commitments.

A player was declared "out-of-game" if it reached the goal state or if it stayed dormant for 3 moves, at which point its score was computed. Each player's outcome depended solely on its own performance.

## 3.1 Model Construction

We wanted the model to be able to generalize to environments which varied the number of players, the size of the board-game, and the task dependency between players. To do this, the model explicitly reasoned about others' level of helpfulness, rather than their utility functions.

We described agents' helpfulness along two dimensions with range $[0, 1)$.

- Cooperation $(c)$ - measured an agent's willingness to share resources with others in the game through initiating and agreeing to beneficial proposals.
- Reliability $(r)$ - measured agents' willingness to keep their commitments in the game through delivering the chips they had agreed to.

Given some action $a$, opponent $j$, and the state of the game $s$, an agent's utility function depended on the following features.

- The helpfulness measure of agent $i$, denoted $P_i$.
- Agent $i$'s estimate of the agent $j$'s helpfulness, denoted $P_j$. This was estimated as the fraction of times $j$ was cooperative and reliable when interacting with $i$ in the past, decayed by a discount factor.
- The expected value of taking action $a$ given the state of the environment $s$, denoted $EV_i(a \mid s)$. This quantity estimated the likelihood of getting to the goal, negatively correlated with the number of chips lacked by the agent.
- The expected cost of future ramifications of taking action $a$, denoted $EC_i(a)$. This function rewarded actions that were beneficial to agent $j$ and punished actions that reneged on commitments.

We constructed a utility function that was a linear combination of these features associated with weights that were

tuned empirically. Agents negotiated using this utility function at each communication phase in the game, by performing each action in the subset of actions that fulfilled the following conditions: there were no two actions in the subset that conflicted (for example, two exchange proposals that offered the same chips); the combined utility value for the agent from each action in the subset was highest compared to any other subset with non-conflicting actions. Using this utility function, agents' behavior was contingent on their perception of others, as well as their own helpfulness.

### 3.2 Experimental Design

We used two class of agents in our study. The first consisted of two types: Multiple-Personality (MP) and Single-Personality (SP) agents. Both MP and SP class agents use the model described earlier to make their decisions. However, the cooperation and reliability levels of an SP agent were constant, whereas an MP agent adopted different measures of cooperation and reliability for each personality type of its opponents based on a matching scheme, derived empirically. Both MP and SP agents were adaptive: they changed their behavior as a function of their estimate of others' helpfulness, given the history of their observations. However, the MP agent adopted a unique measure of helpfulness for each player, whereas the measure of helpfulness for the SP agent was constant.

Another class of agents was Peer-Designed (PD) agents, created by graduate-level computer science students at Bar Ilan University who were not given any explicit instructions regarding agents' strategies and reasoning processes.

We classified PD and SP agents as either "helpful" or "unhelpful". Helpful SP agents were those that engaged in cooperative exchanges more than 50% of the time and reneged on their commitments less than 20% of the time. We expected helpful agents to be able to realize opportunities for exchange with each other more often than unhelpful agents and to exceed them in performance, as measured by the score in the game. We also expected that in some cases, unhelpful agents would be able to take advantage of the vulnerability of those helpful agents that allow themselves to be exploited. We hypothesized that the MP agent would be able to identify and reciprocate helpful agents more quickly than SP or PD agents, while staying clear of agents that are unhelpful. As a result, the MP agent would perform better than all other agents in the game.

We evaluated the MP agent by playing a series of repeated games with the other agents in the systems. We allowed agents to update their model of others from game to game. Each agent's final outcome was the aggregate of its scores in all of the games it participated in.

In our experiment we executed 5,040 games, played in 1,080 rounds of three consecutive games each. The board games we used in each round varied the task dependency relationships between players. The players in each game included a MP agent, two SP agents, and one of the PD agents. Each group of four players played all possible task dependency roles, to control for any effect brought about by dependency relationships. Table 1 presents the average score for the MP agent when playing against helpful and unhelpful agents across all games. The scores reported in the table sum over the other players in the game.

| | MP agent | PD and SP agents |
|---|---|---|
| Helpful | 170.6 | 114.8 |
| Unhelpful | 142.5 | 98.2 |

Table 1: Average performance of MP agent against helpful/unhelpful agents (3 repeated games)

The average score achieved by the MP agent was significantly higher than all other agents, regardless of their level of helpfulness. Also, the MP agent's score when playing against helpful agents (170.6) was higher than its score when playing against unhelpful agents (142.5). Helpful agents also benefited from cooperating with the MP agent: their performance was significantly higher than their unhelpful counterparts (114.8 vs. 98.2).

Further investigation revealed that the MP agent engaged in cooperative exchanges with helpful agents significantly more often than the other agents, while the amount of time the MP agent remained idle when dealing with unhelpful agents was longer than the amount of time other agents remained idle.

Another hypothesis was that any group of agents would increase its overall social welfare when playing with an MP agent, because the MP agent would help them to realize more beneficial exchanges. To evaluate this claim, we ran a series of 2-player repeated games that included SP and PD type agents, but did not include MP agents, and compared it to the performance of each agent type after after including an MP agent in the group. The results are described in Figure 2. The performance of helpful and unhelpful agents increased significantly when interacting with the MP agent. As expected, this increase was more profound for helpful SP and PD agents.

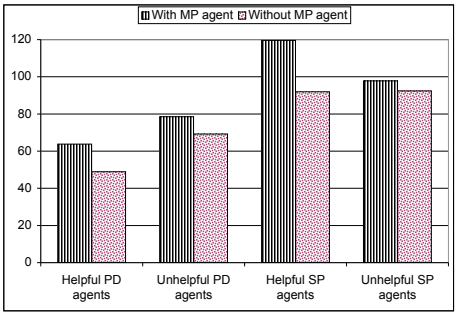


Figure 2: Performance with/without MP agent

| Exchange Type | Helpful agents | Unhelpful agents |
|---|---|---|
| Reciprocal | 60% | 25% |
| Idle | 20% | 39% |

Table 2: Percentage of exchange types proposed by MP agent

| | | # reached goal | Private score |
|---|---|---|---|
| People | $RD = 0$ | 2.47 | 151.3 |
| | $RD > 0$ | 2.8 | 252.34 |
| PD | $RD = 0$ | 1.1 | 82.4 |

Table 3: Results for people vs. PD agents

## 4 The Influence of Reward Dependencies on Decision Making

In the work presented here [4], we investigated the effect of social dependencies on players' behavior in CT. We varied the social dependency between players by including a "reward dependency factor" $RD$ in the scoring function; If $RD$ was zero, a player's score was independent of the performance of other players; if it was non-zero, a player's score was a combination of that player's individual score and a weighted average of the individual scores of all the other players.

We hypothesized that (1) higher $RD$ will increase the amount of cooperation between agents. In particular, we expected agents to give other agents chips more frequently and to ask for fewer chips in exchange when $RD$ is higher. (2) when $RD$ weight is high, agents will score higher and reach the goal more often.

The CT games were played by groups of four players, the board was 6x6, the palette was 5 colors, and there was a single goal square for all players. Players were able to see the full board but not each others' chips. This restriction separates decisions about chip exchanges from scoring information, thereby making helpful behavior distinct from score optimization computations.

Two classes of experiments were performed, one involving 4-player groups of people and the other two involving 4-player groups of computer agents. The human player groups were drawn from a population of upperclass and master's computer science students at Bar Ilan University who were not experts in negotiation strategies nor in economic theories directly relevant to agent design (e.g., game theory, decision theory). We compared their performance with that of Peer Designed (PD) agents, who were constructed in a similar fashion as described in Section 3.

A comparison of the results in Table 3 for human players when $RD = 0$ with those when $RD > 0$ supports this hypothesis. The average private score of all games played by people in which $RD > 0$ was significantly higher than in the games where $RD = 0$ In addition, the number of human players who reached the goal in games in which $RD > 0$ was significantly higher than for games with $RD = 0$ This shows that people realized more opportunities for exchange when their performance depended on others. Thus, the main hypotheses regarding $RD$ are supported by the results of games played by people. Interestingly, the PD designs were not influenced by the reward dependencies, and agents did not act significantly different in either condition. This suggests that implicit mention of reward-dependence in the design specification may not affect behavior. In contrast, this same incidental mention of $RD$ in instructions to people playing the game did engender different behavior as discussed below.

Another interesting discovery was that the average private score for people was significantly higher than the average private score of the PDs in these games. Furthermore, the average number of people reaching the goal in these games was significantly higher than the average number of PDs reaching the goal. Further investigation revealed that this was because people were significantly more likely to engage in cooperative exchanges.

## 5 Conclusion and Future Work

In this paper, we have motivated the need for understanding the decision-making strategies people deploy when computer systems are among the members of the groups in which they work. It has reviewed several studies, all using the CT framework, which analyzed the effects of various social settings on people's behavior, and built computer agents to match people's expectations in these settings.

In the future, we plan to extend the CT formalism in several realms. First, we are designing a system for evaluation of CT models, which would be able to dynamically configure dependent and independent variables, run a series of CT games using an online database of game configurations, game status, and results.

Second, we are constructing a model for repeated negotiation between players which reasons about the reputation of agents. This model will incorporate such features as reward and punishment, and the affinity of players to each other based on their actions. It will learn the extent to which these features affect decision making through incorporating observations of people's play.

## References


[1] C.F. Camerer. *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press, 2003.

[2] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1), 1977.

[3] Y. Gal, A. Pfeffer, F. Marzo, and B. Grosz. Learning social preferences in games. In *Proc. 19th National Conference on Artificial Intelligence (AAAI)*, 2004.

[4] B. Grosz, S. Kraus, S. Talman, and B. Stossel. The influence of social dependencies on decision-making. Initial investigations with a new game. In *Proc. 3rd International Joint Conference on Multi-agent systems (AAMAS)*, 2004.

[5] J.H. Kagel and A.E. Roth, editors. *The hanbook of experimental economics*. Princeton University Press, 1995.

[6] F. Marzo, Y. Gal, A. Pfeffer, and B. Grosz. Social preferences in relational contexts. In *IV Conference on Collective Intentionality*, 2004.

[7] S. Talman, Y. Gal, M. Hadad, and S. Kraus. Adapting to agents' personalities in negotiation. In *Proc. 4th International Joint Conference on Multi-agent systems (AAMAS)*, 2005.

# Unreal GOLOG Bots

**Stefan Jacobs** and **Alexander Ferrein** and **Gerhard Lakemeyer**
RWTH Aachen University
Computer Science Department
Ahornstr. 55, 52056 Aachen, Germany
{sjacobs,ferrein,gerhard}@cs.rwth-aachen.de

## Abstract

Even though reasoning and, in particular, planning techniques have had a long tradition in Artificial Intelligence, these have only recently been applied to interactive computer games. In this paper we propose the use of READYLOG, a variant of the logic-based action language GOLOG, to build game bots. The language combines features from classical programming languages with decision-theoretic planning. The feasibility of the approach is demonstrated by integrating READYLOG with the game UNREAL TOURNAMENT.

## 1 Introduction

Interactive computer games have come a long way since the introduction of *Pac-Man* many years ago. In particular, the availability of sophisticated graphics engines for regular PCs has made it possibly to create complex realistic 3D scenarios with multiple players controlled by either the computer or by humans. Recent examples of such games are HALF LIFE 2 [Valve Corporation, 2005], MEDAL OF HONOR [Electronic Arts Inc., 2005], or UNREAL TOURNAMENT [Epic Games Inc., 2005], which is the focus of this paper. What makes these games challenging for both humans and computers is their fast pace and the fact that a player usually has only incomplete or uncertain knowledge about the world.

Artificial Intelligence so far has had rather little impact on the development of computer-controlled players, also called game bots. In the commercial world, for example, simple and easy-to-use scripting languages are often preferred to specify games and agents [Berger, 2002], and finite state machines are a popular means to specify reactive behavior [Fu and Houlette, 2004]. To turn such agents into challenging opponents for humans, they are often given more knowledge about the game situation and more powerful capabilities than are available to the human player.

Nevertheless, interactive games and, in particular, UNREAL TOURNAMENT have caught the attention of AI recently. Kaminka et al. [Kaminka *et al.*, 2002] have proposed UNREAL TOURNAMENT 2004 as a framework for research in multi-agent systems.[1] Their own work has focused on areas like navigation, mapping and exploration. In [Munoz-Avila and Fisher, 2004], hierarchical planning techniques are used to devise long-term strategies for UNREAL TOURNAMENT bots. Recently, Magerko at al. [Magerko *et al.*, 2004] have connected the powerful rule-based system SOAR [Lewis, 1999] to Haunt II, an extension of UNREAL TOURNAMENT.

Our own work fits into this line of work on symbolic reasoning applied to UNREAL TOURNAMENT, with a focus on real-time decision-theoretic planning techniques.[2] In particular, we propose to use READYLOG [Ferrein *et al.*, 2004], a variant of the action language GOLOG [Levesque *et al.*, 1997], for the specification of game bots. GOLOG has been applied to control animated characters before by Funge [1998]. However, there the scenario was much simpler with complete knowledge about the world and no uncertainty.

Roughly, READYLOG is a high-level programming language with the usual constructs found in imperative programming languages and additional ones which allow agents to choose among alternative actions. Here, actions are understood as in AI planning, with effects and preconditions specified in a logical language. Built into the language is also a decision-theoretic planning component, which can deal with uncertainty in the form of stochastic actions. In general, decision-theoretic planning can be very time consuming. As we will see, combining it with explicit programming of behavior, the user can control the amount of planning so that it becomes feasible for real-time applications. We have demonstrated this already by using READYLOG to control soccer playing robots [Ferrein *et al.*, 2004], and we are using the same highly-optimized Prolog implementation for the work described in this paper.

The rest of the paper is organized as follows. In the next section we give a brief introduction to UNREAL TOURNAMENT 2004, followed by a very brief discussion of READYLOG and its foundations. In Section 4 we discuss how UNREAL TOURNAMENT can be modelled in READYLOG. We then present some experimental results and conclude.

[1] In the area of strategy games, Buro [Buro, 2003] is also developing an open-source game to serve as a testbed for AI techniques.

[2] Real-time planning is also considered in [Orkin, 2004], but without the decision-theoretic aspect.

## 2 UNREAL TOURNAMENT 2004

UNREAL II and UNREAL TOURNAMENT 2004 [Epic Games Inc., 2005] are two state-of-the-art interactive computer games. While the former is mainly a single-player game, the latter is a multi-player game. Here we focus on using and modifying the bot framework of UNREAL TOURNAMENT 2004 because the bots available therein are programmed to behave like human adversaries for training purposes.

The engine itself is mainly written in C++ and it cannot be modified. In contrast the complete Unreal Script (in the following USCRIPT) code controlling the engine is publicly available and modifiable for each game. For instance, introducing new kinds of game play like playing soccer in teams or the game of Tetris have been implemented on the basis of the Unreal Engine. All this can be defined easily in USCRIPT, a simple, object-oriented, Java-like language which is publicly available.

In UNREAL TOURNAMENT 2004 ten types of gameplay or game modes have been implemented and published. For our work the following game types are of interest:

- *Deathmatch* (DM) is a game type where each player is on its own and struggles with all other competitors for winning the game. The goal of the game is to score points. Scoring points is done by disabling competitors and secondary goal is not getting disabled oneself. If the player gets disabled he can choose to re-spawn[3] in a matter of seconds and start playing again. To be successful in this type of game one has to know the world, react quickly, and recognize the necessity to make a strategic withdrawal to recharge. An interesting subproblem here is the games where only two players or bots compete against each other in much smaller arenas. In this setting one can compare the fitness of different agents easily.
- *Team Deathmatch* (TDM) is a special kind of Deathmatch where two teams compete against each other in winning the game with the same winning conditions as in Deathmatch. This is the most basic game type where team work is necessary to be successful. Protecting teammates or cooperating with them to disable competitors of the other team are examples of fundamental strategies.
- *Capture the Flag* (CTF) is a strategical type of game play. The game is played by two teams. Both teams try to hijack the flag of the other team to score points. Each flag is located in the team base. In this base the team members start playing. Scoring points is done by taking the opposing team's flag and touching the own base with it while the own flag is located there. If the own flag is not at the home base no scoring is possible and the flag has to be recaptured first. If a player is disabled while carrying the flag he drops it and if it is touched by a player of an opponent team, the flag is carried further to the opponents home base. If the flag is touched by a teammate who owns the flag it is teleported back to its base.

  To win such a game the players of a team have to cooperate, to delegate offensive or defensive tasks, and to communicate with each other. This game type is the first one which rewards strategic defense and coordinated offense maneuvers.

Note that the above game types include similar tasks. A bot being able to play Team Deathmatch has to be able to play Deathmatch just in case a one-on-one situation arises. Furthermore Capture the Flag depends on team play just like the Team Deathmatch.

## 3 READYLOG

READYLOG is an extension of the action language GOLOG, which in turn is based on the situation calculus. We will briefly look at all three in turn.

The situation calculus, originally proposed in [McCarthy, 1963], is a dialect of first-order logic intended to represent dynamically changing worlds. Here we use the second-order variant of the situation calculus proposed by Reiter [Reiter, 2001].[4] In this language the world is seen as a sequence of *situations* connected by actions. Starting in some initial situation called $S_0$, each situation is characterized by the sequence of actions that lead to it from $S_0$. Given a situation $s$, the situation which is reached after performing an action $a$ is denoted as $do(a, s)$. Situations are described by so-called *relational* and *functional fluents*, which are logical predicates and functions, respectively, that have as their last argument a situation term.

Dynamic worlds are described by so-called *basic action theories*. From a user's point of view their main ingredients are a description of the initial situation, action precondition axioms and successor state axioms, which describe when an action is executable and what the value of a fluent is after performing an action. For example,

$$Poss(moveto(r, x), s) \equiv HasEnergy(r, s)^{5}$$

may be read as agent $r$ can move to position $x$ if it has enough energy. A simple successor-state axiom for the location of $r$ could be

$$loc(r, do(a, s)) = x \equiv a = moveto(r, x) \vee loc(r, s) = x \wedge \neg(moveto(r, y) \wedge x \neq y)$$

Roughly, this says that the location of $r$ is $x$ after some action just in case the action was a move-$r$-to-$x$ action or $r$ was already there and did not go anywhere else. We remark that successor state axioms encode both effect and frame problems and were introduced by Reiter as a solution to the frame problem [Reiter, 2001]. Furthermore, a large class of precondition and successor state axioms can easily be implemented in Prolog, just like GOLOG which we now turn to.

GOLOG [Levesque *et al.*, 1997] is a logic programming language based on the situation calculus. GOLOG offers control structures familiar from imperative programming languages like conditionals, loops, procedures, and others:

[3] 'Re-spawning' means the reappearance of a player or an item such that it becomes active again.

[4] Second-order logic is needed to define while-loops of programs, among other things.

[5] All free variables are assumed to be universally quantified.

**primitive actions:** $\alpha$ denotes a primitive action, which is equivalent to an action of the situation calculus.

**sequence:** $[e_1, e_2, \ldots, e_n]$ is a sequence of legal GOLOG programs $e_i$.

**test action:** $?(\phi)$ tests if the logical condition $\phi$ holds.

**nondeterministic choice of actions:** $e_1|e_2$ executes either program $e_1$ or program $e_2$.

**nondeterministic choice of arguments:** $pi(v, e)$ chooses a term $t$, substitutes it for all occurrences of $v$ in $e$, and then executes $e$;

**conditionals:** $if(\phi, e_1, e_2)$ executes program $e_1$ if $\phi$ is true, otherwise $e_2$;

**nondeterministic repetition:** $star(e)$ repeats program $e$ an arbitrary number of times;

**while loops:** $while(\phi, e)$ repeats program $e$ as long as condition $\phi$ holds:

**procedures:** $proc(p, e)$ defines a procedure with name $p$ and body $e$. The procedure may have parameters and recursive calls are possible.

We remark that the semantics of these constructs is fully defined within the situation calculus (see [Levesque *et al.*, 1997] for details). Given a GOLOG program, the idea is, roughly, to find a sequence of primitive actions which corresponds to a successful run of the program. These actions are then forwarded to the execution module of the agent like moving to a particular location or picking up an object.

READYLOG [Ferrein *et al.*, 2004] extends the original GOLOG in many ways. It integrates features like probabilistic actions [Grosskreutz, 2000], continuous change [Grosskreutz and Lakemeyer, 2000], on-line execution [De Giacomo and Levesque, 1998], decision-theoretic planning [Boutilier *et al.*, 2000], and concurrency [De Giacomo *et al.*, 2000; Grosskreutz, 2002]. Its primary use so far has been as the control language of soccer playing robots in the ROBOCUP middle-size league.

For the purpose of controlling UNREAL game bots, perhaps the most interesting feature of READYLOG is its decision-theoretic component. It makes use of nondeterministic as well as stochastic actions, which are used to model choices by the agent and uncertainty about the outcome of an action. To make a reasoned decision about which actions to choose, these are associated with utilities and the decision-theoretic planner computes the optimal choices (also called policies) by maximizing the accumulated expected utility wrt a (finite) horizon of future actions. This is very similar to computing optimal policies in Markov decision processes (MDPs) [Puterman, 1994].[6]

## 4 Modelling UNREAL in READYLOG

The UNREAL bots are described by a variety of fluents which have to be considered while playing the game. All of the fluents have a time stamp associated such that the bot is able to know how old and how precise his state information are.

**Identifier fluents:** In the set of identifier fluents the bots name, the currently executed skill, together with unique ids describing the bot and the skill can be found, among others.

**Location fluents:** The location fluents represent the bots location in a level, its current orientation, and its velocity.

**Bot Parameter fluents:** Health, armor, adrenaline, the currently available inventory in which the items are stored, and the explicit amount of each inventory slot is saved in this set of fluents. In the inventory additional game objective items can be found such as a flag in CTF.

**Bot Visibility fluents:** Here information about the objects in the view range of the agent are found. These information are distinguished in a teammate and an opponent set. They contain the bots identifier and its current location. In games without team play the set of friends stays always empty during gameplay.

**Item Visibility fluents:** Here the information about the currently visible and non visible items can be found. If an item is not visible at its expected position a competitor took it away and it reappears after a specific time. The definite re-spawn time of the item is unknown in general. The explicit re-spawn time is only available, if the bot itself took the item.

Bots in UNREAL TOURNAMENT 2004 are able to use the skills *stop, celebrate, moveto, roam, attack, charge, moveattack, retreat*, and *hunt*. All actions from UNREAL are modelled in the READYLOG framework as stochastic actions and successor state axioms are defined for all the fluents. Details are left out for space reasons.

Our framework is very flexible and allows for modelling different tasks in various ways, combining decision-theoretic planning with explicit programming. We begin by showing two extreme ways to specify one task of the bot, collecting health items. One relies entirely on planning, where the agent has to figure out everything by itself, and the other on programming without any freedom for the agent to choose.

The example we use for describing the different approaches, their benefits, and their disadvantages is the collection of health items in an UNREAL level. Because collecting any one of them does not have a great effect the agent should try to collect as many as possible in an optimal fashion. Optimal means that the bot takes the optimal sequence which results in minimal time and maximal effect. Several constraints like the availability have to be taken into account.

The first and perhaps most intuitive example in specifying the collection of health packs is the small excerpt from a READYLOG program shown in Program 4.1. Using decision-theoretic planning alone, the agent is able to choose in which order to move to the items based upon the reward function. The search space is reduced by only taking those navigation nodes into account which contain a health item.

The first action of the excerpt is a test action which binds the list of all health node in the current level to the free variable $HealthNodeList$. The $solve$ statement initiates the plan-

[6] What makes READYLOG different from ordinary MDPs is that the state space can be represented in a compact (logical) form and the control structure of a program allow a user to drastically reduce the search space.

**Program 4.1** READYLOG program to collect health powerups by setting up an MDP to solve. We scale down the search space by regarding the health nodes only. The reward function rewards low time usage and higher bot health.

```
  ...
  ?(getNavNodeHealthList( HealthNodeList ) ),
  solve( while( true,
                pickBest( healthnode, HealthNodeList,
                        moveto( epf_BotID, healthnode ) ) ),
         Horizon, f_HealthReward ),
  ...

function( f_HealthReward, Reward,
          and( [   CurrentTime = start,
                   TmpReward = epf_BotHealth - CurrentTime,
                   Reward = max( [TmpReward, 0] ) ] )
        ). % of simple_reward
```

ning process. Up to the horizon $Horizon$ the loop is optimized using the reward function $f\_HealthReward$ which simply awards the health status of the bot discounted over the planning time. Note that we assume a continuously changing world during plan generation. The $pickBest$ statement projects the best sequence of $moveto$ actions for each possible ordering of health nodes. This results in the optimal action sequence given the bot's current location as health nodes which are far away are honored a lower reward.

Note that in this first basic example all calculations are up to the agent. Information about availability of items, the distance or the time the agent has to invest to get to the item become available to the agent as effects of the *moveto* action. While easy to formulate, the problem of Program 4.1 is its execution time. With increasing horizon the computation time increases exponentially in the size of the horizon. All combinations of visiting the nodes are generated and all stochastic outcomes are evaluated. For example, in a setting with $Horizon = 3$ and $\#HealthNodes = 7$ the calculation of the optimal path from a specific position takes about $50$ seconds,[7] which makes this program infeasible at present.

The next idea in modelling the health collection is to further restrict the search by using only a subset of all available health nodes. The example shown previously took all health navigation nodes of the whole map into account, whereas a restriction of those nodes is reasonable. Items which are far away are not of interest to the agent. Because of this restriction the real-time demands are fulfilled in a better way but they are still not acceptable for the UNREAL domain. In the same setting as above ($Horizon = 3$ and $\#HealthNodes = 7$ from which only $Horizon+1 = 4$ health navigation nodes are chosen) the calculation of the optimal path lasts about $8$ seconds.

A much more efficient way to implement this action sequencing for arbitrary types is to program the search explicitly and not to use the underlying optimization framework. For example, filtering the available nodes and ordering them afterwards in an optimal way by hand is a much better way to perform on-line playing. The example described above is depicted in Program 4.2.

[7] The experiments were carried out on a Pentium 4 PC with 1.7GHz and 1GB main memory.

**Program 4.2** READYLOG program to collect health items which is able to be applied on-line. The method getNextVisNavNodes returns a list of navigation nodes with length Horizon which are of type Type ordered with increasing distance from location Loc and a minimal confidence of availability of either $0.9$ or $0.5$. The ordering is done by the underlying Prolog predicate sort. If one item matches the mentioned requirements, the agent travels there, and recursively calls the collect method again until Horizon is reached.

```
proc( collect( Type, Horizon ),
      if( neg( Horizon = 0 ),
          [
             ?( and( [  Loc = epf_BotLocation,
                        getNextVisNavNodes( Loc, Horizon, Type,
                                            0.9, TmpVisList ),
                        lif( TmpVisList = [],
                             getNextVisNavNodes( Loc, Horizon, Type,
                                                 0.5, VisList ),
                             VisList = TmpVisList ),
                        lif( neg( VisList = [] ),
                             VisList = [HeadElement|_TailElements],
                             HeadElement = nothing ),
                        NewHorizon = Horizon - 1
                      ] ) ),
             if( neg( VisList = [] ),
                 [  moveto( epf_BotID, HeadElement ),
                    collect( Type, NewHorizon )
                 ] )
          ] )
      ). % of collect( Type, Horizon )
```

This example of how modelling health collection can be done is far from optimal from a decision-theoretic point of view. There are no backup actions available if something goes wrong and no projection of outcomes is applied during execution. On the other hand, the execution of Program 4.2 is computationally inexpensive. Arbitrary horizons for collecting an item can be given to the program without an exponential blow-up.

Given the pros and cons of the two examples above, it seems worthwhile to look for a middle ground. The idea is to allow for some planning besides programmed actions and to further abstract the domain so that the search space becomes more manageable. Instead of modelling each detail for every action simpler models are introduced which do not need that much computational effort when planning.

To illustrate this we use an excerpt from our actual implementation of the *deathmatch agent* (Program 4.3). Here an agent was programmed which chooses at each action choice point between the outcomes of a finite set of actions. It has the choice between collecting a weapon, retreating to a health item, and so on based on a given reward function. The main part of the agent is the non-deterministic choice which represents the action the agent performs next. It has the choice between roaming and collecting items, attacking an opponent, or collecting several specific items. The decision which action to take next is performed based on the reward of the resulting state. Note also that the non-deterministic choices are restricted by suitable conditions attached to each choice. This way many choices can be ruled out right away, which helps prune the search space considerably.

## 5 Experimental Results

In our implementation we connected READYLOG and UNREAL via a TCP connection for each game bot. With this

**Program 4.3** Part of READYLOG program implementing an agent which is able to play Deathmatch games in UNREAL. The agent has several choices available and projects to choose the best action to execute. The results of this agent are presented in table 1.

```
proc( agent_dm( Horizon ),
      [ while( true,
               [ solve( [ nondet([if( f_SawOpponent = false,
                                      roam( epf_BotID ) ),
                                  if( f_SawOpponent = true,
                                      moveattack( epf_BotID,
                                            f_GetNextOppBot ) ),
                                  .....
                                  if( f_ItemTypeAvailable( health ),
                                      collect( health, 1 ) ),
                                  if(and([f_BotHasGoodWeapon = false,
                                     f_ItemTypeAvailable(weapon) = true
                                            ] ),
                                      collect( weapon, 1 ) )
                                  ] )
                        ], Horizon, f_DMReward )
               ] )
      ] ). % of agent_dm( Horizon )

function( f_DMReward, Reward,
          and([.....
               lif( epf_BotHealth < 150, RewardHealth1 = -1 ),
               lif( epf_BotHealth < 100, RewardHealth2 = -5 ),
               .....
               lif( epf_BotArmor > 135, RewardArmor4 = 20 ),
               .....
               RewardScore = -200*(CurrentMaxScore-MyScore),
               .....
               Reward = RewardHealth1 + RewardHealth2 + . + RewardScore
              ] ) ). % of f_DMReward
```

Table 1: UNREAL deathmatch results generated in our framework. The setting was as follows: GoalScore = 9, LevelTime = 6 min, SkillLevel = skilled. We present the median result of five experiments for each entry here.

| Level Name | #Player | RB only | RB vs. UB |
|---|---|---|---|
| Training Day | 2 | 9:6 | 8 : 9 |
| Albatross | 2 | 9:5 | 8 : 9 |
| Albatross | 4 | 9:8:5:3 | 8:1 : 9:5 |
| Crash | 2 | 8:7 | 7 : 8 |
| Crash | 4 | 9:7:5:3 | 8:5 : 9:6 |

Table 2: UNREAL Capture the Flag results generated in our framework. The setting was as follows: GoalScore = 5, LevelTime = 6 min, SkillLevel = skilled. We present here the median result of five experiments for each entry.

| Level Name | #Players | RB only | RB vs. UB | Mixed |
|---|---|---|---|---|
| Joust | 2 | 5:3 | 5:3 | - |
| Maul | 4 | 1:0 | 0:1 | 2:1 |
| Face Classic | 6 | 2:1 | 0:1 | 2:1 |

connection the programs transmit all information about the world asynchronously to provide the game-bot with the latest world information and receive the action which the bot shall perform next until a new action is received. With this setup and after implementing an agent to play different styles of play, we conducted several experiments.

The most important thing to be mentioned before attending to the explicit results is that the game is highly influenced by luck. Letting two original UNREAL bots compete in the game can result in a balanced game which is interesting to observe or in an unbalanced game where one bot is much more lucky than the others and wins unchallenged with healthy margin. Because of that we did run every test several times to substantiate our results.

Table 1 shows the results of the deathmatch agent which we described in the last section. In this and the next table the first column contains the name of the level we used for testing. The second column shows the total number of players competing in this level. In the following columns the results of different settings of the game are represented. The token UB stands for the original UNREAL bot. RB represents the READYLOG bot. "RB only" means that only READYLOG bots competed. "RB vs. UB" means that the READYLOG bots compete against the UNREAL bots. The entries in line 3 and 5 mean the total ranking of the four competing bots, i.e. the winning bot got a score of 9, the second a score of 8, the third a score of 5, and the fourth a score of 3. In the column "RB vs. UB" the first two entries show the results of the READYLOG bots against the two UNREAL bots.

Next we consider the capture-the-flag agent, which was implemented based on the team deathmatch agent. Here we focused on the implementation of a multi-agent strategy to be able to play Capture the Flag on an acceptable level.

We introduced two roles to implement a strategy for this type of game which we called *attacker* and *defender*. The attacker's task is to try to catch the opponents flag and to hinder the opponents from building up their game. The defender's task is to stay in the near vicinity of the own flag and to guard it. If the own flag is stolen its job is to retrieve it as fast as possible.

Each role was implemented based on a simple set of rules based on the state of each team's flag: the two flags can each be in three states, *at home, carried*, or *dropped*. For each of the resulting nine combinations of the two flags we implemented a small program for each role. E.g. if the own flag is in the state dropped the defender's task is to recapture it by touching the flag.

For several states we introduced nondeterministic choices for the agent. It is able to choose between collecting several items or trying to do its role-related tasks.

The results can be interpreted as follows: In the one-on-one level *Joust* the READYLOG bot is surprisingly strong in gameplay. We confirmed those results in other one-on-one levels. We think this is due to the goal directed behavior of our attacker. The agent does not care much about items and mainly fulfills its job to capture the flag and recapture the own flag. The column titled "Mixed" in Table 2 shows the result where READYLOG and UNREAL bots together made up a team.

There exist several problems which we describe here but could not attend to because of time constraints. First of all the bots always choose the same paths in the map. This is not a big problem in games against UNREAL bots but humans observe and learn this behavior fast and are able to use this to their advantage.

## 6 Conclusions

We implemented a framework which enables us to control UNREAL game bots using the logic-based action language READYLOG, which enables the user to mix programmed actions decision-theoretic planning for intelligent game play. Different game types were implemented and experiments were carried out, where READYLOG bot is able to compete with the original UNREAL bot.

One can argue that the difficulties of our approach lie in the modeling of the domain instead of modeling the behavior of the bots. On the one hand, this is true because designing a good reward functions is a subtle issue. On the other hand, it is easier to model the effects of actions and letting the agent decide to choose the appropriate actions. Finding a middle ground between full decision-theoretic planning and programming leaves some choices by the bot resulting in more flexible behavior.

While our current bots can certainly be improved in many ways, perhaps the most important message of this paper is that logic-based action languages, which for the most part have only been considered in the theoretical KR community, can actually be used in challenging environments like interactive computer games.

## References


[Berger, 2002] L. Berger. Scripting: Overview and Code Generation. In *AI Game Programming Wisdom*, volume 1, pages 505–510. Charles River Media, 2002.

[Boutilier *et al.*, 2000] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-Theoretic, High-Level Agent Programming in the Situation Calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 355–362, Menlo Park, CA, USA, July 2000. AAAI Press.

[Buro, 2003] M. Buro. Real Time Strategy Games: A new AI Research Challenge. In *Proceedings of the International Joint Confercence on AI*, Acapulco, Mexico, 2003. AAAI Press.

[De Giacomo and Levesque, 1998] G. De Giacomo and H. J. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. Technical report, Department of Computer Science, University of Toronto, Toronto, Canada, 1998.

[De Giacomo *et al.*, 2000] G. De Giacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, 2000.

[Electronic Arts Inc., 2005] Electronic Arts Inc. http://www.ea.com/, last visited in January 2005.

[Epic Games Inc., 2005] Epic Games Inc. http://www.unrealtournament.com/, last visited in February 2005.

[Ferrein *et al.*, 2004] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line decision-theoretic golog for unpredictable domains. In *Proc. of 27th German Conference on AI*, 2004.

[Fu and Houlette, 2004] D. Fu and R. Houlette. The Ultimate Guide to FSMs in Games. In *AI Game Programming Wisdom*, volume 2, pages 3–14. Charles River Media, 2004.

[Funge, 1998] J. Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, Toronto, Canada, 1998.

[Grosskreutz and Lakemeyer, 2000] H. Grosskreutz and G. Lakemeyer. cc-Golog: Towards More Realistic Logic-Based Robot Controllers. In *AAAI-00*, 2000.

[Grosskreutz, 2000] H. Grosskreutz. Probabilistic Projection and Belief Update in the pGOLOG Framework. In *CogRob-00 at ECAI-00*, 2000.

[Grosskreutz, 2002] H. Grosskreutz. *Towards More Realistic Logic-Based Robot Controllers in the GOLOG Framework*. PhD thesis, RWTH Aachen University, Knowledge-based Systems Group, Aachen, Germany, 2002.

[Kaminka *et al.*, 2002] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholder, and S. Tejada. Game Bots: A Flexible Test Bed for Multiagent Research. *Communications of the ACM*, 45(2):43–45, 2002.

[Levesque *et al.*, 1997] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31:59–84, April-June 1997.

[Lewis, 1999] R. L. Lewis. Cognitive modeling, symbolic. In *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, Cambridge, Massachusetts, USA, 1999.

[Magerko *et al.*, 2004] B. Magerko, J. E. Laird, M. Assanie, A. Kerfoot, and D. Stokes. AI Characters and Directors for Interactive Computer Games. In *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Confercence, San Jose, CA*. AAAI Press, 2004.

[McCarthy, 1963] J. McCarthy. Situations, Actions and Causal Laws. Technical report, Stanford University, 1963.

[Munoz-Avila and Fisher, 2004] H. Munoz-Avila and T. Fisher. Strategic Planning for Unreal Tournament Bots. In *AAAI Workshop on Challenges in Game AI*, San Jose, CA, USA, July 2004.

[Orkin, 2004] J. Orkin. Symbolic Representation of Game World State: Toward Real-Time Planning in Games. In *AAAI Workshop on Challenges in Game AI*, San Jose, CA, USA, July 2004.

[Puterman, 1994] M. Puterman. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley, New York, USA, 1994.

[Reiter, 2001] R. Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[Valve Corporation, 2005] Valve Corporation. http://www.valvesoftware.com/, last visited in January 2005.

# Knowledge Organization and Structural Credit Assignment

**Joshua Jones and Ashok Goel**
College of Computing
Georgia Institute of Technology
Atlanta, USA 30332
{jkj, goel}@cc.gatech.edu

## Abstract

Decomposition of learning problems is important in order to make learning in large state spaces tractable. One approach to learning problem decomposition is to represent the knowledge that will be learned as a collection of smaller, more individually manageable pieces. However, such an approach requires the design of more complex knowledge structures over which structural credit assignment must be performed during learning. The specific knowledge organization scheme chosen has a major impact on the characteristics of the structural credit assignment problem that arises. In this paper, we present an organizational scheme called Externally Verifiable Decomposition designed to facilitate credit assignment over composite knowledge representations. We also describe an experiment in an interactive strategy game that shows that a learner making use of EVD is able to improve performance on the studied task more rapidly than by using pure reinforcement learning.

## 1 Introduction

The need for decomposition in learning problems has been widely recognized. One approach to making learning in large state spaces tractable is to design a knowledge representation composed of small pieces, each of which concerns a more compact state space than the overall problem. Techniques that would be intractable for the problem as a whole can then be applied successfully to each of the learning subproblems induced by the set of components.

Such composite knowledge representations, however, require the design of top-level structures that combine the knowledge that will be stored at individual components into a usable whole that encodes knowledge about the complete problem. These structures raise new issues for credit assignment. Specifically, there is a need to perform structural credit assignment over the top-level structure during learning.

The temporal credit assignment problem takes as input the outcome of a sequence of actions by an agent and gives as output a distribution over the actions in the sequence, where the output distribution specifies the relative responsibility of the actions for the outcome. In contrast, the structural credit assignment problem takes as input the outcome of a single action by an agent and gives as output a distribution over the components of the agent, where the output distribution specifies the relative responsibility of the components for the outcome. In this work, we are interested (only) in the structural credit assignment problem as it pertains to a learning agent's knowledge. In particular, we are interested in specifying and organizing knowledge components so as to enable accurate and efficient structural credit assignment over the resulting structure.

The question then becomes what might be the design principles for organizing knowledge and what additional knowledge might be encoded with each component to facilitate structural credit assignment? In this paper, we present an organizational and encoding scheme that we call Externally Verifiable Decomposition (or EVD). We also describe experimental results in an interactive strategy game, comparing reinforcement learning with EVD.

## 2 Externally Verifiable Decomposition

We begin by informally describing the EVD scheme with an illustrative example from an interactive strategy game called Freeciv (http://www.freeciv.org). We provide a formal description of EVD later in the paper.

### 2.1 Freeciv

FreeCiv is an open-source variant of a class of Civilization games with similar properties. The aim in these games is to build an empire in a competitive environment. The major tasks in this endeavor are exploration of the randomly initialized game environment, resource allocation and development, and warfare that may at times be either offensive or defensive in nature. Winning the game is achieved most directly by destroying the civilizations of all opponents. We have chosen FreeCiv as a domain for research because the game provides challenging complexity in several ways. One source of complexity is the game's goal structure, where clean decomposition into isolated subgoals is difficult or impossible. The game is also partially observable. The map is initially largely hidden, and even after exploration does not reflect state changes except in portions directly observed by a player's units. A consequence is that the actions of opponents may not be fully accessible. Also, the game provides a very

large state space, making it intractable to learn to play well without decomposition.

It is not necessary to understand the game completely for the purpose of understanding this study, but some specifics are in order. The game is played on a virtual map that is divided into a grid. Each square in this grid can be characterized by the type of terrain, presence of any special resources, and proximity to a source of water such as a river. In addition, each square in the grid may contain some improvement constructed by a player. One of the fundamental actions taken while playing FreeCiv is the construction of cities on this game map, an action that requires resources. In return for this expenditure, each city produces resources on subsequent turns that can then be used by the player for other purposes, including but not limited to the production of more cities. The quantity of resources produced by a city on each turn is based on several factors, including the terrain and special resources surrounding the city's location on the map, the construction of various improvements in the squares surrounding the city, and the skill with which the city's operations are managed. As city placement decisions are pivotal to success in the game, an intelligent player must make reasoned choices about where to construct cities.

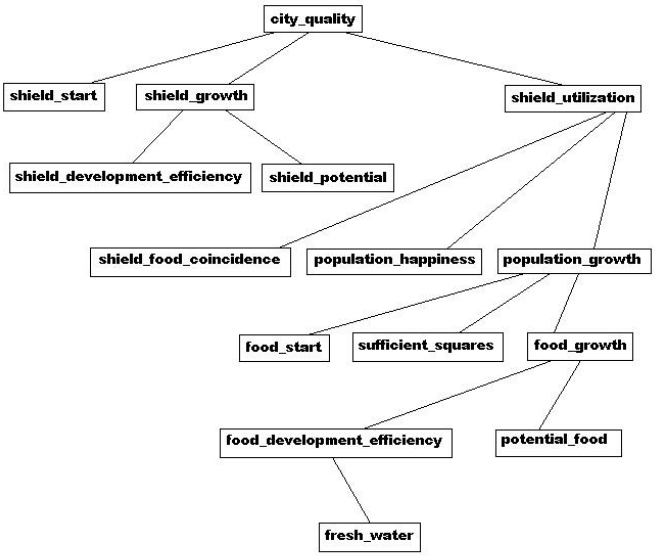


Figure 1: City Estimate Decomposition

## 2.2 Learner Design

We have designed an agent that plays FreeCiv. In this study, we are focused on evaluating EVD for a relatively small but still challenging part of the game. To that end, we have applied both EVD and Q-learning [Watkins, 1989] to a part of the module responsible for making decisions about city placement, specifically a part responsible for estimating the expected resource production over time if a city were built at a particular map location. This estimate is a state abstraction that can be used by a higher level reasoner to make decisions about where to place cities. In the experiment described in this paper, we are not concerned with the success of a higher level reasoner, but only in acquiring the knowledge needed to produce state abstractions that accurately project resource production.

### EVD Learner Design

Informally, the EVD scheme for organizing an agent's decision-making process has the following characteristics (we provide a formal description of EVD later). Firstly, from a top-level perspective, knowledge is organized in a state-abstraction hierarchy, progressively aggregating and abstracting from inputs. Figure 1 illustrates the hierarchy used to apply EVD in our chosen problem setting. This decomposition is a simplification of actual game dynamics, but is sufficient for the purposes of this study. The output of this structure is one of nine values representing the expected resource production of the terrain surrounding the map tile represented by the inputs. Specifically, the value represents the (short and long term) estimated resource production on each turn relative to a baseline function. The baseline is computed as a function of the number of turns for which the city in question has been active. This baseline accounts for the fact that absolute resource production is expected to increase over time. Note that no single estimate value may be correct for a given set of inputs, so the goal of learning is to minimize rather than eliminate error in these estimates.

The leaf nodes in the hierarchy discretize raw features of the world. For example, *food_start* takes as input a raw perception available from the environment (a set of numbers) that represents the food production value of game tiles in the inspected area. This leaf node produces as output an integral value from 1 to 5, representing the food resources initially available to the prospective city in a form usable by the parent node. These discretization functions in leaf nodes are hard-coded in this study.

Nodes at any level higher than leaf nodes in the hierarchy view the outputs of child nodes as features and aggregate and abstract them into states at their level. These abstractions made at intermediate nodes are learned; that is, each node contains a learner (the specific type of which is not specified by EVD) capable of mapping the vector of inputs to an output value. In this work, we use a simple table-lookup procedure within each node to perform the mapping, where learning is performed via a simple scheme that gradually changes table entries as examples are presented. For example, the *population_growth* component contains a three dimensional table indexed by the output values of *food_start*, *sufficient_squares* and *food_growth*, and produces an integral output value from 1 to 5, abstractly representing the expected population growth for the prospective city.

The most important feature of EVD is the association of predictive knowledge with each non-leaf component. Specifically, this knowledge establishes failure (success) conditions for each output value available to the component with which it is associated. In general these conditions could indicate "hard" success or failure, or values along some gradient between success and failure, interpretable as severity of failure and/or confidence in local failure. Here, we deal only with

digital success or failure. The predictive knowledge associated with a component can be thought of as encoding the semantics of the state abstraction knowledge acquired by the learner within the component.

This predictive knowledge is encoded in structural credit assignment functions, one of which is associated with each agent component. These functions are used to reflect upon the correctness of each component's action as information becomes available through perception subsequent to value production. The set of values produced by each component's function form an (unnormalized) distribution over the agent structure, as required for structural credit assignment. As an example, the *population_growth* node has an associated function that takes as input the actual population of a city, the number of turns that have passed since the city was built, and the value that was produced by the *population_growth* node's table during previous inference, and returns a boolean value indicating whether the value produced is correct in light of the first two parameters. This function was hand coded during the design phase, representing the intended semantics of the state abstraction to be produced by the node.

We term the property of a decomposition that is constructed to allow the encoding of this type of predictive knowledge "external verifiability" because each of the components is designed to have semantics that can be verified based on available percepts. Note that the technique is useful only when values produced by components can be verified in retrospect, when percepts become available after action selection – if they could be verified with certainty at the time values are produced, there would be no need for learning within components. Instead, the knowledge used for verification could simply be used to produce values directly.

This principle of external verifiability is used when designing a problem decomposition, guiding the choices that are made in terms of problem framing and the definition of agent components. The prescription of EVD is that these choices must be made so that the necessary predictive knowledge can be defined for each component. Beyond the facilitation of credit assignment during learning, this type of decomposition also has the advantage that agent components have meaningful, known semantics, allowing the results of learning to be easily interpreted.

Inference over an EVD structure is straightforward. In order to produce an output, each component recursively queries its children to obtain its inputs, and then uses its learner to map these inputs into an output value. This recursion is terminated at the leaves, at each of which a feature of raw input state is discretized and returned.

**Q-Learning Agent Design**

A separate Q-learning agent was also implemented for the purposes of comparison. The outputs of the EVD learner's input mappers are composed into a feature vector that is used as the input state description for the Q-learner. The set of values that can be produced as outputs by the EVD learner form the set of actions available to the Q-learner. This setup is intended to provide an I/O environment for the Q-learner that matches that of the EVD learner closely. Each time the action selected by the Q-learner corresponds to an incorrect resource production estimate for the current turn, a reward of -1 is given. If the action corresponds to a correct value for the current turn, a reward of 0 is given. Exploration is handled by initializing the value of all state-action pairs to 0, the highest value possible under this reward scheme.

## 3 EVD Credit Assignment Procedure

While predictive knowledge forms the basis for credit assignment over structures resulting from the application of EVD, it is not sufficient in and of itself. Predictive knowledge establishes failure conditions for each component. However, this knowledge does not capture the dependencies among agent components, and these dependencies are also significant in determining the failure status at each node. For this reason, the failure conditions used for credit assignment must incorporate both the local predictive knowledge at each component and some information about top-level knowledge structure.

This work focuses on decompositions with a hierarchical structure, that progressively aggregate and abstract from raw state features. For these state abstraction hierarchies, the observation necessary is that failure at a given component may be due either to erroneous knowledge (mapping from inputs to outputs) stored locally, or may be due to errors at the inputs arising from faults nearer the leaves. This structural characteristic, along with the functions enabling retrospective local verification when feedback becomes available, forms the basis for the credit assignment process associated with this decomposition technique.

Because the verification functions associated with each agent component may in principle be arbitrarily complex, it is advantageous to avoid evaluations whenever possible. In order to limit the number of evaluations required during each learning episode, we view incorrect behavior at a node as the production of a value that is both registered as a failure by the local verification function and that (recursively) results in incorrect behavior at the parent. Only the first condition is required for behavior at the root of the hierarchy to be viewed as incorrect. Viewing error in this way means that during some learning episodes, components that produce actions inconsistent with the local verification function may not be given weight by the credit assignment process, if the error is not propagated to the root of the hierarchy. The negative repercussion of this choice is that some opportunities for learning may not be fully exploited. However, if an action taken by some agent component is truly to be seen as incorrect, it is reasonable to require that there be some set of circumstances under which the erroneous action contributes to an overall failure. If such a situation exists, the credit assignment process will eventually recommend a modification at the component. If no such situation exists, it is safe to allow the component to continue with the "erroneous" behavior. The major benefit of this view is that when a component is found to have chosen an action that agrees with the local verification function, none of the components in the subtree rooted at the evaluated component need to be examined.

Given this view of error, the set of local verification functions associated with each component, and knowledge of the hierarchical structure of the decomposition, the structural

```
bool EVD_assign_credit(EVD d, percepts P)

  bool problem ← false

  if d→F(d→a, P) == 1
    return false
  end

  forall children c of d
    if EVD_assign_credit(c, P) == true
      problem ← true
    end
  end

  if !problem
    mark e
  end
  return true
end
```

Figure 2: Pseudo-code for EVD structural credit assignment. Each node has an associated structural credit assignment function as described above, denoted 'F' here. Each component is also expected to store its last action, 'a'.

credit assignment process is as shown in Figure 2. The function is called when new percepts become available from the environment (here, on each new turn), and results in marking the nodes identified as responsible for failure, if any. When invoked, the procedure evaluates the verification function at the root of the hierarchy based on the relevant value previously selected at that component. If the verification is successful, no further action is taken. If an error is indicated, each child is visited, where the procedure is recursively repeated. If and only if no error can be found at any child, the current node is marked as being in error. The base case for this recursive procedure is achieved by defining leaves in the hierarchy as correct; that is, inputs representing raw state are never considered to be a source of error, but are provided with "dummy" verification functions that yield 1 for all inputs. This procedure treats error as a digital property of agent components, not making distinctions in degree of failure. Notice that this procedure also makes the commitment that error is due either to local knowledge *or* to erroneous inputs. Also note that this credit assignment procedure is purely structural. That is, no temporal credit assignment is handled by this algorithm. For this reason, the percepts 'P' can be directly attributed to the last action 'a' taken by each component. This is why cached last action values can be used in the algorithm above. In order to address both structural and temporal credit assignment, the method described here could be used to distribute credit structurally after another technique has distributed credit temporally.

Based on the result of structural credit assignment, learning is performed at each node that has been marked as erroneous, by whatever procedure is applicable to the type(s) of learners used within the components. Note that the decomposition method and accompanying method for structural credit assignment make no prescription whatsoever in terms of the knowledge format or learning procedure that is used within each node. In this work, a simple table-update routine was used within each component. However, in principle any other type of learning technique desired could exist within each component.

# 4 Experiments

In order to provide evidence that the decomposition technique and associated structural credit assignment method outlined above provide advantages over learning in a flat problem space, we have applied the technique to a problem within a strategy game playing agent, and compared the results with an RL implementation, specifically Q-learning, as discussed previously.

## 4.1 Procedure

Because we train and evaluate the learners in an on-line, incremental fashion, we cannot apply the standard training set/test set approach to evaluation. Rather, we evaluate the learners' performance improvement during training by segmenting the sequence of games played into multi-game blocks, and comparing overall error rate between blocks. In this way, we are able to compare error rate around the beginning of a training sequence with the error rate around the end of that sequence.

Errors are counted on each turn of each game by producing a value (equivalently, selecting an action), finishing the turn, perceiving the outcome of the turn, and then determining whether the value produced correctly reflects the resource production experienced on that turn. If the value is incorrect, an error is counted. Note that this error counting procedure contrasts with another possibility; producing a value only at the beginning of each game, and counting error on each turn of the game based on this value, while continuing to learn on each turn. While this alternative more closely matches the intended *use* of the learned knowledge, we chose to instead allow a value to be produced on each turn in order to reflect the evolving state of knowledge as closely as possible in the error count. A negative consequence of this choice is that some overfitting within games may be reflected in the error count. However, a decrease in error rate between the first and last block in a sequence can be seen as evidence of true learning (vs. overfitting), since any advantage due to overfitting should be as pronounced in the first group of games as in the last. Also note that error counting was consistent for both the EVD-based learner and the Q-learner.

In each trial, a sequence of games is run, and learning and evaluation occurs on-line as described above. The EVD-based learner is trained on sequences of 175 games, while the Q-learner is allowed to train on sequences of 525 games. We trained the Q-learner on sequences three times longer than those provided to the EVD learner to determine whether the Q-learner's performance would approach that of the EVD learner over a longer training sequence. As described above, we segment these sequences of games into multi-game blocks for the purpose of evaluation; the block sized used is 7 games. Each game played used a (potentially) different randomly generated map, with no opponents. The agent always builds a city on the first occupied square, after making an estimate of the square's quality. Building in the first randomly generated occupied square ensures that the learners will have opportunities to acquire knowledge in a variety of states. Though this

| | EVD agent | Q-learning agent | |
|---|---|---|---|
| | $7^{th}$ block | $7^{th}$ block | $21^{st}$ block |
| Without city improvements | 24% | (4%) | 1% |
| With city improvements | 29% | 7% | 10% |

Table 1: Average percent decrease (or increase, shown in parentheses) in error for decomposition-based learning implementation from block 1 to 7, and for the Q-learning agent from block 1 to blocks 7 and 21.

setup is simpler than a full-fledged game, it was sufficient to illustrate differences between the learners. In order to compensate for variation due to randomness in starting position and game evolution, results are averaged over multiple independent trial sequences. Each result for the EVD learner is an average of 60 independent trials. Each result for the Q-learner is an average over 25 independent trials; each trial is time consuming, as each trial for the Q-learner is three times as long as for the EVD-learner, and it did not seem likely that further trials with the Q-learner would offer significantly more information.

To compare the speed with which learning occurs in the two agents, we ran two separate sets of trials. The first set of trials was run in an environment where no city improvements were constructed in the area surrounding the city. The second set of trials did allow for the construction of city improvements, but had an identical environment in all other ways. For each set of environmental conditions, we measure the quality of learning by comparing the average number of errors counted in the first block of the sequences to the number of errors counted in the last block. In the case of the Q-learner, we make two comparisons. The first compares error in the first block to the block containing the 175th game, illustrating decrease in error over the same sequence length provided to the EVD learner. We also compare error in the first block to error in the last block of the Q-learner's sequences, to determine whether the Q-learner's improvement will approach that of the EVD learner over sequences three times as long. We perform this evaluation separately for each of the two environmental setups.

### 4.2 Results

The results of the experiment described above are summarized in Table 1. The EVD based learner is able to produce a greater improvement in error rate in each case, as compared to the Q-learner, both after the same number of games and after the Q-learner has played three times as many games. For the two scenarios, the average improvement in error rate is 26.5%, compared to only 1.5% after the same number of training examples for Q-learning. The decrease in error across a typical sequence was not strictly monotonic, but did exhibit progressive decrease rather than wild fluctuation. Even after three times as many games had been played by the Q-learning agent, the decrease in error rate is significantly less than that achieved using EVD after only seven blocks. In one case, it appears that learning has not yielded an advantage in error rate in the Q-learning agent even after 525 games. Examining the complete set of results for intervening blocks does mitigate this impression to some extent, as an overall downward trend is observed, with some fluctuations. However, given that the fluctuations can be of greater magnitude than the decrease in error due to Q-learning, the learning that has been achieved after this number of games does not appear significant. Based on the significant difference in observed learning rate, these trials provide evidence that the decomposed structure and accompanying structural credit assignment capabilities of EVD do offer an advantage in terms of allowing learning to occur more quickly in a large state space.

## 5 Formal Description of EVD Structure

As has been discussed, this paper is focused on decompositions that progressively abstract away from raw state features through a hierarchy of components. Abstraction hierarchies can be viewed as handling a class of tasks, termed select-1-out-of-$n$, in a way that has been identified and formally described by Bylander, Johnson and Goel [Bylander *et al.*, 1991]. We base our formal description of EVD structure on this class of tasks. The select-1-out-of-$n$ task is defined as follows:

**Definition 5.1** *Let $C$ be a set of choices. Let $P$ be a set of parameters. Let $V$ be a set of values. Let an assignment of values in $V$ to the parameters $P$ be represented by a function $d : P \rightarrow V$. Then let $D$ be the set containing all possible parameter assignments $d$. The* select-1-out-of-$n$ *task is defined as a tuple, $< P, V, C, s >$, where $s$ is a function $s : D \rightarrow C$.*

In practice, each of the parameters in $P$ may have a distinct set of legal values. In this case, $V$ is the union of the sets of legal input values to each parameter $p \in P$, and $D$ is restricted to contain only functions $d$ that provide legal assignments of values to parameters.

Before formally defining EVD structure, we need to define an *input mapper*.

**Definition 5.2** *An* input mapper *is defined as a tuple $< p, V, C, \mathcal{T} >$, where $p$ is a single input parameter, $V$ is the set of values that can be taken by the parameter, and $C$ is the set of possible output values. $\mathcal{T}$ is a function $\mathcal{T} : V \rightarrow C$ that implements the translation of input values to the choice alphabet.*

Now we can formally define EVD structure.

**Definition 5.3** *An* EVD *is recursively defined as a tuple $< P, V, C, \mathcal{P}, \mathcal{S}, \mathcal{L}, F >$, where $P$ is the set of input parameters, $V$ is the set of values that can be taken by those parameters, and $C$ is the set of choices that form the output of the judgement. $\mathcal{P}$ is a tuple $< P_1, ..., P_r >$ such that $\{P_1, ..., P_r\}$ is a partition of the parameters $P$ of rank $r$. That is, $P_1, ..., P_r$ are non-empty disjoint sets whose union is $P$. $\mathcal{S}$ is a tuple $< s_1, ..., s_r >$, where $s_i$ is an* EVD *with parameters $P_i$, values $V$ and choices $C$ if $|P_i| > 1$, or an* input mapper *with parameter $p, p \in P_i$, values $V$ and choices $C$ if $|P_i| = 1$. $\mathcal{L}$ is an arbitrary learner with domain $C^r$ and range $C$. $F$ is a function $F : e \times C \rightarrow \{1, 0\}$, where $e$ is a representation of feedback perceived from the environment.*

Once again, $V$ represents the union of the values taken by all EVD parameters; value assignments and functions involving value assignments are restricted to handle legal assignments only. This treatment of $V$ is for notational convenience. Similarly, some subtrees may return only a subset of $C$, and $\mathcal{L}$ at the parent node need not handle outputs of a subtree that cannot legally be produced. The function $F$ encodes predictive knowledge about the knowledge encoded in the component, as described above.

Evaluation of an EVD is handled in two steps. First, determine the input to $\mathcal{L}$ by evaluating each $s_i \in \mathcal{S}$, and then produce as output the result of applying $\mathcal{L}$ to the generated input vector. Input mappers are evaluated by applying $\mathcal{T}$ directly to the value of the sole parameter $p$. Learning over the EVD structure as a whole proceeds by first using the credit assignment technique described previously, and then applying feedback to the learner within each EVD component as dictated by the outcome of credit assignment.

## 6 Discussion

The intended contribution of this work is in making explicit the connection between structural credit assignment and decomposition of learning problems via composite knowledge representation, and in describing an approach that drives the design of knowledge representations based on the needs of structural credit assignment. The connection between decomposition and structural credit assignment has been recognized by Dietterich in his work on hierarchical reinforcement learning, where he refers to the problem as hierarchical credit assignment [Dietterich, 1998]. However, the MAXQ method takes a different approach to decomposition that is not directly driven by the need to perform credit assignment over the resulting structure, and focuses on temporal rather than state abstractions.

Layered learning [Whiteson *et al.*, 2005] makes use of decomposition hierarchies to address large learning problems. In layered learning, each component's learner is trained in a tailored environment specific to the component. The EVD technique is more akin to what is called "coevolution" of components in work on layered learning, where all of the learners in the decomposition hierarchy are trained as a complete system in the actual target domain. Some notion of external verifiability is implicit in hierarchies built to be trained with coevolution, as the evaluation functions used for each component must be evaluable based on available percepts. Here, we are explicit about the need to design decompositions specifically around this property, we allow the use of arbitrary (possibly heterogeneous) learners within each component, and provide a procedure for credit assignment over the decomposition that can help to limit evaluations. An additional distinction is that EVDs focus on state abstraction. These decompositions aim to limit the number of inputs to each component, ensuring a learning problem of manageable dimensionality at each component. In contrast, layered learning focuses on temporal abstraction, where components responsible for selection of abstract actions are not necessarily shielded from the need to consider many raw state features.

Work on Predictive State Representations (PSRs) [Littman *et al.*, 2001] outlines rationale for using state representations that directly encode predictions about future events. In a general way, the notion that knowledge should have a predictive interpretation is central to this work as well. However, the specific problems of decomposition and structural credit assignment that motivate this work are not the focus of PSRs, and there are clearly significant differences between PSRs and the work described here.

This work is an extension of our previous efforts [Jones and Goel, 2004]. In addition to comparing the effectiveness of EVD with reinforcement learning over a flat representation, this paper extracts and begins to formalize the key design principles from our previous work in the hopes that these principles may be useful to researchers designing knowledge representations for learning in other domains.

## 7 Conclusions

This paper presents an approach to designing a composite knowledge representation for a learning agent that is directly driven by the needs to perform structural credit assignment over the resulting top-level structure. The experiment described here provides evidence that the approach and accompanying credit assignment technique are sufficient for learning, and that the decomposition increases the tractability of learning in a large state space. This means that if a problem framing and set of components can be defined according to the principle of external verifiability for a given problem, EVD-based knowledge structure and credit assignment may be used to accelerate learning. In principle, this type of decomposition should be compatible with a variety of learning techniques within each component, and even with a heterogeneous set of techniques. Such combinations have the potential to create learners for complex problems with varying characteristics.

## References


[Bylander *et al.*, 1991] T. Bylander, T.R. Johnson, and A. Goel. Structured matching: a task-specific technique for making decisions. *Knowledge Acquisition*, 3:1–20, 1991.

[Dietterich, 1998] T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proc. 15th International Conf. on Machine Learning*, pages 118–126. Morgan Kaufmann, San Francisco, CA, 1998.

[Jones and Goel, 2004] J. Jones and A. Goel. Hierarchical Judgement Composition: Revisiting the structural credit assignment problem. In *Proceedings of the AAAI Workshop on Challenges in Game AI, San Jose, CA, USA*, pages 67–71, 2004.

[Littman *et al.*, 2001] M. Littman, R. Sutton, and S. Singh. Predictive representations of state, 2001.

[Watkins, 1989] C. J. Watkins. *Learning from delayed rewards*. PhD thesis, Cambridge university, 1989.

[Whiteson *et al.*, 2005] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59(1):5–30, 2005.

# Requirements for resource management game AI

**Steven de Jong, Pieter Spronck, Nico Roos**
Department of Computer Science, Universiteit Maastricht, Netherlands
Email: {steven.dejong, p.spronck, roos}@cs.unimaas.nl

## Abstract

This paper examines the principles that define resource management games, popular and challenging constructive computer games such as SIMCITY and VIRTUAL U. From these principles, it is possible to derive requirements for intelligent programs designed to play such games, as a replacement of a human player.

A first step for research in the domain of intelligent programs playing resource management games is presented, in the form of a hybrid AI approach that combines abductive planning with evolutionary learning. The results obtained by this approach are promising.

## 1 Artificial intelligence in interactive computer games

One of the main goals of AI research always has been the development of artificial intelligence that is as versatile as human intelligence. For many years, the game of CHESS has been the *drosophila melanogaster* of artificial intelligence research [McCarthy, 1990], but in the last decades, it is argued that games such as CHESS are not able to address all abilities of intelligence sufficiently, because they are abstract and completely deterministic [Pfeiffer and Scheier, 1999; Laird and van Lent, 2001]. Computers are good at calculation and therefore inherently good at dealing with abstract and deterministic problems, but being good at calculation clearly is not the same as being able to make intelligent decisions [Pfeiffer and Scheier, 1999].

In recent years, the computer games industry has received increasing attention from the artificial intelligence community, because interactive computer games, while being a closed world, can require a computer to perform tasks that currently only humans are able to perform sufficiently well. Interactive computer games are therefore seen as an ideal testbed for alleged computer intelligence [Laird and van Lent, 2001]. Potential tasks for AI in computer games can vary from entertaining the human player to actually being able to play an entire game.

Since designers of interactive computer games aim at a commercial rather than a scientific goal, computer controlled players for such games are rarely, if ever, designed to explore the game autonomously to arrive at high-quality decision making capabilities (i.e., high-quality game AI). Therefore, if game AI is applied in practise, it is often static and only encompasses the designers' insight on what the game constitutes. We observe three problems here.

First of all, from a scientific perspective, we should aim at developing AI methods that do not need designers' insight in order to work well. A method that works well because it reproduces knowledge programmed ahead of time, may be able to play an intelligent game, but cannot be considered to be intelligent itself. Moreover, human designers may make mistakes, or fail to take into account relevant information, which makes the game AI of inferior quality.

Second, methods using knowledge programmed ahead of time will not be able to adapt when a game's parameters change, or when opponents are encountered that follow strategies not taken into account. In commercial computer games, the specifications are often changed to resolve balance issues. Adaptive AI methods offer an advantage here: if changes are needed, the methods can adapt easily and do not need to be rebuilt.

Third, even if our only goal is to develop computer-controlled players for commercial games, we must realize that most human players expect that a computer plays fairly (i.e., it does not use knowledge that human players do not have) [Scott, 2002]. For example, Garry Kasparov was furious with Deep Blue when the computer had beaten him in CHESS, because the grandmaster was convinced that it had been using information provided during the game by a team of CHESS players [Jayanti, 2003].

In this paper, we examine the principles that define resource management games, which are interactive computer games with a constructive nature. These principles lead to a list of requirements for intelligent programs designed to play such games, as a replacement of a human player. Exploratory research in the domain of intelligent programs playing resource management games

shows that hybrid AI approaches, combining abductive planning with evolutionary learning, are a possible way of dealing with these requirements.

In section 2 of this paper, we address the relevance of research in the domain of resource management games. In section 3, we discuss the principles underlying these games and derive the required capabilities of resource management game players (including computer-controlled players). Section 4 presents a possible solution method that is able to deal with these capabilities, and section 5 continues with a brief overview of experiments that compare the performance of this method with more conventional methods. In section 6, we conclude and look at future work.

## 2 Relevance of research into resource management games

We define resource management games as interactive computer games where the main problem for the player[1] is to use limited resources to construct and maintain a complex virtual environment. These games have proved to be challenging and entertaining, because of their many layers of complexity, arising from a rather small and understandable set of actions. They usually require a player to construct buildings and move units in a large grid world. While playing a resource management game, the player tries to reach a certain goal by carefully distributing limited resources. A famous example is the game SIMCITY, in which the player builds and manages a city.

Resource management games share many ideas with strategy games such as WARCRAFT [Laird and van Lent, 2001; Buro, 2004; Ponsen and Spronck, 2004], but their nature and game play differ significantly from these games, as explained below.

The main difference in nature is that in strategy games, the player constructs buildings and controls units of a virtual army in order to defeat opponent armies, whereas in resource management games, the focus is on the construction and long-term maintenance of buildings, transport networks, et cetera. Due to this difference in nature, strategy games are in use by many military organizations as a source of inspiration [Laird and van Lent, 2001], whereas resource management games receive more attention from economists and managers [Sawyer, 2002]. One might summarize that strategy games have a destructive nature, whereas resource management games have a constructive nature.

One of the two main differences in game play between the genres is the fact that strategy games are always finite games – once the enemy has been defeated, the game ends – whereas resource management games do not need to be finite, with goals such as 'build and maintain a large city' or 'transport many passengers'. A second difference in game play is that most strategy games progress in near-continuous time, hence the name real-time strategy (RTS) games [Ponsen and Spronck, 2004], whereas most resource management games progress in clearly observable discrete time. Near-continuous games, such as real-time strategy games, enable all players to move at the same time, which entails that players must possess both the capacity to respond quickly to urgent matters and the capacity to think about strategic problems. Discrete or turn-based games, such as resource management games, are similar in pace to games such as CHESS: each player takes his turn to perform a limited number of actions, and is allowed to think about which actions to perform. Because of this, the challenges posed by discrete games often require more structured thinking and less intuitive response than those posed by near-continuous games – for example, a player has to construct a sound planning towards some end goal.

There are three reasons why it is relevant to perform research in the area of AI for computer-controlled players for resource management games. First, because of the many challenges involved in playing interactive computer games, a computer-controlled player for such games must be able to deal with many aspects of human-level intelligence [Laird and van Lent, 2001]. Resource management games are able to address aspects such as high-level planning, which are not often found in other genres of interactive computer games, but rather in classical board games such as CHESS. Therefore, resource management games can be said to bridge the gap between classical games and interactive computer games. Second, problems found in many other domains (such as other genres of computer games, management and economics) closely resemble problems typically found in resource management games. Third, resource management games are being developed not only as pure entertainment, but also as educative tools. For example, Harvard's Business School is known to use resource management games in its student training program [Sawyer, 2002]. AI techniques that are able to deal with educative resource management games such as VIRTUAL U can be valuable sources of strategic information for scientists and students; in other words, the solutions AI techniques come up with, can be analysed and used as inspiration for people playing these games.

## 3 Game principles and players' capacities

A typical resource management game is played in a simulated world, usually a grid that the player looks at in bird's eye view. As time passes, a player and/or the game itself can place various kinds of structures on the grid, for example roads and buildings. The content of the grid defines the state of the game world, which is internally represented by a set of parameters. The game's dynamics are defined by a set of actions that determine the transition from the current state of the game into a new state. Both the game itself and the player can execute these actions. When playing the game, the player has to cope with many temporary objectives that eventually en-

[1] Henceforth, we will use the term 'player' to indicate both a human player and a computer-controlled player, unless indicated otherwise.

able him to reach the end goals of the game. Playing a resource management game requires thorough planning: the player must find out which actions to perform in order to eventually reach the end goals of the game, given the current state of the game world. To be able to do this, he must take into account the fact that resources and dynamical aspects play an important role[2].

Resource management games use a set of principles that lead to challenging and surprising game play for the player. We will illustrate these principles with a basic resource management game called FACTORY, in which the player can build houses and factories in order to control unemployment rates in a city. Figure 1 illustrates a possible state and interface for this game. The goal of the game is to fill the entire grid and reach a stable unemployment rate of 0%. A player can place houses and factories on the grid using the buttons BUILD HOUSE and BUILD FACTORY. The game's response to placing objects on the grid is a new value for the player's money and the current unemployment rate in the city.

The game's actual progress is determined by one or more of the principles outlined below. If we are aiming at developing a computer-controlled player for resource management games, we must keep in mind that such a player must be able to address all of these principles.

1. *Dependence on resources* – Every resource management game contains this principle, which entails that resources are limited and that there are dependencies between some of them. If for example we build one factory, for which we require 100,000 euros, at most 100 unemployed people will find a job and will start paying taxes.
2. *Dependence on location* – This principle implies that the location on the grid where we build objects influences the effect on the game world. For example, if we build a factory too close to a residential area, people will start complaining about pollution and decide to stop paying taxes, whereas if we build the factory too far from the residential area, people will refuse to go to work.
3. *Dependence on time* – This principle implies that in some cases, the effect of a certain action is delayed. For example, if we build a factory now, it will be finished in 12 months (game time).
4. *Need for planning* – A player cannot execute all possible actions immediately from the beginning of the game. For example, the player must gather at least 100,000 euros before a factory can be built. Furthermore, the game rules might stipulate that a factory can only be built if there are sufficient people available. These people must have housing, which means we can only build a factory after we have built at least a few houses for the workers to live in. Thus, if the player wants to build a factory, he must come up with a sequence of actions that makes the game progress from the current state to a state in which building a factory is possible.
5. *Competition* – In many resource management games, the player has to compete with computer-controlled opponents or co-operate with allies, sharing limited resources effectively.
6. *Multiple objectives* – Resource management games often require the player to pursue more than one objective at once in order to reach the end goals of the game. For example, in the FACTORY game, the player will be trying to build as many factories as possible, but also to keep his financial situation healthy and to outperform any competition, and he does all this to reach the game's goal: creating a city that has an unemployment rate of 0%.
7. *Non-determinism* – Many games include elements of non-determinism. Practical implementations vary from adding odd-behaving competitors or natural disasters to a game to introducing noise on the effects of rules that specify game dynamics. For example, if we build a factory, we expect 100 people to find a job, but the actual number of people that do find a job varies.

From these seven principles, we derive the following required capabilities for a player of resource management games. He must be able (1) to cope with resource dependencies, (2) to allocate space effectively, (3) to predict future game states, (4) to create a sound planning, (5) to handle competition , (6) to perform multi-objective reasoning, and (7) to deal with non-determinism. All of these capabilities are actually used for only one task: selecting actions that should be executed. In deterministic games, the sequence of actions selected by each of the players defines the outcome completely. In non-deterministic games, the sequence of actions also plays an important role in the outcome – after all, if the outcome is determined more by randomness than by the actions executed, the game is more a game of chance than a real resource management game. Thus, action selection is the core task of any resource management player, including computer-controlled players.

## 4 Methods

There are many ways of developing a computer-controlled player that can perform action selection in resource management games, using the required capacities outlined in the previous section. A possible approach is to use *hybrid AI*, which we define as AI in which elements of symbolic and behavior-based artificial intelligence are combined. We give three reasons here to support our statement that a hybrid AI approach should be suitable for the domain of resource management games.

[2] In this respect, the planning problems encountered in resource management games differ substantially from classical planning problems, that are successfully addressed by various symbolic AI techniques. In these classical planning problems, the focus is on state transitions, usually in a deterministic or even static environment without competitors.

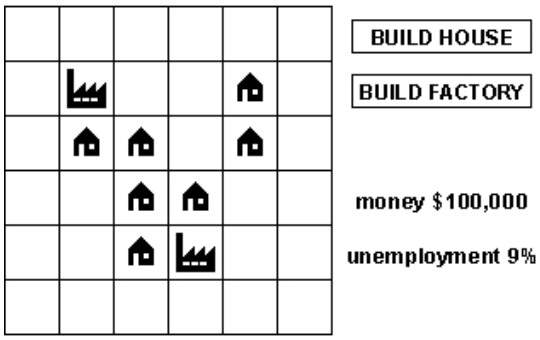


Figure 1: FACTORY: a simple resource management game.

First, in resource management games, we see that some of the required capabilities for computer-controlled players are typically provided by symbolic AI approaches, such as the ability to plan, whereas other skills are typically provided by behavior-based approaches, such as the ability to deal with non-determinism.

Second, in most resource management games, the capacity to plan is especially important for a player to be successful, as has been explained above. Intuitively, we would say that symbolic planning methods would therefore be a suitable way of dealing with resource management games [Fasciano, 1996]. However, planning in a dynamical environment with many parameters, such as a resource management game, leads to search spaces that are usually too large to handle with regular planning algorithms. Moreover, many games do not provide their players with perfect information – for example, in SIMCITY, a variety of random events takes place. Finally, if a game includes some form of competition, such as the game DUNGEON KEEPER, it becomes highly nondeterministic and requires the player to perform adversarial planning, which is harder than regular planning.

Third, research supports the power of a hybrid approach in many genres of interactive computer games. For example, a hybrid AI approach leads to satisfying performance in role playing games [Spronck *et al.*, 2003] and strategy games [Ponsen and Spronck, 2004].

In [de Jong, 2004], research is presented that compares the performance of a hybrid AI approach to that of a purely symbolic and a purely behavior-based approach, to determine whether hybrid AI is indeed a good approach of choice for resource management games. For this comparison, three solution methods have been devised, viz.

1. *a behavior-based approach using a neural network*. The input of the fully connected network consists of the values of all parameters in the game world. The output consists of a preference value for each action. Thus, in each round, the network is used to derive preferences for all actions, based on the current state of the game. The action that is most preferred is then executed, if possible. The neural network is trained by weight optimization with a genetic algorithm, by means of offline learning.
2. *an abductive planner*. This method builds a search tree, consisting of nodes representing actions and arcs representing a post-/precondition relationship between these actions (similar to figure 2). It then starts at the root of this tree and finds an action that (1) is currently executable and (2) contributes the most to the goal of the game. The latter should be determined using domain knowledge, namely a heuristic that provides the required information.
3. *evolutionary planning heuristics*. This hybrid method is a combination of the aforementioned two methods. Using a neural network as described under 1, we derive preferences for all actions, given the current state of the game. Then, these preferences are used as a heuristic for tree traversal in an abductive planner.

The approach presented under 3 follows quite intuitively from the other two approaches, since both approaches have a problem. First, the purely behavior-based approach cannot be expected to perform well in all games, because planning is a task that is too complex for a neural network. Second, as explained under 2, an abductive planner must possess domain knowledge in order to determine which action contributes most to the goal of the game. Without such domain knowledge, the planner would have to build and traverse the entire search tree, which is not a feasible task due to the size of the search space and the fact that there are non-deterministic factors. In the case of rather small single-player games, such as the games which were experimented upon, we can easily provide the planner with domain knowledge; developing a heuristic for efficient tree traversal is not difficult here. In larger games however, we would like the computer-controlled player to function without requiring additional information (such as a heuristic) programmed in ahead of time by developers. As has been mentioned earlier, many commercial games are changed frequently to resolve balance issues, and in practise, this often means that the game AI needs to be changed as well. An adaptive system has an obvious advantage here.

The evolutionary planning heuristics approach deals

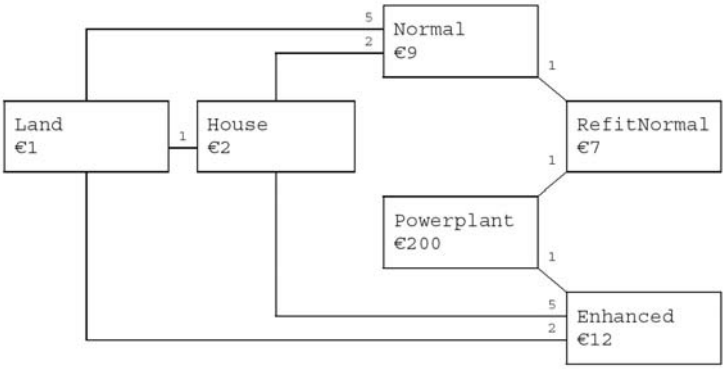


Figure 2: The schedule of actions in the experimental game PLANNING II.

with these two problems by providing a heuristic that is not developed ahead of time, but learned by repeatedly playing the game. This leads to both a less complex task for the neural network part of the method, and an adaptive heuristic for the abductive planner part.

## 5 Experiments and results

This section presents the results obtained by [de Jong, 2004] using the methods described in the previous section. The power of these methods has been assessed in small resource management games that are quite complicated to solve and representative for many games in the genre. All experiments were performed in the resource management game builder TAKING CONTROL [de Jong, 2004], which enables users to define and play their own resource management games, based on a rigid but feature-rich formalism of parameters and actions.

We will now provide a more detailed analysis of the game developed for one particular experiment, namely the game labelled PLANNING II in [de Jong, 2004]. The game is reminiscent of the FACTORY game discussed earlier. The player must build housing and factories. The location of these buildings does not effect the game's progress – only the quantity is a factor of influence. There are two types of factories, viz. coal-operated factories and electricity-operated factories – the latter are more enhanced and produce more output (i.e., more money). In order to be able to build enhanced factories, the player must first build a power plant, which is an expensive building. In addition to building new enhanced factories, it is also possible to refit existing normal, coal-operated factories, which then become enhanced ones. A schedule of actions and preconditions in this game is represented in figure 2. For example, the figure shows that in order to build a normal (factory), a player must acquire 5 pieces of land and 2 houses, and must invest 9 euros. The goal of the game is to make as much money as possible within a given number of turns, for example 300, with a given amount of money in the first round, for example 50 euros.

We will look at three interesting aspects of the game to gain insight into its complexity. First, we can observe that the game is clearly divided into three phases. In the first phase, a player must strive for as many normal factories as possible. This phase ends when the player has earned enough money to buy a power plant with. A short second phase of the game then starts, ending with the power plant being built. Then, there is a third phase in which the player must try to obtain as many enhanced factories as possible, either by refitting normal factories or by building completely new enhanced factories. Thus, in each of the phases of the game, the player must pursue different objectives, and in the last phase, there are alternatives for reaching the objective.

Second, we can observe that it is possible to determine the search space size of this game. If it is played for only 300 rounds, the search tree resulting from it contains $7^{300} \approx 3.4 * 10^{253}$ possible action sequences, since in each round, the player can choose one of the six actions or pass [de Jong, 2004]. Clearly, an undirected search process in this tree will not finish within feasible time.

Third, it is possible to derive a good solution for this game by implementing a rule-based approach with a set of intuitive rules such as: $powerplants = 1 \wedge money > 6 \wedge normalfactories > 0 \rightarrow RefitNormalFactory$. The solution obtained by this method in case of a game starting with 50 euros and running for 300 rounds, turns out to be 15,780 euros [de Jong, 2004].

In the experiment, the score of the three methods presented earlier has been compared to that of the rule-based approach. A first notable result is that the purely behavior-based approach does not find a satisfactory solution even after hundreds of generations; its best performance is a game in which nothing at all is built, resulting in a score equal to the initial amount of money, i.e., 50 euros. Depending on the heuristic used, the abductive planner is able find the same solution as the rule-based approach (15,780 euros), but as has been mentioned before, it then uses knowledge that has been added ahead of time; without such knowledge, the search space cannot be traversed efficiently by the planner. The evolutionary planning heuristics approach is also able to find this solution of 15.780 euros, but it does not require the

inclusion of additional knowledge. It does require time to learn (approximately 100 evolutionary runs of 50 individuals each), but for a problem with a search space size of $3.4 * 10^{253}$ action sequences, this can be considered to be an excellent achievement.

Four additional experiments using hybrid AI methods are described in [de Jong, 2004]. The evolutionary planning heuristics approach has been used in one of these additional experiments, with comparable results. At the time of writing, large quantities of complex resource management problems are being addressed by the three methods presented, with preliminary results indicating that evolutionary planning heuristics outperform both other methods on average, due to the fact that the heuristics found are actually better suited to the problems than the heuristics developed by hand by the developers of these problems.

All experiments performed lead to the same conclusion, namely that hybrid AI approaches are able to address many of the skills required for the direction of a computer-controlled player for resource management games, without requiring developers to program significant parts of the behavior ahead of time. Instead, the hybrid AI approach uses trial and error to find satisfactory strategies. This makes hybrid AI approaches both quite straightforward to develop and highly adaptive when it comes to changes in the game at hand. If the definition of a game changes (for example, a power plant now costs 100 euros), all we need to do is perform a new learning process which, possibly starting from the current solution method, will derive a better solution method.

## 6 Conclusions and future work

Developing computer-controlled players for resource management games is a challenging task, because these games have many layers of complexity, arising from a small and understandable set of actions. Playing a resource management game requires many skills that currently only human intelligence possesses sufficiently. Furthermore, resource management games are relevant for research because their degree of realism makes them valuable tools for educative purposes.

Exploratory research in the domain of computer-controlled players for resource management games shows that hybrid AI approaches, in which various techniques from symbolic and behavior-based AI are combined, are able to learn how to play our simple resource management game, without requiring the programming of significant elements of the player's behavior ahead of time. This indicates hybrid AI approaches might be fair choice for the development of computer-controlled players for resource management games, especially since they are straightforward to develop and robust to changes in game definitions.

For future research, it would be important to determine whether hybrid approaches can be used in more complex games than the examples presented here, such as product chain management games and games in which the location of objects built plays an important role. Moreover, the principle of competition and its relationship to adversarial planning should be addressed.

## References


[Buro, 2004] Michael Buro. Call for AI research in RTS games. In *Proceedings of the AAAI-04 workshop on AI in games, San Jose 2004*, pages 139–142, 2004.

[de Jong, 2004] Steven de Jong. Hybrid AI approaches for playing resource management games. Master's thesis, Universiteit Maastricht, the Netherlands, 2004.

[Fasciano, 1996] Mark Fasciano. Real-time case-based reasoning in a complex world. Technical report, The University of Chicago, USA, 1996.

[Jayanti, 2003] Vikram Jayanti. Game over: Kasparov and the machine. Documentary, 2003. http://www.imdb.com/title/tt0379296.

[Laird and van Lent, 2001] John Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. *AI Magazine*, Summer 2001:15–25, 2001.

[McCarthy, 1990] John McCarthy. Chess as the drosophilia of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 227–237. Springer-Verlag, 1990.

[Pfeiffer and Scheier, 1999] Rolf Pfeiffer and Christian Scheier. *Understanding Intelligence*. MIT Press, 1999.

[Ponsen and Spronck, 2004] M. Ponsen and P.H.M. Spronck. Improving adaptive game AI with evolutionary learning. In Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass, editors, *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)*, pages 389–396, Wolverhampton, UK, 2004. University of Wolverhampton.

[Russell and Norvig, 2003] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[Sawyer, 2002] B. Sawyer. Serious games: Improving public policy through gamebased learning and simulation. *Foresight and Governance Project, Woodrow Wilson International Center for Scholars*, 2002.

[Scott, 2002] B. Scott. The illusion of intelligence. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 16–20. Charles River Medias, 2002.

[Spronck *et al.*, 2003] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Improving opponent intelligence through offline evolutionary learning. *International Journal of Intelligent Games and Simulation*, Vol.2, No.1:20–27, 2003.

# Path Planning in Triangulations

**Marcelo Kallmann**
USC Institute for Creative Technologies
13274 Fiji Way
Marina del Rey CA 90292
kallmann@ict.usc.edu

## Abstract

This paper presents in detail how the techniques described in my previous work [Kallmann *et al*., 2003] can be used for efficiently computing collision-free paths in a triangulated planar environment.

The method is based on a dynamic Constrained Delaunay Triangulation (CDT) where constraints are the obstacles in the planar environment. The main advantage of relying on a triangulated domain is that the size of the adjacency graph used for searching paths is usually much smaller than in grid-based search methods. As a result much more efficient planning can be achieved.

## 1 Introduction and Related Work

The Delaunay Triangulation and its constrained version [Preparata *et al*., 1985] [Anglada, 1997] [Floriani *et al*., 1992] are important tools for representing planar domains and have been used in several applications.

This work focuses on using the Constrained Delaunay Triangulation (CDT) as the main data structure to represent planar environments composed of polygonal obstacles, in order to efficiently determine collision-free paths.

Obstacles are considered to be constraints, and specific constraint insertion and removal routines are available for updating the CDT whenever obstacles appear, disappear or change position [Kallmann *et al*., 2003].

Let $n$ be the total number of vertices in a given set of obstacles. The initial construction of the CDT can be done in optimal O($n$ log $n$) time using a divide-and-conquer algorithm [Chew, 1987]. After the initial construction, the CDT can be updated in order to reflect eventual changes in the obstacle set. For this matter, several insertion and removal routines are available in the Computational Geometry literature.

Having the CDT up-to-date, a path joining any two given points (or a failure report) is obtained in two phases. First, a graph search performed over the CDT triangles adjacency graph determines a sequence of free triangles (a *channel*) joining both points. Finally, a linear pass algorithm computes the shortest path inside channels. The obtained path is the shortest in the homotopy class [Hershberger *et al*., 1994] determined by its channel, but it might not be the globally shortest one.

The main advantage of the method is that, due to the underlying CDT, the size of the graph used for searching for paths is O($n$), reducing the time required for determining shortest paths to the time required by the graph search itself, which can be performed in O($n$ log $n$) time with any Dijkstra-type search method [Cormen *et al*., 1993]. In addition to that, as already mentioned, the CDT can be efficiently updated when obstacles change position.

The main application that motivated the development of this method is the navigation of characters in planar environments for virtual reality and game-like applications. Based on high-level decision mechanisms, characters decide where to go and determine a goal position to walk to. The path planner module is responsible to find a collision-free path towards the desired position. In such applications, handling dynamic environments and fast determination of paths are important; guaranteed shortest paths are not required.

The classical problem of finding shortest paths in the plane [Mitchell *et al*., 1998] has been studied since a long time. Efficient sub-quadratic approaches are available [Mitchell *et al*., 1996] and an optimal algorithm has been proposed taking O($n$ log $n$) time and space, where $n$ is the total number of vertices in the obstacle polygons [Hershberger *et al*., 1999].

However practical implementations are still based on grid-based search [Koenig, 2004] or on visibility graphs [Kreveld *et al*., 2000]. Unfortunately, grid-based methods lead to large graphs when fine grids are used and visibility graphs can have $\Omega(n^2)$ edges in the worst case.

Even if not popular in real applications, good alternatives are available, as the *pathnet* graph [Mata *et al*., 1997], constructed from a planar subdivision. The sparsity of

pathnets can be controlled in order to get as close as desired to the global shortest path. This control is done by choosing the number $k$ of rays emanating from the source node, resulting in a graph of size O($kn$).

The method presented herein is also based on a planar subdivision, but speed is preferred over control of the global optimality of paths. A graph of fixed size O($n$) is used, which is implicitly defined by the subdivision. This choice allows faster determination of paths and allows to dynamically update the subdivision in order to cope with dynamic environments.

## 2 Method Overview

The method can be divided in three main steps.

**Step 1** Given a set of polygonal obstacles, a CDT having as constraints the edges of the obstacles is constructed. In order to consider discs of arbitrary radius $r$, obstacles can be grown by $r$ [Laumond, 1987] before insertion in the CDT, reducing the problem to planning paths for a point [Latombe, 1991] (see Figure 1).

During run-time obstacles are allowed to be inserted, removed or displaced in the CDT as required. The CDT is able to dynamically take into account these changes and detailed algorithms are available in previous work [Kallmann *et al*., 2003].

The CDT implicitly defines the polygonal domain used by the path search. It fills with triangles both the interior and the exterior of obstacles and ensures that the edges of obstacles are also edges of the triangulation (see Figure 2). If an edge of the triangulation is an obstacle edge, the edge is said to be *constrained*. Therefore triangulation edges can be of two types: constrained or non-constrained.

**Step 2** Given two points $p_1$ and $p_2$, a graph search is performed over the adjacency graph of the triangulation, defining the shortest channel (according to the graph) connecting $p_1$ and $p_2$. This process first locates the triangle containing $p_1$, and then applies an A* search in the adjacency graph until the triangle containing $p_2$ is found. If the entire graph is searched and $p_2$ is not reached, a failure report is generated. Section 3 describes this process.

**Step 3** Obtained channels are equivalent to triangulated simple polygons, and thus the *funnel algorithm* [Chazelle, 1982] [Lee *et al*., 1984] can be applied in order to determine the shortest path joining $p_1$ and $p_2$ inside the channel. This takes linear time with respect to the number of vertices in the channel. For completeness purposes, the funnel algorithm is briefly described in section 4.

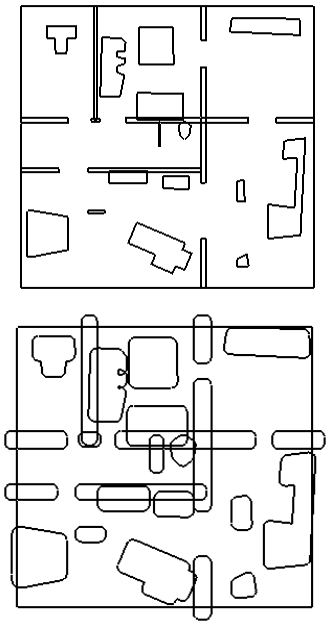

**Figure 1.** Obstacles inside a rectangular domain (top) and their grown versions (bottom). Growing obstacles ensures that paths maintain a given distance from the original objects.

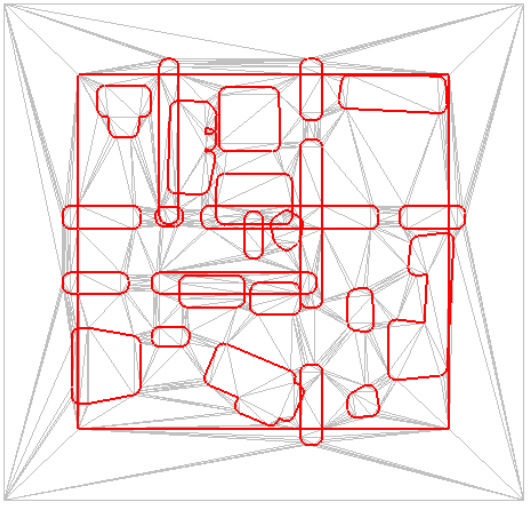

**Figure 2.** Grown obstacles inserted in the CDT. Edges of obstacles become constrained edges.

## 3 Channel Search

The polygonal domain considered by the path planner is implicitly defined as all triangles sharing non-constrained edges, starting from one given triangle.

**Point Location** Given two points $p_1$ and $p_2$, the first step is to determine the triangle $t_1$ that contains $p_1$. A point location routine is required for finding $t_1$. For robustness

purposes, the same routine may also determine if $p_1$ lies outside the CDT domain.

Good results were obtained with the visibility walk approach [Devillers, 2001]. Starting with a seed triangulation vertex *v*, one triangle *t* adjacent to *v* is selected. Then, *t* is switched to the adjacent triangle *t'* such that the common edge of *t* and *t'* divides $p_1$ and *t* in different semi planes. If more than one edge can be selected, a random choice is taken in order to avoid possible loops. Intuitively, *t'* is now closer to $p_1$ than *t*. This process of switching triangles is continuously repeated. At a certain point it is no more possible to switch of triangles, and the last triangle *t* visited contains $p_1$. Rare cases may produce an exponential search, and for avoiding that, whenever the visibility walk is detected to traverse the total number of triangles a linear search over the remaining triangles is performed. The point location routine needs to be carefully crafted, specifically in relation to the used geometric primitives.

**Graph Search** Once triangle $t_1$ is found, a graph search over the triangulation adjacency graph is performed, starting from $t_1$ until the triangle containing $p_2$ is found, without traversing constrained edges. Note that it is essential to have the triangulation described by an efficient data structure [Guibas *et al.*, 1985], permitting to retrieve all adjacency relations in constant time. This is the case not only for the graph search step, but also for several other computations presented in this paper.

The considered connectivity graph is depicted in figure 3. A starting node has the same position as $p_1$. This node is then connected to the midpoint of each non-constrained edge of triangle $t_1$. This process is continuously repeated, expanding each node of the graph to the two opposite edges of the same triangle, if these edges were not yet reached and are not constrained. At each step, the edge with less cost accumulated is selected to be expanded. The cost is the Euclidian distance measured along the graph edges. The search finishes when the triangle containing $p_2$ is reached, and the shortest channel is determined by the history of traversed triangles in the branch from $p_1$ to $p_2$.

An additional A* heuristic cost function was included based on the Euclidian distance from the current leaf node to $p_2$. Additional cost information can be associated to triangles in order to indicate, for instance, different properties of the terrain being traversed. Note that constrained edges are not expanded guaranteeing that a path will never traverse them.

The graph shown in figure 3 captures the cost of a canonical path passing through the center of the non-constrained triangulation edges. This solution has shown to be more accurate than using the center of each triangle.

At the end of the search process, a channel joining $p_1$ and $p_2$ is determined. Figure 4 illustrates such a channel. We define the first and last triangles of the channel by connecting additional edges to $p_1$ and $p_2$ respectively.

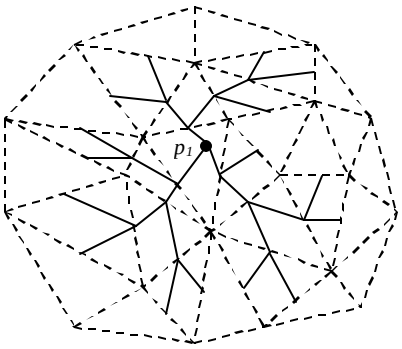


**Figure 3.** The connectivity graph (solid lines) is implicitly defined by the triangulation (dashed lines).

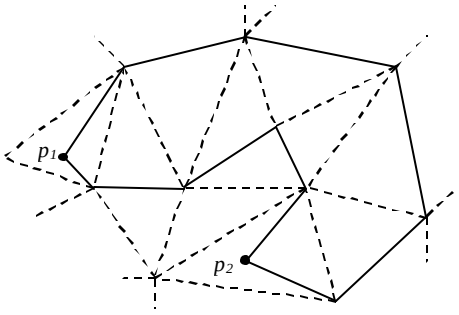


**Figure 4.** Channel (solid lines) joining point $p_1$ to $p_2$.

## 4 Paths Inside Channels

With the channel determined, the problem is now reduced to find the closest path inside a triangulated simple polygon.

For this, the *funnel algorithm* [Chazelle, 1982] [Lee *et al.*, 1984] can be applied for linearly determining the shortest path inside the channel. This algorithm is briefly reviewed here for completeness purposes, following the description of Hershberger and Snoeyink [Hershberger *et al.*, 1994].

Let *p* be a point and *uv* be a segment (figure 6). The shortest paths from *p* to *v* and from *p* to *u* may travel together for a while. At some point *a* they diverge and are concave until they reach *u* and *v*. The funnel is the region delimited by segment *uv* and the concave chains to *a*, and *a* is its *apex*. The vertices of the funnel are stored in a double-ended queue, a *deque*.

Figure 5 illustrates the insertion process of a new vertex *w*. Points from the *v* end of the deque are popped until *b* is reached, because the extension of edge *ab* is not below *w* as occurred with previous popped points. If the apex of the

previous funnel is popped during the process, then $b$ becomes the new funnel apex. Note that edge $bw$ is on the shortest path from $p$ to $w$. A similar symmetrical process is performed if the new vertex is between the extended edges of the upper concave chain of the funnel. Figures 7 and 8 show some examples of paths and channels obtained from CDTs.

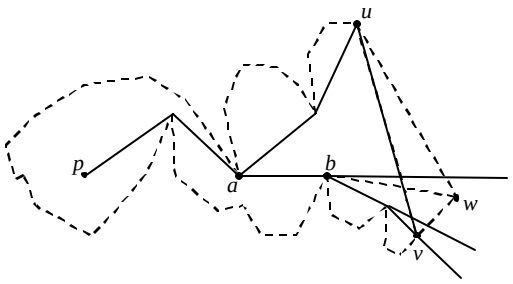


**Figure 5.** The funnel algorithm

## 5 Results and Extensions

Examples of obtained channels and paths are presented in Figures 6, 7, 8 and 9. For instance it can be noticed in Figure 6 that the number of cells (i.e. triangles) in the triangulation is much smaller than the number of cells that would be required in a fine grid. Furthermore, the contour of obstacles is precisely described and not subject to a cell resolution choice.

**Direct Visibility Test** Given points $p_1$ and $p_2$, the obtained path joining the two points is not necessarily the globally shortest one. For instance, it is possible to obtain a case where $p_1$ and $p_2$ are visible through a straight line, but the path obtained from the planner is a different path. This kind of situation mostly happens when several possible solutions with similar lengths are available, as in Figure 7. A specific *direct visibility* test was implemented in order to detect such cases. When this test is activated, before performing the graph search to find a path, a straight walk in the triangulation [Devillers, 2001] is performed. The walk consists in traversing all the triangles that are intersected by the line segment $p_1p_2$, starting from $p_1$, towards $p_2$. If during the traversal a constrained edge is crossed, the test fails. Otherwise, the triangle containing $p_2$ is reached and the test succeeds, meaning that a straight line segment is the global shortest path between $p_1$ and $p_2$.

This test has shown to be beneficial in particular for the application of controlling characters in virtual environments when users usually remark if characters don't choose a straight line path whenever possible.

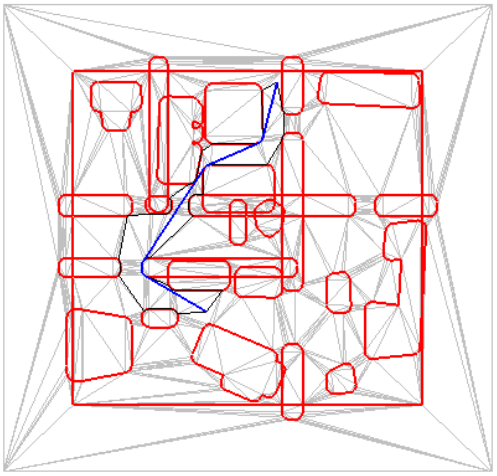

**Figure 6.** Example of a path and its channel.

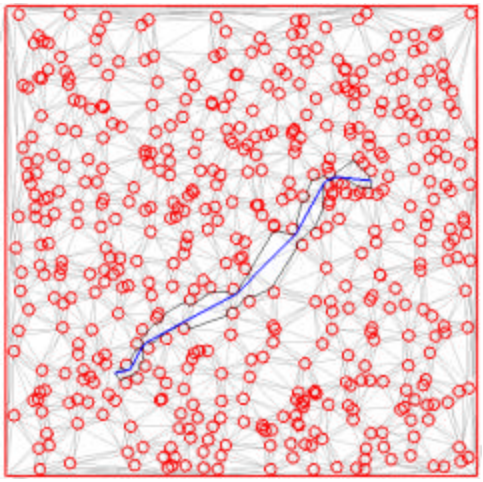

**Figure 7.** A path, its channel, and the CDT of 500 heptagons.

Other useful routines have been efficiently implemented based on the underlying CDT: ray-obstacle intersections, point-in-obstacle queries, and Boolean operation of obstacles.

Obstacles may also be defined as open polygons, i. e. as polygonal lines. Polygonal lines can be grown and inserted in the CDT, similarly to closed polygons. Figure 8 exemplifies one case based on line segments.

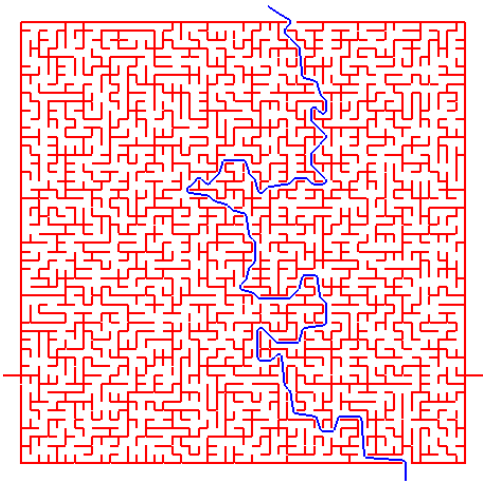

**Figure 8.** A maze composed of 2600 segments and one example path obtained. Each segment is considered to be one open obstacle (the CDT is not shown for clarity).

## 6 Conclusions

This paper presents methods for fast path planning in triangulated planar environments. The presented techniques were fully implemented.

The algorithms presented here are also useful for several other related purposes. For instance, the algorithm does not only computes shortest paths, but also the channels containing the paths. Such information can be very useful for spatial reasoning algorithms, and for bounding steering maneuvers when following planned paths.

The implemented software is being integrated with several other grid-based search methods for the purpose of evaluation, and will be soon available for research purposes.

## Acknowledgments

The project or effort described here has been sponsored by the U.S. Army Research, Development, and Engineering Command (RDECOM). Statements and opinions expressed do not necessarily reflect the position or the policy of the United States Government, and no official endorsement should be inferred.

## References

[Anglada, 1997] M. V. Anglada. "An Improved Incremental Algorithm for Constructing Restricted Delaunay Triangulations", Computer & Graphics, 21(2):215-223, 1997.

[Chazelle, 1982] B. Chazelle. A Theorem on Polygon Cutting with Applications. In Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science, 339-349, 1982.

[Chew, 1987] L. P. Chew, "Constrained Delaunay Triangulations", Proceedings of the Annual Symposium on Computational Geometry ACM, 215-222, 1987.

[Cormen *et al.*, 1993] T. Cormen, C. Leiserson, and R. Rivest. "Introduction to Algorithms", MIT Press, Cambridge, MA, 1993.

[Devillers, 2001] O. Devillers, S. Pion, and M. Teillaud, "Walking in a Triangulation", ACM Symposium on Computational Geometry, 2001.

[Floriani *et al.*, 1992] L. de Floriani and A. Puppo. "An On-Line Algorithm for Constrained Delaunay Triangulation", Computer Vision, Graphics and Image Processing, 54:290-300, 1992.

[Guibas *et al.*, 1985] L. Guibas and J. Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams", ACM Transaction on Graphics, 4:75-123, 1985.

[Hershberger *et al.*, 1994] J. Hershberger and J. Snoeyink. "Computing Minimum Length Paths of a given Homotopy Class", Computational Geometry Theory and Application, 4:63-98, 1994.

[Hershberger *et al.*, 1999] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Comput.*, 28(6):2215-2256, 1999.

[Kreveld *et al.*, 2000] M. V. Kreveld, M. Overmars, O. Schwarzkopf, and M. de Berg. "Computational Geometry: Algorithms and Applications", ISBN 3540-65620-0 Springer-Verlag, 2000.

[Latombe, 1991] J.-C. Latombe. Robot Motion Planning. Kluwer Academic Publishers, ISBN 0-7923-9206-X, Boston, 1991.

[Laumond, 1987] J.-P. Laumond, "Obstacle Growing in a Nonpolygonal World", Information Processing Letters 25, 41-50, 1987.

[Lee *et al.*, 1984] D. T. Lee and F. P. Preparata. Euclidean Shortest Paths in the Presence of rectilinear barriers. Networks. 14(3):393-410, 1984.

[Mitchell *et al.*, 1996] J. S. B. Mitchell. "Shortest paths among obstacles in the plane", International Journal on Computation Geometry Applications 6, 309-332, 1996.

[Mitchell *et al.*, 1998] J. S. B. Mitchell. "Geometric shortest paths and network optimization", in J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry, Elsevier Science, Amsterdam, 1998.

[Mata *et al.*, 1997] C. S. Mata, and J. S. B. Mitchell. "A New Algorithm for Computing Shortest Paths in Weighted Planar Subdivisions". Proceedings ACM Symposium on Computational Geometry, 264-273, Nice, France, 1997.

[Preparata *et al*., 1985] F. P. Preparata and M. I. Shamos. Computational Geometry: An Introduction. Springer-Verlag, ISBN 3540961313, 1985.

[Kallmann *et al*., 2003] M. Kallmann, H. Bieri, and D. Thalmann, "Fully Dynamic Constrained Delaunay Triangulations", In Geometric Modelling for Scientific Visualization, G. Brunnett, B. Hamann, H. Mueller, L. Linsen (Eds.), ISBN 3-540-40116-4, Springer-Verlag, Heidelberg, Germany, pp. 241-257, 2003.

[Koenig, 2004] S. Koenig, "A Comparison of Fast Search Methods for Real-Time Situated Agents", AAMAS'04, July 19-23, New York, 2004.

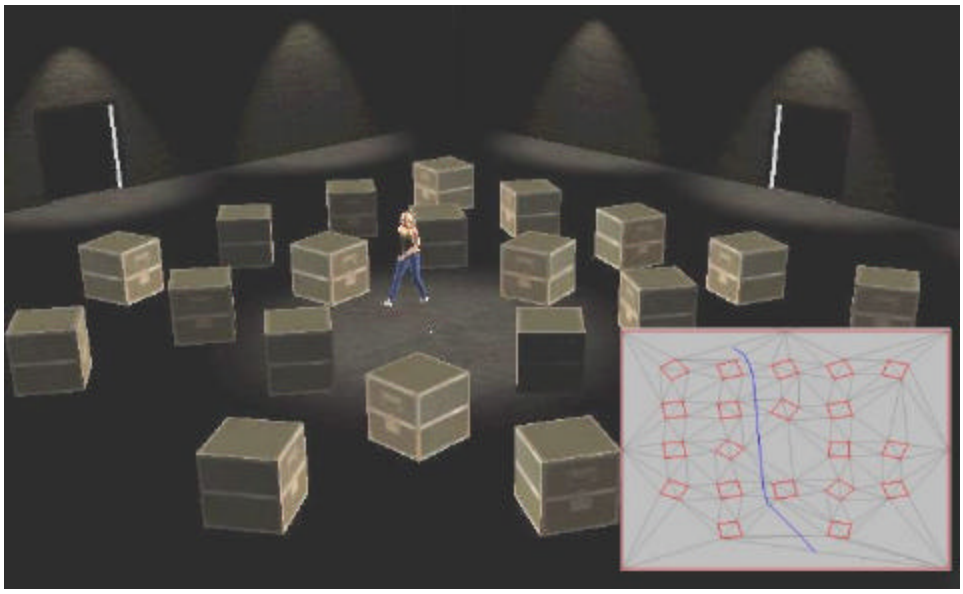

**Figure 9.** The image shows an interactive application where the virtual human is able to walk to a selected location without colliding with the boxes inside the room. Note that in this application the polygons representing the boxes are not grown before insertion in the CDT. Therefore found paths are further optimized to maintain a desired *clearance distance* from the obstacles. This illustrates a possible tradeoff between the number of triangles considered during the channel search and additional computation required to derive paths for the given channels.

# Interfacing the D'Artagnan Cognitive Architecture to the Urban Terror First-Person Shooter Game

**Bharat Kondeti, Maheswar Nallacharu, Michael Youngblood and Lawrence Holder**
University of Texas at Arlington
Department of Computer Science and Engineering
Box 19015, Arlington, TX 76019
{kondetibharat,mailbackmahesh}@yahoo.com, {youngbld,holder}@cse.uta.edu

## Abstract

The D'Artagnan Cognitive Architecture (DCA) is a multi-agent framework that supports the study of attention as a means to realize intelligent behavior by weighting the influence of different agents as they collectively determine the next action. We have interfaced the DCA to the Urban-Terror (UrT) first-person shooter game and defined several worlds of increasing complexity in order to test the DCA's ability to perform well in these worlds and demonstrate the usefulness of shifting attention among different agents. We have implemented several reflex agents and a path planner to help DCA play the UrT game. Experimental results indicate that a DCA-based player using a combination of action-determining agents can be successful when no single agent can complete the task.

## 1 Introduction

Interactive computer games have been considered human-level AI's "killer app" [Laird and van Lent, 2001] in that current games have a sufficient level of realism to require human-level intelligence to play well. Laird and van Lent's work along these lines with the SOAR cognitive architecture and the Unreal Tournament game explored the current limits of AI to play these games [Laird, 2002]. Motivated from this challenge, but with an alternative view of the design of cognitive architectures, we have begun development on the D'Artagnan Cognitive Architecture (DCA) and an interface between it and the Urban Terror (UrT) first-person shooter game. The DCA is a novel approach to cognitive architectures based on a Minskian society-of-agents approach [Minsky, 1988] of psychologically-inspired, possibly competing, agents with a global focus-of-attention influence over the agents to achieve robust, human-consistent intelligent behavior. In previous work we have presented metrics for human-consistency and comparison of human and DCA behaviors [Youngblood and Holder, 2003].

In this paper we describe the DCA, the UrT game, and the interface between the two. Fundamental to a player's performance in such environments is the ability to reason spatially. Therefore, we have also implemented a path planner based on the work of [Hill, 2002] that generates a topological graph from the UrT world maps and uses the graph to find paths between the agent's starting and goal location. While there is a large body of work in representations and algorithms for path planning in many domains (e.g., see O'Neill's [2004] work on a mesh representation for game worlds to support path planning), our work is unique in its ability to automatically generate a topological graph from the UrT's map representation, which is the means by which different UrT world scenarios are distributed to gamers.

To test the DCA approach, we define multiple tasks in five different UrT maps and evaluate the performance of a reflex-agent-based DCA while playing UrT. Our goal is to evaluate the hypothesis that the DCA consisting of multiple action-generating agents, controlled by a global attention agent, can accomplish tasks too difficult for a single-agent-based DCA. This hypothesis is similar to that confirmed in Reynolds' [1999] work, where he exhibited human-consistent steering behavior using a linear combination of numeric values from lower-level behaviors (e.g., flee danger, avoid obstacles). However, the DCA must choose among a discrete set of actions for which we propose an approach based on an adaptive focus of attention.

## 2 D'Artagnan Cognitive Architecture (DCA)

The D'Artagnan Cognitive Architecture (DCA, http://ailab.uta.edu/dca) [Youngblood, 2000; Youngblood and Holder, 2003] is based on the work of existing cognitive architectures, robotics research, and human-consistent cognitive models, centered on a specific task. DCA consists of twelve components, or models of cognition (see Figure 1). The twelve DCA models consist of the action model, action evaluation model, attention model, decision model, effectual model, emotion model, goal model, learning model, learning evaluation model, memory model, perceptual model, and reflex model. Models are implemented using one or more agents in a multi-agent framework, e.g., several different types of learning techniques may be underlying the learning model, and all compete for the bandwidth to influence DCA's behavior. When connected, these components form an architectural system for learning and adapting to an environment. Figure 1 depicts one possible set of connections between models, but in reality the models are completely connected. The communication bandwidth across a connec-

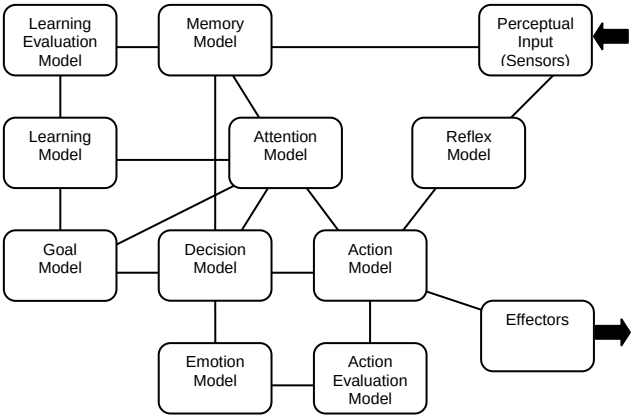


**Figure 1. The D'Artagnan Cognitive Architecture (DCA).**

tion varies dynamically, as controlled by the attention model.

When percepts are received and stored into memory, the appropriate models are triggered and start processing. Some models will select goals and propose actions based on learning from past actions, while other models will use deliberative planning to determine the next action leading toward a goal. The proposed actions are available at any time for the action model, which selects the action to take. The selected action is executed by the effectors.

Selection among the set of possible actions is affected by a number of factors. The learning model not only generates knowledge and possible actions, but also evaluates past decisions to learn the action's effectiveness for a particular goal. The emotion model provides a suppressor or enabler signal of an action based on environmental situations. Strong emotions can completely inhibit certain actions and enable others (e.g., fear will inhibit charging an opponent and enable retreat). The final dispatcher of influence is the attention model, which controls the weights of all edges and thus controls communication between models and the confidence level of possible decisions generated by these models. The attention model also controls the timing of decisions, following the anytime paradigm, to produce the best possible decision at given time intervals. Based on the human ability to focus the mind on different thought mechanisms for different situations, the attention model can stop and restart other models to enforce desired behavior.

## 3 Urban Terror (UrT)

We have begun development on interfacing DCA to a visually and tactically realistic urban warfare simulator called Urban Terror (UrT, http://www.urbanterror.net), which is built on top of the cross-platform (Win32, Linux, Mac OS X) Quake III Arena game engine. UrT is a first-person shooter (FPS) game developed by Silicon Ice Development. At present UrT is offered as an entertainment-based game, and to our knowledge, has not been deployed for any other commercial or military use. As part of this project we have

**Figure 2. Urban area map (left) used by the DCA-UrT project and a screen shot of the game (right).**

implemented an interface to the UrT game that allows the DCA (or any other system) to extract perceptual information from the UrT game and perform actions in UrT. UrT supports several challenging world maps (e.g., Figure 2 depicts an UrT urban map and a game screenshot) and game scenarios (e.g., capture the flag, bomb-defuse, and free for all). We have also defined our own simplified games within these worlds that still portray realistic urban warfare scenarios. We have developed mechanisms for logging game information in XML in order to extract a player's behavior. Eventually, we plan to have human players play our worlds in order to capture their play, which will serve as part of an evaluation metric for an agent's consistency with human behavior.

## 4 DCA-UrT Interface

The DCA and UrT are interfaced via shared memory to exchange percepts and actions. The shared memory is used to read and write percepts and actions with lower communication latency and lower computational burden on the game engine. A visual interface called UrTInterface (see Figure 3) has been developed for UrT to display all the percept information that can be obtained from the game and also acts as a virtual keyboard to play the game. This section describes the main aspects of interfacing DCA to UrT. For more interface details, see [Kondeti, 2005].

### 4.1 Modifications to UrbanTerror

Since UrbanTerror (UrT) is a realistic shooter game, with sophisticated worlds and adversaries, and DCA is still in its infancy, a number of modifications had to be done to UrT before DCA can play it. The main concern of DCA is to navigate towards the goal while avoiding obstacles. So the opponents in the game had to be removed since their goal is to kill the player and end the game. This is done by deliberately not allowing the game to start any opponents in the game, giving DCA more time to successfully navigate to the goal without getting killed by opponents.

The goal of the DCA is to get the opponent's flag. The rules for "Capture the Flag" mode in UrT require the player

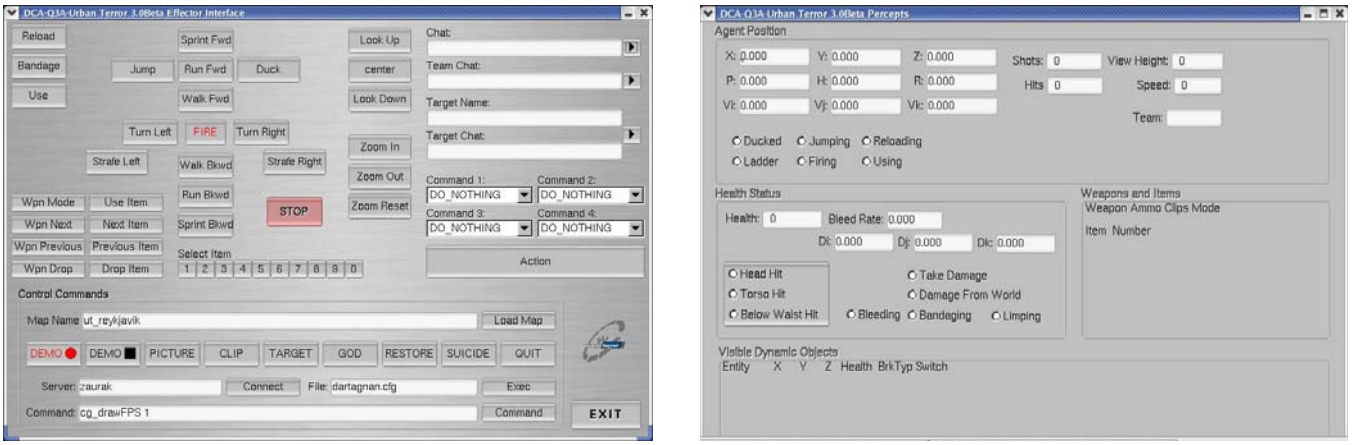


**Figure 3. DCA-UrT effector interface (left) and percept interface (right).**

to get to the opponent's flag and return back to his base without being killed or losing the flag. Since there are no opponents and the task for DCA to navigate toward the goal is itself very difficult, the complexity of the game is minimized by finishing the game once the DCA player gets the opponent flag, avoiding the burden of navigating back to the starting location with the flag.

The game is also modified to log information about the time, step, action, player health and player location, which is sufficient to determine if the DCA was able to finish the task, and if so, the time and number of actions taken to reach the goal.

## 4.2 Modifications to DCA

DCA is modified to handle the percepts obtained from UrT, consisting of 33 different percepts related to the player, 11 different percepts about entities which include opponents if they are present and all the different dynamic objects, and 4 different percepts about weapons. DCA can choose from 29 different actions that can be sent to the game.

A portion of DCA was implemented to produce a reflex agent based cognitive architecture to test the working of DCA with UrT. This includes a perceptual agent for getting information from the UrT environment, a basic memory agent, an attention agent, several reflex agents, a bread-crumb agent, a path-planning agent, an action agent and an effector agent to send actions back to UrT. The implemented portion of the DCA model for UrT is shown in Figure 4 along with the main communication links between the agents.

Only the percept agent and effector agent are interfaced to the local shared memory. All the information that is read by the percept agent from shared memory is first sent to the memory agent. The memory agent creates a data structure of all the information and places the data in the network for the reflex and path-planning agents to consume. All the reflex agents (described in the next section) process this information and send an action to the action agent. The path-planning agent determines waypoints for the BreadCrumb agent. The action agent receives actions from different reflex agents and selects an action based on a policy. Since there are no agents to evaluate the action taken, the policy is hard-coded into the action agent. Each link from a reflex agent is given a weight by the attention agent, and the action is chosen according to this weight. The action thus determined is sent to the effector agent.

## 4.3 Reflex Agents

There are four reflex agents implemented within the DCA-UrT interface. Some reflex agents maintain a small amount of state information to determine whether or not they are stuck at the same position for a while. The Random Agent depicted in Figure 4 represents one of two random reflex agents: with goal information and without goal information. For the random reflex agent with no goal location the agent randomly chooses one of the navigation actions. For the random reflex agent with goal location the agent always tries to move in a straight path towards the goal. If there are any obstacles in between, and the agent determines that it is struck at the obstacle, then the agent randomly chooses one of several evasive sequences of steps and sends them all sequentially as its actions, in order to move away from the obstacle.

The Ping Agent on the other hand knows the distance to the nearest obstacle in its line of sight. If this distance becomes less than a threshold value, the Ping Agent takes one of the sequences of random steps and continuously takes these sequences of steps until the distance to the nearest obstacle is greater than the threshold value.

For the BreadCrumb agent obstacle-free subgoals are provided which when followed take the agent to the final goal. These subgoals are chosen such that a minimum of them are required to reach the goal. These subgoals are visually identified and hard-coded into the agent, or generated by the path planning agent (see next section). Even though subgoals provided for the bread crumb agent are obstacle free, there is a possibility that the agent may get stuck at an edge of an obstacle or there might be a small obstacle in between subgoals. To accomplish the goals in such scenarios the BreadCrumb agent is played along with

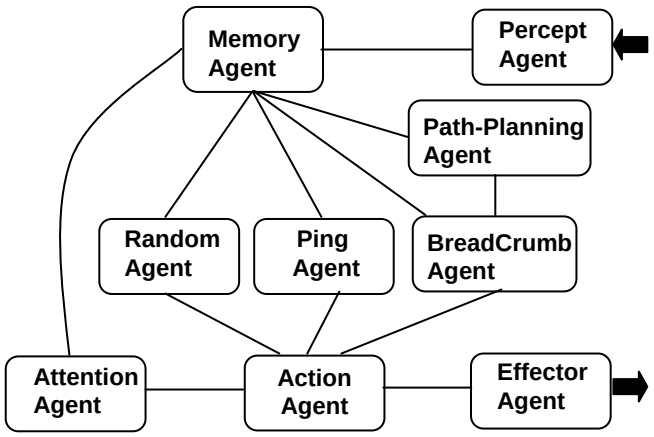


**Figure 5. DCA model for UrT interface.**

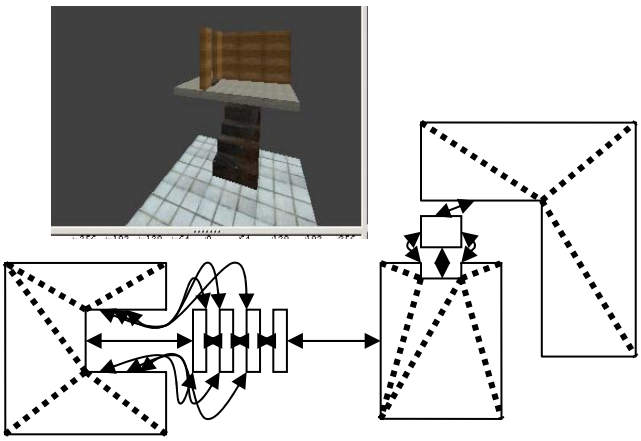

**Figure 4. Sample map (above) and its topological graph (below).**

one of the random agents. The attention agent keeps track of whether the DCA player is stuck and directs the action agent to place additional weight on the actions from the random agent to get itself free from the obstacle.

### 4.4 Path Planning Agent

The ability to navigate is a fundamental component of intelligent behavior in real-time simulated worlds. We have developed a path planning agent for the DCA-UrT interface to support the action-generating agents in their quest toward the goal. Currently, the path-planning agent supplies bread crumbs for the BreadCrumb agent to follow. The path planning system is developed in two phases. The first phase converts the UrT game world map representation into a topological graph (TOG) representation of the game world. This is a complex process, since the world map representation was not designed to be used for path planning. The second phase is the one used in the path-planning agent, i.e., it involves searching the TOG to extract path information from the graph structure. We provide an overview of the process here. For more information, see [Nallacharu, 2005].

The first phase of the path planning process consists of the following four tasks. For reference, Figure 5 shows a small world consisting of two floors connected by stairs with a wall and door on the second floor. The bottom of Figure 5 shows the final topological graph for this world.

1. *Parsing the .map file*. The Quake III family of games, of which UrT is one, describes the worlds using a .map file format. The first step is to parse this file to extract the planes, or "brushes", used to describe the world. For example, the world shown in Figure 2 (known as Reykjavík) consists of 1814 brushes.
2. *Find related brushes*. In this step we calculate the relationships (i.e., intersections) between brushes.
3. *Find brush reachabilities*. Reachability information helps decide if one can traverse from a given brush to another. The reachability represents a directed edge connecting two plane surfaces in three-dimensional space. Reachability information also includes a cost metric, which is directly proportional to the inclination and the height values of the brush planes and represents the physical difficulty in traversing between the two brushes.
4. *Generate topological graph*. The process of generating the topological graph starts at a brush and expands along reachability edges in a breadth first manner.

The bottom of Figure 5 shows the TOG for the top world. Notice that each step of the staircase is a node in the graph and the bottom three steps can all be reached directly (perhaps by jumping) from the first floor.

Given the TOG and a starting and goal location, the path-planning agent can generate a set of points along a path between the two locations. We first identify the nodes containing the desired locations by searching the graph using bounding box information. We then find the shortest path between the start and goal nodes based on the edge costs (recall that some edges cost more because they represent more complicated moves for the agent).

The next step is to determine exactly how to move based on the traversing edge in the TOG. Since the nodes represent concave surfaces, simply moving between midpoints of the common edges may not yield a successful path. So, we use a concave polygon triangulation method to divide up the nodes into convex triangles and augment the path at each edge transition with the result of a sub-search through the midpoints of each surface's triangles. The graph at the bottom of Figure 5 shows the triangulation of the node surfaces.

Finally, the points in the path are communicated to the BreadCrumb agent, which translates them into actions. These actions are sent sequentially to the action model.

## 5 Experimental Results

Two main objectives of the DCA are to show human consistency in its behavior and that a collection of possibly competing models can sometimes accomplish tasks not possible by a single model approach. We have evaluated the human-consistency of the DCA in earlier work using the Quake II game [Youngblood and Holder, 2003]. In the Quake II experiments we defined 100 levels of increasing complexity starting from a single room with the objective right in front

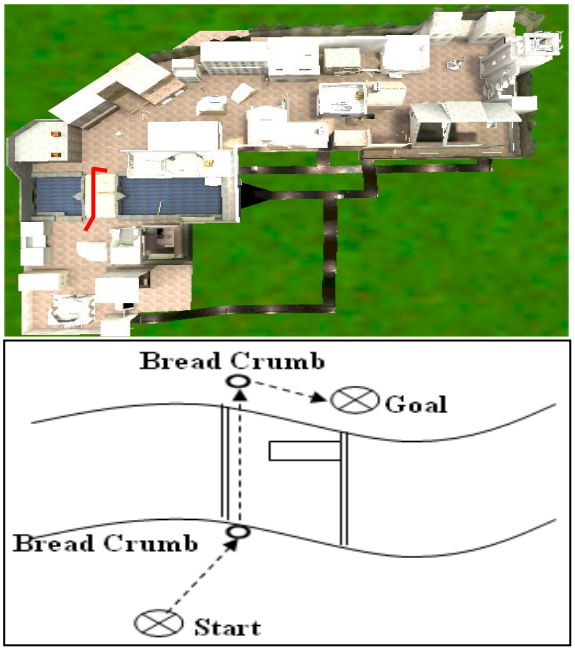


**Figure 6. Rommel map (top) and the task 4 scenario of crossing the canal bridge (bottom).**

of the agent to a complicated world with multiple adversaries. In addition to having the DCA play these levels, we also collected game play data from 24 different human players. We found that the human players clustered into three groups: novice, intermediate and expert. And we found that the DCA player was consistent with the novice-level human play according to a path edit distance based metric. The Urban Terror game provides a much more realistic urban-warfare scenario based on the Quake III Arena game engine.

The experiments reported here are designed to test the hypothesis that multiple action-generating agents working simultaneously within DCA may yield better game-playing performance than a single agent approach. The agents utilized are the two random agents (with and without goal location), the ping agent, the bread crumb agent with hand-crafted crumb locations, and the bread crumb agent with crumb locations provided by the path-planning agent.

### 5.1 Test Maps and Tasks

For our experiments we are using the following five different maps included with UrT.

1. *Reykjavik*. This map represents an urban warfare scenario and consists of four large buildings with three floors each separated by many path ways. This map is pictured in Figure 2.
2. *Rommel*. This map (see top of Figure 6) represents a village scenario where most of the buildings are destroyed. A canal flows through the village and there are many obstacles in the form of rubble.
3. *Docks*. This map depicts a warehouse scenario at the edge of a shallow body of water. This map is the most complex of all the five maps with many buildings and rooms that are interconnected in a complex manner.
4. *Riyadh*. This map portrays a desert scenario and consists of two market places far away from each other. Bezier curves are used to construct all the curved surfaces to give a desert effect.
5. *Twinlakes*. This map is built upon mountains covered with snow. The map is completely covered with trees that are climbable. The map also has two houses at either end of the map with a small pond for each house. For this map also Bezier curves were used to construct the mountainous terrain.

For each of the five maps we defined five tasks consisting of capture-the-flag games of increasing difficulty. Table 1 describes the 25 tasks.

### 5.2 Results

We evaluated the performance of DCA with various reflex agents playing individually and then two or more reflex agents playing together. The metric information used for evaluation is whether the individual reflex agents or cooperative reflex agents were able to accomplish the given task, and if so, the time taken to accomplish the task and the number of actions sent to UrT to finish the task. Each version of the DCA was allowed to play each task three times and the average of the three plays is taken for evaluation purposes.

Figure 7 shows the time take by each of the three single-agent DCAs on the task 3 scenario for the five maps. Except for Rommel, the performance of the bread crumb agent is better than all other agents, and the performance of the ping

**Table 1. Five tasks for each of the five DCA-UrT maps (obstacles are between start and goal locations).**

| | **Reykjavik** | **Rommel** | **Docks** | **Riyadh** | **Twinlakes** |
|---|---|---|---|---|---|
| **1** | Obstacle-free traversal | Obstacle-free traversal | Obstacle-free traversal | Obstacle-free traversal | Obstacle-free traversal; mountainous terrain |
| **2** | One large obstacle | L-shaped obstacle | Two adjoining obstacles with deadend | One small obstacle | One octagonal slippery-surfaced obstacle |
| **3** | Start enclosed by wall with door | Cluster of walls | Two clusters of obstacles far from each other | Platform obstacle | Row of four obstacles |
| **4** | Traverse large L-shaped alley | Cross bridge over canal (see fig. 6) | Cross narrow bridge to floating dock | Traverse around marketplace | Traverse thru tunnel |
| **5** | Obstacle-free staircase traversal | Long traversal, many obstacles | Climb stairs to second floor | Descend ladder to first floor | One room to another through narrow door |

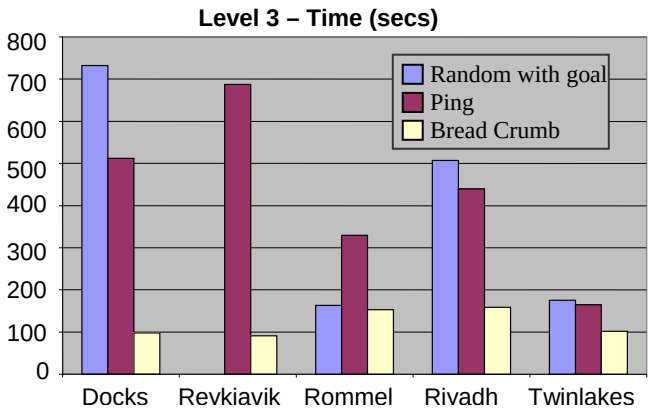


**Figure 7. Average time taken by three agents in the task 3 scenario of the five maps.**

agent is better than that of the random agent with goal information. For Reykjavik the random agent with goal information could not finish the given task, so the values for that agent are not plotted. For Rommel the performance of the random agent with goal information is better than that of the ping agent and almost equivalent to that of the bread crumb agent. This is because the obstacle present is in the form of a slanted wall, and the random agent simply slides along the wall to reach the goal.

Results for tasks 1 and 2 follow the trend of the bread crumb agent taking the least time, followed by the ping agent and then the random agent with goal information. For tasks 4 and 5 only the bread crumb agent was able to complete the levels. Task 4 typically resulted in the random and ping agents getting stuck or falling in the water. Task 5 involved finding a staircase or ladder, which the random and ping agents rarely find.

We do not show the results with the path-planning agent, because they are typically longer times, because the path planner generates many more bread crumbs, each possibly requiring slight turns. For the handcrafted bread crumbs, only 2-4 are needed for all tasks. However, there were some scenarios in which the bread crumb agent became stuck; whereas, the path-planner agent was able to successfully complete the task. In these same cases where the bread crumb agent got stuck, we allow the attention agent to weight the random agents' actions more heavily, which was typically enough to get the player unstuck. Once unstuck, the attention agent readjusted the weight back in favor of the bread-crumb agent in order to complete the task. While this policy was hard-coded, it illustrates how multiple agent approach can combine to accomplish tasks not possible by a single agent approach and illustrates an area in which the DCA can focus efforts to learn such a policy.

## 6 Conclusions

We have successfully integrated the D'Artagnan Cognitive Architecture (DCA) to the Urban Terror (UrT) first-person shooter game to support the further development of the DCA. The definition of increasingly difficult tasks within the realistic maps provided with UrT comprises a challenging evaluation testbed for the DCA and other AI methods. The implementation and combination of the path planner and reflex agents provides a simple, yet effective, agent for playing our simplified UrT game.

We plan to pursue this work along three directions. First, we will further develop the various components of the DCA based on the latest understanding from cognitive and neurosciences. We will also interface and evaluate the DCA in other environments. Our goal is not only to produce a DCA-based agent that performs well in complicated environments, but that also exhibits human-consistent behavior. Second, we will extend the testbed to include additional tasks of increasing complexity by combining components of existing tasks and introducing adversaries. Third, we will make the UrT interface and task set available to others as a testbed for evaluating AI methods and as a mechanism to collect human play to serve as a baseline for measuring human consistency.

## References

[Hill, 2002] R. Hill, C. Han and M. van Lent. Applying perceptually driven cognitive mapping to virtual urban environments. *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pp. 886-893, 2002.

[Kondeti, 2005] B. Kondeti. Integration of the D'Artagnan Cognitive Architecture with Real-Time Simulated Environments. M.S. thesis, Department of Computer Science and Engineering, University of Texas at Arlington, 2005.

[Laird, 2002] J. Laird. Research in Human-Level AI Using Computer Games. *Communications of the ACM*, 45(1): 32-35, 2002.

[Laird and van Lent, 2001] J. Laird and M. van Lent. Human-level AI's Killer Application: Interactive Computer Games. *AI Magazine*, 22:15-25, 2001.

[O'Neill, 2004] J. O'Neill. Efficient Navigation Mesh Implementation. *Journal of Game Development*, Volume 1, Issue 1, 2004.

[Nallacharu, 2005] M. Nallacharu. Spatial Reasoning for Real-Time Simulated Environments. M.S. thesis, Department of Computer Science and Engineering, University of Texas at Arlington, 2005.

[Reynolds, 1999] C. Reynolds. Steering Behaviors for Autonomous Characters. Proceedings of the Game Developers Conference, pp. 763-782, 1999.

[Youngblood, 2002]. G. M. Youngblood. Agent-Based Simulated Cognitive Intelligence in a Real-Time First-Person Entertainment-Based Artificial Environment. M.S. thesis, Department of Computer Science and Engineering, University of Texas at Arlington, 2002.

[Youngblood and Holder, 2003] G. M. Youngblood and L. B. Holder. Evaluating Human-Consistent Behavior in a Real-time First-person Entertainment-based Artificial Environment. *Proceedings of the Sixteenth International FLAIRS Conference*, pp. 32-36, 2003.

# Knowledge-Based Support-Vector Regression for Reinforcement Learning

**Richard Maclin[†], Jude Shavlik[‡], Trevor Walker[‡], Lisa Torrey[‡]**
University of Minnesota – Duluth[†]
Computer Science Department
1114 Kirby Dr, Duluth, MN 55812
rmaclin@d.umn.edu

University of Wisconsin – Madison[‡]
Computer Sciences Department
1210 W Dayton St, Madison, WI 53706
{shavlik,torrey,twalker}@cs.wisc.edu

## Abstract

Reinforcement learning (RL) methods have difficulty scaling to large, complex problems. One approach that has proven effective for scaling RL is to make use of advice provided by a human. We extend a recent advice-giving technique, called Knowledge-Based Kernel Regression (KBKR), to RL and evaluate our approach on the *KeepAway* subtask of the RoboCup soccer simulator. We present empirical results that show our approach can make effective use of advice. Our work not only demonstrates the potential of advice-giving techniques such as KBKR for RL, but also offers insight into some of the design decisions involved in employing support-vector regression in RL.

## 1 Introduction

Reinforcement learning (RL) techniques such as *Q*-learning and SARSA [Sutton and Barto, 1998] are effective learning techniques, but often have difficulty scaling to challenging, large-scale problems. One method for addressing the complexity of such problems is to incorporate advice provided by a human teacher. The approach of using advice has proven effective in a number of domains [Lin, 1992; Gordon and Subramanian, 1994; Maclin and Shavlik, 1994; Andre and Russell, 2001; Kuhlmann et al., 2004].

Recently, Mangasarian et al., [2004] introduced a method called Knowledge-Based Kernel Regression (KBKR) that allows a kernel method to incorporate advice given in the form of simple IF-THEN rules into a support vector method for learning regression (i.e., real-valued) problems. Their technique proved effective on the simple regression problems on which they tested it. In this article we extend the general KBKR approach to RL and test it on a complex game from the RoboCup soccer simulator [Noda et al., 1998] – *KeepAway* [Stone and Sutton, 2001].

In applying the KBKR approach to *KeepAway* we found that we had to make a number of adjustments and extensions, both to the KBKR method and to our representation of the problem. These adjustments and extensions proved critical to effectively learning in the *KeepAway* task.

In the next section of the paper we present the basic KBKR method. In Section 3 we present the RoboCup simulator and *KeepAway* in particular, and discuss some of the difficulties for this game. In Section 4 we discuss how a support-vector regressor can be used in *Q*-learning and then present our reformulation of the KBKR method and some of the issues we had to address in order to get good results. Following that we present results of experiments using our new approach on *KeepAway*. The final sections discuss related research, future directions, and conclusions.

## 2 Knowledge-Based Support Vector Regression

In this section we present the basics of Knowledge-Based Kernel Regression [Mangasarian et al., 2004].

### 2.1 Support Vector Regression

A linear regression problem involves trying to find a set of weights ($w$) and a offset ($b$) to learn a function of the form $f(x) = w^T x + be$, where $T$ indicates the transpose of a vector, $x$ is a vector of numeric features describing a particular instance (e.g., values describing the soccer field as seen from the player's point of view), $f(x)$ is the value that instance is labeled with (e.g., the $Q$ value of taking action *HoldBall*), and $e$ denotes a vector of ones. From now on, for clarity we will omit $e$, with the understanding that $b$ is a scalar.

For a particular set of observed input vectors (a set of states observed during learning) and a corresponding set of $f(x)$ values (the current $Q$ estimates for each of those states), we find a solution to the equation:

$$Aw + b = y$$

where $A$ is the set of states, one row for each input vector, one column for each feature, and $y$ is the set of expected $f(x)$ values, one for each input vector. Since an exact solution is often infeasible, this equation is generalized to:

$$Aw + b \approx y \quad (1)$$

Solutions to this problem are ranked by how well they meet some performance criterion (such as minimum error with respect to the $y$ values).

In a kernel approach, the weight vector $w$ is replaced with its dual form $A^T\alpha$, which converts Eq. 1 to:

$$AA^T\alpha + b \approx y$$

This formulation is then generalized by replacing the $AA^T$ term with a kernel, $K(A,A^T)$, to produce:

$$K(A,A^T)\alpha + b \approx y \quad (2)$$

However, in this article we simply use Eq. 1 above, since linear models are more understandable and scale better to

large numbers of training examples. We use *tile coding* (an example is shown below) to produce the non-linearity in our models. The reader should note that using Eq. 1 is not identical to simply using a linear kernel (*K*) in Eq. 2.

To use a linear programming (LP) method to learn a model we simply have to indicate the expression to be minimized when producing a solution. One common formulation for linear regression models is the following, which we call LP1 so we can refer to it again later:

$$\min_{s \geq 0} \| w \|_1 + \| b \|_1 + C \| s \|_1$$
$$s.t. \quad -s \leq Aw + b - y \leq s$$

In this formulation we use *slack* variables *s* to allow the solution to be inaccurate on some training examples, and we penalize these inaccuracies in the objective function that is to be minimized. We then minimize a weighted sum of the *s* slack terms and the absolute value of weights and the *b* term (the *one-norm*, $\|\cdot\|_1$, computes the sum of absolute values). This penalty on weights (and *b)* penalizes the solution for being more complex. *C* is a parameter for trading off how inaccurate the solution is (the *s* terms) with how complex the solution is (the weights and *b*). The resulting minimization problem is then presented to a linear program solver, which produces an optimal set of *w* and *b* values.

### 2.2 Knowledge-Based Kernel Regression

In KBKR, a piece of advice or domain knowledge is represented in the notation:

$$Bx \leq d \quad \Rightarrow \quad f(x) \geq h^T x + \beta \qquad (3)$$

This can be read as:

> *If certain conditions hold ($Bx \leq d$), the output, f(x), should equal or exceed some linear combination of the inputs ($h^T x$) plus a threshold term ($\beta$).*

The term $Bx \leq d$ allows the user to specify the region of input space where the advice applies. Each row of matrix *B* and its corresponding *d* values represents a constraint in the advice. For example, a user might give the rule:

IF ($distanceA + 2\ distanceB$) $\leq 10$
THEN $f(x) \geq 0.5\ distanceA + 0.25\ distanceB + 5$

For this IF-THEN rule, matrix *B* would have a single row with a 1 in the column for feature *distanceA* and a 2 in the column for *distanceB* (the entry for all other features would be 0), and the *d* vector would be a scalar with the value 10.

In general, the rows of *B* and the corresponding *d* values specify a set of linear constraints that are treated as a conjunction and define the polyhedral region of the input space to which the right-hand side of the advice applies. The vector *h* and the scalar β then define a linear combination of input features that the predicted value *f(x)* should match or exceed. For the above rule, the user advises that when the left-hand side condition holds, the value of *f(x)* should be greater than ½ *distanceA* plus ¼ *distanceB* plus 5. This would be captured by creating an *h* vector with coefficients of 0.5 and 0.25 for the features *distanceA* and *distanceB* (0 otherwise), and setting β to 5.

In this advice format, a user in a reinforcement-learning task can define a set of states in which the *Q* value for a specific action should be high (or low). We later discuss how we numerically represent "high Q."

Mangasarian et al. prove that the advice implication in Eq. 3 is equivalent to the following set of equations having a solution (we have converted to non-kernel form):

$$B^T u + w - h = 0, \quad -d^T u + b - \beta \geq 0, \quad u \geq 0 \qquad (4)$$

"Softening" the first two of these leads to the following optimization problem in the case of linear models (LP2):

$$\min_{s \geq 0, u \geq 0, z \geq 0, \zeta \geq 0} \| w \|_1 + \| b \|_1 + C \| s \|_1 + \mu_1 \| z \|_1 + \mu_2 \zeta$$
$$s.t. \quad -s \leq Aw + b - y \leq s$$
$$-z \leq B^T u + w - h \leq z$$
$$-d^T u + \zeta \geq \beta - b$$

The *z* and $\zeta$ are slack terms associated with the advice; they allow Eq. 4 to be only approximately satisfied. The $\mu_1$ and $\mu_2$ parameters specify how much to penalize these slacks. In other words, these slacks allow the advice to be only partially followed by the learner.

Mangasarian et al., [2004] tested their method on some simple regression problems and demonstrated that the resulting solution would incorporate the knowledge. However, the testing was done on small feature spaces and the tested advice placed constraints on *all* of the input features. In this article we apply this methodology to a more complex learning problem based on RL and the RoboCup simulator.

## 3 RoboCup Soccer: The Game *KeepAway*

We experimented on the game *KeepAway* in simulated RoboCup soccer [Stone and Sutton, 2001]. In this game, the goal of the *N* "keepers" is to keep the ball away from *N-1* "takers" as long as possible, receiving a reward of 1 for each time step they hold the ball (the keepers learn, while the takers follow a hand-coded policy). Figure 1 gives an example of *KeepAway* involving three keepers and two takers.

To simplify the learning task, Stone and Sutton chose to have learning occur only by the keeper who currently holds the ball. When no player has the ball, the nearest keeper pursues the ball and the others perform hand-coded moves to "get open" (be available for a pass). If a keeper is holding the ball, the other keepers perform the "get open" move.

The learnable action choice then is whether to hold the ball or to pass it to another keeper. Note that passing requires multiple actions in the simulation (orienting the body, then performing multiple steps of kicking), but these low-level actions are managed by the simulator and are not addressed in Stone and Sutton's, nor our, experiments.

The policy of the takers is simple; if there are only two takers they pursue the ball. When there are more than two takers, two pursue the ball and the others "cover" a keeper.

For our work we employ the feature representation used by Stone and Sutton. They measure 13 values that define the state of the world from the perspective of the keeper that currently has the ball. These 13 features record geometric properties such as the pair-wise distances between players and the angles formed by sets of three players.

The task is made more complex because the simulator incorporates noise into the information describing the state. In addition, the actions of the agents contain noise. For example, there is a chance the keeper passing the ball to another keeper will misdirect the ball, possibly sending it out

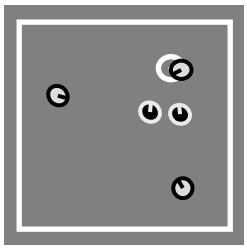

**Figure 1. A sample KeepAway game where there are three keepers (light gray with black outlines), two takers (black with gray outlines), and the ball (currently held by the keeper in the upper right). A game continues until one of the takers holds the ball for at least 5 time steps (0.5 sec) or if the ball goes out of bounds (beyond the white lines).**

of bounds or towards one of the takers. The overall score of a keeper team is measured in terms of how long they are able to hold onto the ball.

Stone and Sutton [2001] demonstrated that this task can be learned with RL. They employed SARSA learning with replacing eligibility traces, and used CMAC's as their function approximator. They used a tile encoding of the state space, where each feature is discretized several times into a set of overlapping bins. For example, one could divide a feature that ranges from 0 to 5 into four overlapping bins of width 2: one bin covering values [0,2], one [1,3], one [2,4] and one [3,5]. This representation proved very effective in their experiments and we use it also.

## 4 Using KBKR for KeepAway

In order to use regression for RL we must formulate the problem as a regression problem. We represent the real-valued *Q* function as a set of learned models, one for each action. The input to each model is the state and each model makes a prediction of the *Q* value for the action. We use one-step SARSA to estimate the *Q* value.

Since incremental training algorithms are not well developed for support vector machines, we employ batch training. We save the series of states, actions, and reinforcements experienced over each set of 100 games, we then stop to train our models, and then use the resulting models in the next chunk of 100 games. When we create training examples from old data, we use the *current* model to compute the *Q* values for the next state in one-step *SARSA* learning since these estimates are likely to be more accurate than those obtained when these old states were actually encountered.

This batch approach is effective but leads to a potential problem. As the game continues data accumulates, and eventually the sets of constraints in LP1 and LP2 become intractably large for even commercial LP solvers. In addition, because the learner controls its experiences, older data is less valuable than newer data.

Hence, we need a mechanism to choose which training examples to use. We do this by taking a stochastic sample of the data. We set a limit on the number of training examples (we currently set this limit to 1500 and have not experimented with this value). If we have no more examples than this limit, we use them all. When we need to discard some examples, we keep a (uniformly) randomly selected 750 (i.e., half our limit) and discard others according to their age. The probability we select an as yet unselected example is $\rho^{age}$ ($\rho$ raised to the power *age*) and we set $\rho$ to a value in [0,1] to produce a data set of our specified maximum size.

In our initial experiments on *KeepAway* we employed kernel-based regression directly using the 13 numeric features used by Stone and Sutton without tile encoding. In these experiments, we found that both Gaussian and linear kernels applied to just these 13 features performed only slightly better than a random policy (the results stay right around 5 seconds – compare to Figure 3's results).

Using Stone and Sutton's tile coding led to substantially improved performance, and we use that technique for our experiments. We provide to our learning algorithms *both* the 13 "raw" numeric features as well as binary features that result from (separately) tiling each of the 13 features. We create 32 binary features per raw feature. We keep the numeric features to allow our methods to explore a wide range of possible features and also since the numeric features are more easily expressed in advice.

One *critical* adjustment we found necessary to add to the KBKR approach (LP2) was to append additional constraints to the constraints defined by the *B* matrix and *d* vector of Eq. 3. In our new approach we added for each feature not mentioned in advice constraints of the form:

$$\min(feature_i) \le feature_i \le \max(feature_i)$$

For example, if *distanceC* ranges from 0 to 20, we add the constraint: $0 \le distanceC \le 20$.

This addresses a severe limitation in the original KBKR method. In the original KBKR approach the advice, when unmodified with slack variables, implies that the right-hand side must be true in *all* cases (no matter what the values of the other features are). For example, if we advise "when *distanceA* is less than 10 we want the output to be at least 100," but do not place any restrictions on the other features' values, the KBKR algorithm cannot include any other features in its linear model, since such a feature could hypothetically have a value anywhere from $-\infty$ to $+\infty$, and one of these extremes would violate the THEN part of advice. By specifying the legal ranges for all features (including the Booleans that result from tiling), we limit the range of the input space that KBKR has to consider to satisfy the advice.

For this reason, we also automatically generate constraints for any of the binary features constrained to be true or false by advice about numeric feature. For example, assume the advice says *distanceA>10*. We then add constraints that capture how this information impacts various tiles. If we had two tiles, one checking if *distanceA* is in [0,10], and a second checking if *distanceA* is in [10,20], we would add constraints that the first tile must be false for this advice and the second tile must be true.

If we tile a feature into several bins where more than one tile might match the constraint – imagine that *distanceA* was divided into tiles [0,5], [5,10], [10,15] and [15,20] – we would add constraints indicating that each of the first two must be false for the constraint (*distanceA>10)* and *one* of the last two must be true. This last constraint equation (that one of the last two tiles must be true) would be:

$$distanceA[10,15] + distanceA[15,20] = 1$$

where *distanceA[X,Y]* denotes the feature value (0 or 1) for the tile of *distanceA* covering the range *[X,Y]*.

In cases where a tile only partially lines up with a constraint included in advice – for example if *distanceA* was covered by tiles [0,4], [4,8], [8,12], [12,16] and [16,20] and the advice included *distanceA>10*, we would still add constraints to indicate that the first two tiles ([0,4] and [4,8]) must be false and then add a constraint that one of the other three tiles must be true (as in the equation shown above). These cases will often occur, since we cannot count on advice lining up with our tiling scheme. Since we conjoin the constraints on the tiles with the original constraint on the numeric feature, it is safe to include tiles that span beyond the original advice's constraint on the numeric feature.

We also needed to extend the mechanism for specifying advice in KBKR in order to apply it to RL. In the original KBKR work the output values are constrained by a linear combination of constants produced by the user (see Eq. 3). However, in RL advice is used to say when the *Q* for some action should be *high* or *low*. So we need some mechanism to convert these terms to numbers. We could simply define *high* to be, say, $\geq 100$ and *low* to be $\leq 50$, but instead we decided to let the training data itself specify these numbers. More specifically, we allow the term *averageQ* to be used in advice and this value is computed over the examples in the training set. Having this term in our advice language makes it easier to specify advice like "in these states, this action is 10 units better than in the typical state."

As briefly mentioned earlier, our linear programs penalize the *b* term in our models. In our initial experiments, however, we found the *b* term led to underfitting of the training data. Recall that our training data for action *A*'s model is a collection of states where *A* was applied. As training progresses, more and more of the training examples come from states where executing action *A* is a good idea, (i.e., action *A* has a high *Q* value in these states). If the *b* term is used in the learned models to account for the high *Q* values, then when this model is applied to a state where action *A* is a bad choice, the predicted *Q* may still be high.

For instance, imagine a training set contains 1000 examples where the *Q* is approximately 100 and 10 examples where the *Q* is 0. Then the constant model $Q = 100$ might be the optimal fit to this training set, yet is unlikely to lead to good performance when deployed in the environment.

One way to address this weakness is to include in the training set more states where the *Q* for action *A* is low, and we partly do this by keeping some early examples in our training set. In addition we address this weakness by highly penalizing non-zero *b* terms in our linear programs (for clarity we did not explicitly show a scaling factor on the *b* term earlier in our linear programs). The hypothesis behind this penalization of *b* is that doing so will encourage the learner to instead use weighted feature values to model the *Q* function, and since our objective function penalizes having too many weights in models, the weights used to model the set of high *Q* values will have captured something essential about these states that have high *Q*'s, thereby generalizing better to future states. Our improved empirical evidence after strongly penalizing *b* supports our hypothesis. In general, one needs to carefully consider how to choose training examples when using a non-incremental learner in RL.

## 5 Experimental Results

We performed experiments on the *KeepAway* task using our approach for incorporating advice into a learner via the KBKR method. As an experimental control, we also consider the same support-vector regressor but without advice. In other words, LP2 described in Section 2 is our main algorithm and LP1 is our experimental control, with both being modified for RL as explained in Section 4. We measure our results in terms of the length of time the keepers hold the ball. Our results show that, on average, a learner employing advice will outperform a learner not using advice.

### 5.1 Methodology

Our experiments were performed on 3 versus 2 *KeepAway* (3 keepers and 2 takers). The takers employed a fixed policy as described in Section 3. The keepers were all learning agents and pooled their experience to learn a single model which is shared by all of the keepers.

The reinforcement signals the learners receive are 0.1 for each step in the game and a 0 when the game ends (when the takers control the ball or the ball goes out of bounds). Our discount rate is set to 1, the same value used by Stone and Sutton [2001]. For our action-selection process we used a policy where we performed an exploitation action (i.e., chose the action with the highest value) 99% of the time and randomly chose an action (exploration) the remaining time, again following Stone and Sutton. We report the average total reinforcement for the learners (the average time the keepers held the ball) over the previous 1000 games.

We set the values of *C*, $\mu 1$, and $\mu 2$ in LP1 and LP2 to be 100/#examples, 10, and 100 respectively. By scaling *C* by the number of examples, we are penalizing the *average* error on the training examples, rather than the *total* error over a varying number of examples. Since the number of weights is fixed in LP1 and LP2, we do not want the penalty due to data mismatch to grow as the number of training examples increases. We tried a small number of settings for *C* for our non-advice approach (i.e., our experimental control) and found this value worked best. We use this same value

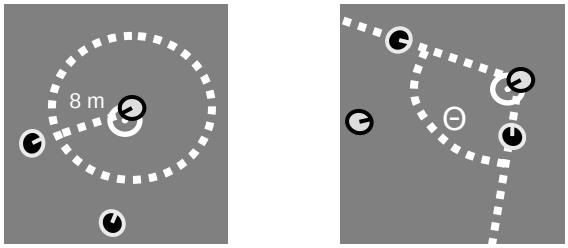


**Figure 2. The two pieces of advice involve a suggestion when to hold the ball (if the nearest taker is at least 8m away) , and when to pass the ball (if a taker is closing in, the teammate is further away than the takers and there is a large passing lane - the value of Θ is ≥ 45°).**

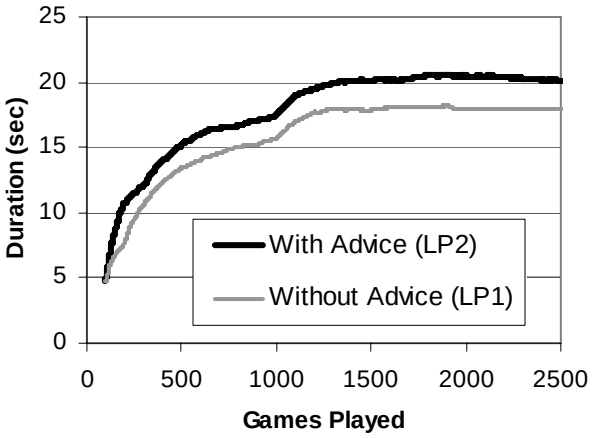


**Figure 3. Results of standard support vector linear regression versus a learner that receives at the start of learning the advice described in the text.**

for our KBKR approach. We simply chose the $\mu 1$ and $\mu 2$ values and have not experimented with different settings.

Each point in our results graphs is averaged over ten runs. The results are reported as a function of the number of games played, although since games are of different length, the amount of experience differs. This result is somewhat mitigated in that we provide at most 1500 state and action pairs to our learners, as discussed above.

### 5.2 Advice We Used

We employed two pieces of advice. The advice is based on advice used in Kuhlman et al., [2004]. The first rule suggests the keeper with the ball should hold it when the nearest taker is at least 8m away (see Fig. 2 left). When this advice applies, it suggests the *Q* for holding should exceed the average for holding by 1 second for each meter the closest taker is beyond 8 meters. The advice in our notation is:

IF *distanceNearestTaker* $\geq$ *8*
THEN *Q(hold)* $\geq$ *averageQ + distanceNearestTaker - 8*

The second piece of advice indicates when to pass the ball (see Figure 2 right). This advice tests whether there is a taker closing in, whether there is a teammate that is further away than either of the takers and whether there is a passing lane (a value of $\Theta$ that is at least 45 degrees) to that teammate. When this advice applies, it suggests that the *Q* for passing to the nearest teammate exceeds the average by 0.1 seconds for each degree (up to 60 degrees, and by 6 seconds for angles larger than 60 degrees).

### 5.3 Results and Discussion

Fig. 3 presents the results of our experiments. These results show that a learner with advice obtains gains in performance due to that advice and retains a sizable advantage in performance over a large number of training examples. Figure 3 indicates that advice-taking RL based on KBKR can produce significant improvements for a reinforcement learner ($p < 0.01$ for an unpaired *t*-test on the performance at 2500 games played). Although other research has demonstrated the value of advice previously (see next section), we believe that the advantages of using a support-vector based regression method make this a novel and promising approach.

Our results are not directly comparable to that in Stone and Sutton [2001] because we implemented our own RoboCup players, and their behavior, especially when they do not have the ball, differs slightly. We also tile features differently than they do. Stone and Sutton's learning curves start at about 6 seconds per game and end at about 12 after about 6000 games (our results are initially similar but our keepers games last longer – possibly due to somewhat different takers). We have not implemented Stone and Sutton's method, but our LP1 is a good proxy for what they do. Our focus here is on the relative impact of advice, rather than a better non-advice solution.

## 6 Related Work

A number of researchers have explored methods for providing advice to reinforcement learners. These include methods such as replaying teaching sequences [Lin, 1992], extracting information by watching a teacher play a game [Price and Boutilier, 1999], and using advice to create reinforcement signals to "shape" the performance of the learner [Laud and DeJong, 2002]. Though these methods have a similar goal of shortening the learning time, they differ significantly in the kind of advice provided by the human.

Work that is more closely related to the work we present here includes various techniques that have been developed to incorporate advice in the form of textual instructions (often as programming language constructs). Gordon and Subramanian [1994] developed a method that used advice in the form IF *condition* THEN *achieve goals* that adjusts the advice using genetic algorithms. Our work is similar in the form of advice, but we use a significantly different approach (optimization by linear programming) to incorporate advice.

In our previous work [Maclin and Shavlik, 1994], we developed a language for providing advice that included simple IF-THEN rules and more complex rules involving multiple steps. These rules were incorporated into a neural network, which learned from future observations. In this earlier work new hidden units are added to the neural network that represent the advice. In this article, a piece of advice represents constraints on an acceptable solution.

Andre and Russell [2001] developed a language for creating RL agents. Their language allows a user to specify partial knowledge about a task using programming constructs to create a solver, but also includes "choice" points where the user specifies *possible* actions. The learner then acquires a policy to choose from amongst the possibilities. Our work differs from theirs in that we do not assume the advice is correct.

In Kuhlmann et al., [2004], advice is in the form of rules that specify in which states a given action is good (or bad). When advice is matched, the predicted value of an action in that state is increased by some fixed amount. Our work differs from this work in that our advice provides constraints on the *Q* values rather than simply adding to the *Q* value. Thus our learner is better able to make use of the advice when the advice is already well represented by the data. We

tested the Kuhlmann et al. method with our no-advice algorithm (LP1), but found it did not improve performance.

A second area of related research is work done to employ support vector regression methods in RL agents. Both Dietterich and Wang [2001] and Lagoudakis and Parr [2003] have explored methods for using support vector methods to perform RL. The main limitation of these approaches is that these methods assume a model of the environment is available (or at least has been learned) and this model is used for simulation and *Q*-value inference. Our approach is a more traditional "model-free" approach and does not need to know the state-transition function.

## 7 Conclusions and Future Directions

We presented and evaluated an approach for applying Mangasarian et al.'s [2004] Knowledge-Based Kernel Regression (KBKR) technique to RL tasks. In our work we have investigated the strengths and weaknesses of KBKR and have developed adjustments and extensions to that technique to allow it to be successfully applied to a complex RL task. Our experiments demonstrate that the resulting technique for employing advice shows promise for reducing the amount of experience necessary for learning in such complex tasks, making it easier to scale RL to larger problems. The key findings and contributions of this paper are:

1. We demonstrated on the challenging game *KeepAway* that a variant of the KBKR approach (LP2) could be successfully deployed in a reinforcement-learning setting. We also demonstrated that "batch" support-vector regression (LP1) can learn in a challenging reinforcement-learning environment without needing to have a model of the impact of actions on its environment.
2. We found that in order for the advice to be used effectively by the KBKR algorithm, we had to specify the legal ranges for *all* input features. Otherwise advice had to be either absolutely followed or "discarded" (via the slack variables of LP2) since, when the model includes any input feature not mentioned in the advice, the THEN part of advice (Eq. 3) can not be guaranteed to be met whenever the current world state matches the IF part. We also augment advice about numeric features by making explicit those constraints on the associated binary features that results from tiling the numeric features.
3. We found that it was critical that our optimization not only penalize the size of the weights in the solution, but that a sizable penalty term should also be used for the "*b*" term (of the "$y = wx + b$" solution) so that the learner does not simply predict the mean *Q* value.
4. Because little work has been done on incremental support vector machines, we chose to learn our *Q* models in a batch fashion. For a complex problem, this large set of states quickly results in more constraints than can be efficiently solved by a linear-programming system, so we had to develop a method for selecting a subset of the available information with which to train our models.
5. We found that without tile coding, we were unable to learn in *KeepAway*. One advantage of using tile coding is that we did not need to use non-linear kernels; the non-linearity of tile coding sufficed.
6. Finally, we looked at simple ways to extend the mechanism used for specifying advice in KBKR. We found it especially helpful to be able to refer to certain "dynamic" properties in the advice, such as the average *Q* value, as a method of giving advice in a natural manner.

Our future research directions include the testing of our reformulated version of KBKR on additional complex tasks, the addition of more complex features for the advice language (such as multi-step plans) and the use of additional constraints on the optimization problem (such as directly including the Bellman constraints in the optimization formulation and the ability to give advice of the form "in these world states, action *A* is better than action *B*"). We believe that the combination of support-vector techniques and advice taking is a promising approach for RL problems.

## Acknowledgements

This research was supported by DARPA IPTO grant HR0011-04-1-0007 and US Naval Research grant N00173-04-1-G026.

## References

[Andre and Russell, 2001] D. Andre and S. Russell, Programmable reinforcement learning agents, *NIPS '02.*

[Dietterich and Wang, 2001] T. Dietterich and X. Wang, Support vectors for reinforcement learning, *ECML '01*.

[Gordon and Subramanian, 1994] D. Gordon and D. Subramanian, A multistrategy learning scheme for agent knowledge acquisition, *Informatica* 17: 331-346.

[Kuhlmann et al., 2004] G. Kuhlmann, P. Stone, R. Mooney and J. Shavlik, Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer, *AAAI '04 Workshop on Supervisory Control of Learning and Adaptive Systems.*.

[Lagoudakis and Parr, 2003] M. Lagoudakis and R. Parr, Reinforcement learning as classification: Leveraging modern classifiers, *ICML '03.*

[Laud and DeJong, 2002] A. Laud and G. DeJong, Reinforcement learning and shaping: Encouraging intended behaviors, *ICML '02.*

[Lin, 1992] L-J. Lin, Self-improving reactive agents based on reinforcement learning, planning, and teaching, *Machine Learning*, 8:293-321.

[Maclin and Shavlik, 1994] R. Maclin and J. Shavlik, Incorporating advice into agents that learn from reinforcements, *AAAI '94.*

[Mangasarian et al., 2004] O. Mangasarian, J. Shavlik and E. Wild, Knowledge-based kernel approximation. *Journal of Machine Learning Research*, 5, pp. 1127-1141.

[Noda et al., 1998] I. Noda, H. Matsubara, K. Hiraki and I. Frank, Soccer server: A tool for research on multiagent systems, *Applied Artificial Intelligence* 12:233-250.

[Price and Boutilier, 1999] B. Price and C. Boutilier, Implicit imitation in multiagent reinforcement learning, *ICML '99.*

[Stone and Sutton, 2001] P. Stone and R. Sutton, Scaling reinforcement learning toward RoboCup Soccer, *ICML '01*.

[Sutton and Barto, 1998] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press.

# Writing Stratagus-playing Agents in Concurrent ALisp

**Bhaskara Marthi, Stuart Russell, David Latham**
Department of Computer Science
University of California
Berkeley, CA 94720
{bhaskara,russell,latham}@cs.berkeley.edu

## Abstract

We describe Concurrent ALisp, a language that allows the augmentation of reinforcement learning algorithms with prior knowledge about the structure of policies, and show by example how it can be used to write agents that learn to play a subdomain of the computer game Stratagus.

## 1 Introduction

Learning algorithms have great potential applicability to the problem of writing artificial agents for complex computer games [Spronck *et al.*, 2003]. In these algorithms, the agent learns how to act optimally in an environment through experience. Standard "flat" reinforcement-learning techniques learn very slowly in environments the size of modern computer games. The field of hierarchical reinforcement learning [Parr and Russell, 1997; Dietterich, 2000; Precup and Sutton, 1998; Andre and Russell, 2002] attempts to scale RL up to larger environments by incorporating prior knowledge about the structure of good policies into the algorithms.

In this paper we focus on writing agents that play the game Stratagus (stratagus.sourceforge.net). In this game, a player must control a medieval army of units and defeat opposing forces. It has high-dimensional state and action spaces, and successfully playing it requires coordinating multiple complex activities, such as gathering resources, constructing buildings, and defending one's base. We will use the following subgame of Stratagus as a running example to illustrate our approach.

**Example 1** *In this example domain, shown in Figure 1, the agent must defeat a single ogre (not visible in the figure). It starts out with a single peasant (more may be trained), and must gather resources in order to train other units. Eventually it must build a barracks, and use it to train footman units. Each footman unit is much weaker than the ogre so multiple footmen will be needed to win. The game dynamics are such that footmen do more damage when attacking as a group, rather than individually. The only evaluation measure is how long it takes to defeat the ogre.*

Despite its small size, writing a program that performs well in this domain is not completely straightforward. It is not immediately obvious, for example, how many peasants should be trained, or how many footmen should be trained before attacking the ogre. One way to go about writing an artificial agent that played this program would be to have the program contain free parameters such as **`num-peasants-to-build-given-single-enemy`**, and then figure out the optimal setting of the parameters, either "by hand" or in some automated way. A naive implementation of this approach would quickly become infeasible[1] for larger domains, however, since there would be a large number of parameters, which are coupled, and so exponentially many different joint settings would have to be tried. Also, if the game is stochastic, each parameter setting would require many samples to evaluate reliably.

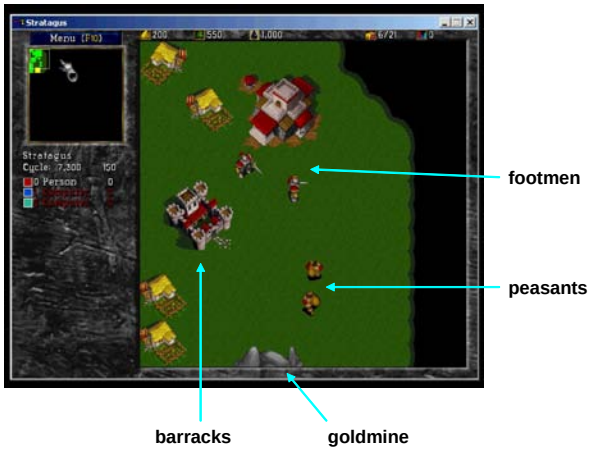


Figure 1: An example subgame of Stratagus.

The field of reinforcement learning [Kaelbling, 1996] addresses the problem of learning to act optimally in sequential decision-making problems and would therefore seem to be applicable to our situation. However, standard "flat" RL algorithms scale poorly to domains the size of Stratagus. One reason for this is that these algorithms work at the level of primitive actions such as "move peasant 3 north 1 step". These algorithms also provide no way to incorporate any prior knowledge one may have about the domain.

[1] A more sophisticated instantiation of this approach, combined with conventional HRL techniques, has been proposed recently [Ghavamzadeh and Mahadevan, 2003].

Hierarchical reinforcement learning (HRL) can be viewed as combining the strengths of the two above approaches, using *partial programs*. A partial program is like a conventional program except that it may contain *choice points*, at which there are multiple possible statements to execute next. The idea is that the human designer will provide a partial program that reflects high-level knowledge about what a good policy should look like, but leaves some decisions unspecified, such as how many peasants to build in the example. The system then learns a *completion* of the partial program that makes these choices in an optimal way in each situation. HRL techniques like MAXQ and ALisp also provide an additive decomposition of the value function of the domain based on the structure of the partial program. Often, each component in this decomposition depends on a small subset of the state variables. This can dramatically reduce the number of parameters to learn.

We found that existing HRL techniques such as ALisp were not directly applicable to Stratagus. This is because an agent playing Stratagus must control several units and buildings, which are engaged in different activities. For example, a peasant may be carrying some gold to the base, a group of footmen may be defending the base while another group attacks enemy units. The choices made in these activities are correlated, so they cannot be solved simply by having a separate ALisp program for each unit. On the other hand, a single ALisp program that controlled all the units would essentially have to implement multiple control stacks to deal with the asynchronously executing activities that the units are engaged in. Also, we would lose the additive decomposition of the value function that was present in the single-threaded case.

We addressed these problems by developing the *Concurrent ALisp* language. The rest of the paper demonstrates by example how this language can be used to write agents for Stratagus domains. A more precise description of the syntax and semantics can be found in [Marthi *et al.*, 2005].

## 2 Concurrent ALisp

Suppose we have the following prior knowledge about what a good policy for Example 1 should look like. First train some peasants. Then build a barracks using one of the peasants. Once the barracks is complete, start training footmen. Attack the enemy with groups of footmen. At all times, peasants not engaged in any other activity should gather gold.

We will now explain the syntax of concurrent ALisp with reference to a partial program that implements this prior knowledge. Readers not familiar with Lisp should still be able to follow the example. The main thing to keep in mind is that in Lisp syntax, a parenthesized expression of the form `(f arg1 arg2 arg3)` means the application of the function `f` to the given arguments. Parenthesized expressions may also be nested. In our examples, all operations that are not part of standard Lisp are in boldface.

We will refer to the set of buildings and units in a state as the *effectors* in that state. In our implementation, each effector must be given a command at each step (time is discretized into one step per 50 cycles of game time). The command may be a no-op. A concurrent ALisp program can be multi-

```
(defun top ()
  (spawn ''allocate-peasants''
    #'peas-top nil *peas-eff*)
  (spawn ''train-peasants'' #'townhall-top
    *townhall* *townhall-eff*)
  (spawn ''allocate-gold''
  #'alloc-gold nil)
  (spawn ''train-footmen'' #'barracks-top nil)
  (spawn ''tactical-decision''
    #'tactical nil))
```

Figure 2: Top-level function

```
(defun peas-top ()
  (loop
    unless (null (my-effectors))
    do (let ((peas (first (my-effectors))))
        (choose ''peas-choice''
          (spawn (list ''gold'' peas)
            #'gather-gold nil peas)
          (spawn (list ''build'' peas)
            #'build-barracks nil peas)))))
```

Figure 3: Peasant top-level function

threaded, and at any point, each effector is assigned to some thread.

Execution begins with a single thread, at the function `top`, shown in Figure 2. In our case, this thread simply creates some other threads using the **`spawn`** operation. For example, the second line of the function creates a new thread with ID "allocate peasants", which begins by calling the function `peas-top` and is assigned effector `*peas-eff*`.

Next, examine the `peas-top` function shown in Figure 3. This function loops until it has at least one peasant assigned to it. This is checked using the **`my-effectors`** operation. It then must make a choice about whether to use this peasant to gather gold or to build the barracks, which is done using the **`choose`** statement. The agent must learn how to make such a choice as a function of the environment and program state. For example, it might be better to gather gold if we have no gold, but better to build the barracks if we have plentiful gold reserves.

Figure 4 shows the `gather-gold` function and the `navigate` function, which it calls. The navigate function navigates to a location by repeatedly choosing a direction to move in, and then performing the move action in the environment using the **`action`** operation. At each step, it checks to see if it has reached its destination using the **`get-env-state`** operation.

We will not give the entire partial program here, but Figure 5 summarizes the threads and their interactions. The `allocate-gold` thread makes decisions about whether the next unit to be trained is a footman or peasant, and then communicates its decision to the `train-footmen` and `train-peasants` threads using shared variables. The

```
(defun gather-gold ()
  (call navigate *gold-loc*)
  (action *get-gold*)
  (call navigate *base-loc*)
  (call *dropoff*))

(defun navigate (loc)
  (loop
    with peas = (first (my-effectors))
    for s = (get-env-state)
    for current = (peas-loc peas s)
    until (equal current loc)
    do (action ''nav''
        (choose *N* *S* *W* *E*))))
```

Figure 4: Gather-gold and navigate functions

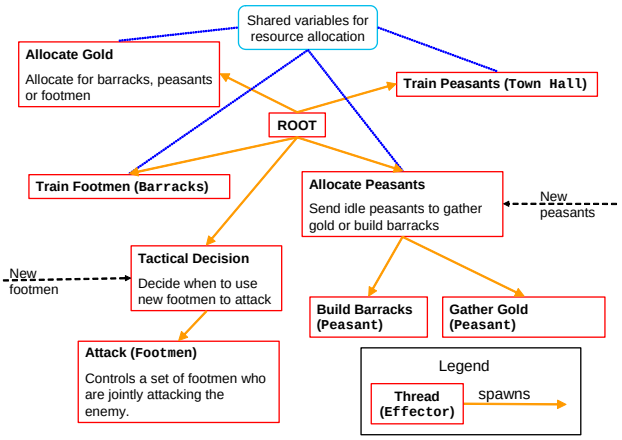


Figure 5: Structure of the partial program for the example domain

`tactical-decision` thread is where new footman units are assigned. At each step it chooses whether to launch an attack or wait. The attack is launched by spawning off a new thread with the footmen in the tactical thread. Currently, it just consists of moving to the enemy and attacking, but more sophisticated tactical manouvering could be incorporated as prior knowledge, or learnt by having choices within this thread.

## 3 Semantics

We now give an informal semantics of what it means to execute a partial program. We view each state of the environment as having a set of effectors, such that the set of actions allowed at that state is the set of assignments of individual actions to each effector. Thus, we have to make sure that the **`action`** statements in all the threads execute simultaneously. Also, we would like **`choose`** statements to execute simultaneously as much as possible. This is based on the intuition that it is easier to represent and learn the value function for a single joint choice than for a set of sequential choices, each depending on the previous ones. Finally, no time is assumed to elapse in the environment except when the **`action`** statements are being executed, and each joint action takes exactly one time step. Section 6 describes how to fit Stratagus into this framework.

We build on the standard semantics for interleaved execution of multithreaded programs. At each point, there is a set of threads, each having a call stack and a program counter. There is also a set of global shared variables. All this information together is known as the *machine state* $\theta$. We also refer to the *joint state* $\omega = (s, \theta)$ where $s$ is the environment state. A thread is said to be an *action thread* in a given joint state if it is at an **`action`** statement, a *choice thread* if it at a **`choice`** statement, and a *running thread* otherwise.

Given a particular joint state $\omega$, there are three cases for what happens next. If every thread with effectors assigned to it is an action thread, then we are at an *joint action state*. The joint action is done in the environment, and the program counters for the action threads are incremented. If every thread is either an action thread or a choice thread, but we are not at an action state, then we are at a *joint choice state*. The agent must simultaneously make a choice for all the choice threads, and their program counters are updated accordingly. If neither of these two cases holds, then some external scheduling mechanism is used to pick a thread from the running threads whose next statement is then executed.

It can be shown that a partial program together with a Markovian environment yields a *semi-Markov Decision Process* (SMDP), whose state space consists of the joint choice states. The set of "actions" possible at a joint state corresponds to the set of available joint choices, and the reward function of making choice $u$ in $\omega$ is the expected reward gained in the environment until the next choice state $\omega'$.

The learning task is then to learn the Q-function of this SMDP, where $Q(\omega, u)$ is the expected total future reward if we make choice $u$ in $\omega$ and act optimally thereafter. Once a Q-function is learnt, at runtime the agent simply executes the partial program, and when it reaches a choice state $\omega$, picks the choice $u$ maximizing $Q(\omega, u)$. We will discuss how to do this efficiently in Section 4.

## 4 Approximating the Q-Function

It is infeasible to represent the function $Q(\omega, u)$ exactly in large domains for two reasons. First, the joint state space is huge, resulting in too many parameters to represent or learn. Second, in situations with many effectors, the set of joint choices, exponential in the number of effectors, will be too large to directly maximize over during execution.

A solution to both these problems is provided by approximating the Q-function as a linear combination of features :

$$Q(\omega, u) = \sum_{k=1}^{K} w_k f_k(\omega, u)$$

We can control the number of parameters to learn by setting $K$ appropriately. Also, if features are chosen to be "local", i.e. to each depend on a small subset of the choice threads, then the maximization can, in many cases, be performed efficiently [Guestrin *et al.*, 2002] using nonsequential dynamic programming. Some example features for the Stratagus subgame are :

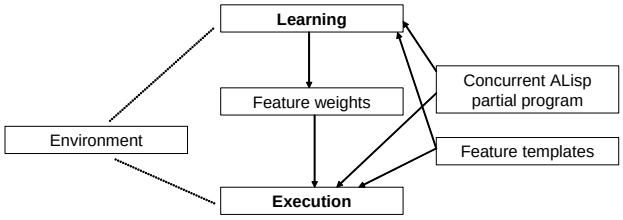


Figure 6: Architecture of the system

1. A feature $f_{\text{gold}}$ that counts how much gold we have in state $\omega$
2. A feature $f_{\text{attack}}$ that is 1 if there are at least 3 footmen in the tactical thread and $u$ involves choosing to start an attack, and 0 otherwise.
3. A feature $f_{\text{dist, 3}}$ that returns the distance of peasant 3 to his destination after making the navigation choice in $u$.

Thus, features may depend on the environment state, thread states, or memory state of the partial program. They may also depend on the choices made by any subset of the currently choosing threads.

To further reduce the number of parameters, we make use of relational feature templates [Guestrin *et al.*, 2003]. The feature $f_{\text{dist, 3}}$ above refers to a specific peasant, but it seems reasonable to expect that the weight of this feature should be the same regardless of the identity of the peasant. To achieve this, we allow the specification of a single "feature template" $f_{\text{dist}}$ that results in a distance feature for each peasant in a given state, all sharing the same weight $w_{\text{dist}}$.

In our implementation, the set of feature templates must be specified by the user (as Lisp functions). The performance of the learning algorithms can be strongly dependent on the specific features used, and in practice feature engineering takes much more time than writing the partial program.

## 5 Learning

Figure 6 summarizes the overall system architecture. The user provides a partial program and features to the learning algorithm, which uses experience in the environment to learn a set of weights. The learnt weights can then be used, in conjunction with the partial program and features, to act in the environment.

Since the learning task is to learn the Q-function of an SMDP, we can adapt the standard SMDP Q-learning algorithm to our case. The algorithm assumes a stream of samples of the form $(\omega, u, r, \omega')$. These can be generated by executing the partial program in the environment and making joint choices randomly, or according to some exploration policy. After observing a sample, the algorithm performs the online update

$$\vec{w} \leftarrow \vec{w} + \alpha \left( r + \max_{u'} Q(\omega', u'; \vec{w}) - Q(\omega, u; \vec{w}) \right) \vec{f}(\omega, u)$$

where $\alpha$ is a learning-rate parameter that decays to 0 over time.

The above algorithm does not make explicit use of the procedural or thread structure of the partial program - learning is centralized and acts on the entire joint state and joint choice. In recent work, we have developed an improved algorithm in which the learning is done separately for each thread. The algorithm also makes use of the procedural structure of the partial program within each thread. To achieve this, the system designer needs to specify in advance a *reward decomposition* function that takes the reward at each timestep and divides it among the threads.

## 6 Experiments

We implemented the Concurrent ALisp language and the above algorithms on top of standard Lisp. We interfaced with the Stratagus game using a socket. Time was discretized so that every 50 steps of game time (typically about a quarter of a second) corresponds to one "timestep" in the environment. For simplicity, we made the game static, i.e., it pauses at each step and waits for input from the ALisp program, but the algorithms ran fast enough that it should be possible to make the environment dynamic. To fit Stratagus into our framework, in which a joint action at a state must assign an individual action to all the effectors, we added a `noop` action that can be assigned to any unit for which there is no specific command on a given timestep.

We ran the original learning algorithm on the domain from Example 1. Videos of the policies over the course of learning can be found on the web[2]. The initial policy trains no new peasants, and thus collects gold very slowly. It also attacks immediately after each footman is trained. In contrast, the final policy, learnt after about 15000 steps of learning, trains multiple peasants to ensure a constant supply of gold, and attacks with groups of footmen. Thanks to these improvements, the time to defeat the enemy is reduced by about half.

## 7 Scaling up

It is reasonable to ask how relevant the above results are to the overall problem of writing agents for the full Stratagus game, since the example domain is much smaller. In particular, the full game has many more state variables and effectors, and longer episodes which will require more complex policies. These will increase the complexity of each step of learning and execution, the amount of sampled experience needed to learn a good policy, and the amount of input needed from the human programmer.

We first address the complexity of each step of the algorithms. Since we are using function approximation, the complexity of our algorithms doesn't depend directly on the number of joint states, but only on on the number of features. We believe that it should be possible to find good feature sets that are not so large as to be bottlenecks, and are working to verify this empirically by handling increasingly large subdomains of Stratagus.

The larger number of effectors will typically result in more threads, which will increase the complexity of each joint choice. As discussed in Section 4, a brute-force algorithm for making joint choices would scale exponentially with the number of threads, but our algorithm grows exponentially in

[2] http://www.cs.berkeley.edu/∼bhaskara/ijcai05-videos

the tree-width of the coordination graph and only linearly with the number of threads. By making each feature depend only on a small "local" subset of the choosing threads, we can usually make the treewidth small as well. Occasionally, the treewidth will still end up being too large. In this case, we can use an approximate algorithm to find a reasonably good joint choice rather than the best one. Our current implementation selectively removes edges from the coordination graph. Methods based on local search in the space of joint choices are another possibility. Once again, this is unlikely to be the major bottleneck when scaling up.

The cost of executing the partial program itself is currently negligible, but as partial programs become more complex and start performing involved computations (e.g. path-planning), this may change. It should be noted that this issue comes up for any approach to writing controllers for games, whether or not they are based on reinforcement learning. One intriguing possibility in the HRL approach is to treat this as a *meta-level control problem* [Russell and Wefald, 1991] in which the amount of computation is itself decided using a choice statement. For example, an agent may learn that in situations with few units and no immediate combat, it is worth spending time to plan efficient paths, but when there is a large battle going on, it's better to quickly find a path using a crude heuristic and instead spend computation on the joint choices for units in the battle.

The amount of experience needed to learn a good policy is likely to be more of a concern as the domains get increasingly complex and the episodes become longer. The number of samples needed will usually increase at least polynomially with the episode length. This can be mitigated by the use of reward shaping [Ng *et al.*, 1999]. Note also that concurrent ALisp is not wedded to any one particular learning algorithm. For example, we have extended *least-squares policy iteration* [Lagoudakis and Parr, 2001], which aims to make better use of the samples, to our situation. Algorithms that learn a model of the environment along with the Q-function [Moore and Atkeson, 1993] are another promising area for future work.

Finally, the amount of human input needed in writing the partial program, features, reward decomposition, and shaping function will increase in more complex domains. A useful direction to pursue in the medium-term is to learn some of these instead. For example, it should be possible to add a feature selection procedure on top of the current learning algorithm.

## 8 Conclusion

We have outlined an approach to writing programs that play games like Stratagus using partial programming with concurrent ALisp, and demonstrated its effectiveness on a subdomain that would be difficult for conventional reinforcement learning methods. In the near future, we plan to implement our improved learning algorithm, and scale up to increasingly larger subgames within Stratagus.

## References


[Andre and Russell, 2002] D. Andre and S. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI*, 2002.

[Dietterich, 2000] T. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *JAIR*, 13:227–303, 2000.

[Ghavamzadeh and Mahadevan, 2003] M. Ghavamzadeh and S. Mahadevan. Hierarchical policy-gradient algorithms. In *Proceedings of ICML 2003*, pages 226–233, 2003.

[Guestrin *et al.*, 2002] C. Guestrin, M. Lagoudakis, and R. Parr. Coordinated reinforcement learning. In *ICML*, 2002.

[Guestrin *et al.*, 2003] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational mdps. In *IJCAI*, 2003.

[Kaelbling, 1996] L. Kaelbling. Reinforcement learning : A survey. *Journal of Artificial Intelligence Research*, 1996.

[Lagoudakis and Parr, 2001] M. Lagoudakis and R. Parr. Model-free least squares policy iteration. In *Advances in Neural Information Processing Systems*, 2001.

[Marthi *et al.*, 2005] B. Marthi, S. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In *Proceedings of IJCAI 2005*, 2005. to appear.

[Moore and Atkeson, 1993] Andrew Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, October 1993.

[Ng *et al.*, 1999] A. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations : theory and application to reward shaping. In *ICML*, 1999.

[Parr and Russell, 1997] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 9*, 1997.

[Precup and Sutton, 1998] D. Precup and R. Sutton. Multi-time models for temporally abstract planning. In *Advances in Neural Information Processing Systems 10*, 1998.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. *Do the right thing: studies in limited rationality*. MIT Press, Cambridge, MA, USA, 1991.

[Spronck *et al.*, 2003] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. Online adaption of game opponent ai in simulation and in practice. In *Proceedings of the Fourth International Conference on Intelligent Games and Simulation*, 2003.

# Defeating Novel Opponents in a Real-Time Strategy Game

**Matthew Molineaux[1], David W. Aha[2], and Marc Ponsen[3]**
[1]ITT Industries; AES Division; Alexandria, VA 22303
[2]Navy Center for Applied Research in Artificial Intelligence;
Naval Research Laboratory (Code 5515); Washington, DC 20375
[3]Department of Computer Science and Engineering;
Lehigh University; Bethlehem, PA 18015
[1,2]*first.last*@nrl.navy.mil [3]mjp304@lehigh.edu

## Abstract

The *Case-Based Tactician (CAT)* system, created by Aha, Molineaux, and Ponsen (2005), uses case-based reasoning to learn to win the real-time strategy game *Wargus*. Previous work has shown CAT's ability to defeat a randomly selected opponent from a set against which it has trained. We now focus on the task of defeating a selected opponent while training on others. We describe CAT's algorithm and report its cross-validation performance against a set of Wargus opponents.

## 1 Introduction

Research on artificial intelligence (AI) and games has an extraordinary history that dates from 1950. Automated game-playing programs now exist that outperform world champions in classic board games such as checkers, Othello, and Scrabble (Schaeffer, 2001). These efforts brought about significant advancements in search algorithms, machine learning techniques, and computer hardware.

In recent years, AI researchers (e.g., Laird & van Lent, 2001; Buro, 2003) have begun focusing on complex strategy simulation games that offer new challenges, including but not limited to partially observable environments, asynchronous gameplay, comparatively large decision spaces with varying abstraction levels, and the need for resource management processes.

Learning to win such games may require more sophisticated representations and reasoning capabilities than games with smaller decision spaces. To assist their studies, Ponsen and Spronck (2004) developed a lattice for representing and relating abstract states in *Wargus*, a moderately complex real-time strategy game which mimics the popular commercial game *WarCraft II*. They also sharply reduced the decision space by employing a high-level language for game agent actions. Together, these constrain the search space of useful plans (i.e., *strategies*) and state-specific subplans (i.e., *tactics*). This approach allowed them to focus on the game at a strategic level, abstracting away the minutiae of individual actions. They developed a genetic algorithm and a technique called *dynamic scripting* to learn plans spanning the entire game, which win against a fixed opponent. We build on this success and approach the same challenge, without foreknowledge of the adversary.

We have developed a case-based approach for playing *Wargus* that learns to select appropriate tactics based on the current state of the game. This approach is implemented in the *Case-based Tactician (CAT)* system. We will report learning curves that demonstrate that its performance improves against one opponent from training against others. CAT is the first case-based system designed to defeat an opponent that uses tactics and strategies that it has not trained against.

In Section 2, we review some case-based approaches in games research. In Section 3, we describe the domain knowledge available to CAT and how this reduces the complexity of finding effective strategies for defeating *Wargus* opponents. CAT's case-based approach is examined in section 4. We review our empirical methodology and CAT's results in Section 5, and close with a discussion in Section 6 that highlights several future research objectives.

## 2 Background: Case-based Reasoning Research in Large Decision-Space Games

Below, we contrast some of the prior Case-Based Reasoning research focusing on large decision-space games with CAT.

ROBOCUP SOCCER is a popular CBR focus. Wendler and Lenz (1998) described an approach for identifying where simulated agents should move, while Wendler et al. (2001) reported strategies for learning to pass. Gabel and Veloso (2001) instead used a CBR approach to select members for a team. Karol et al. (2003) proposed a case-based action selection algorithm for the 4-legged league. While these real-time environments are challenging strategically, they do not involve complicating dimensions common to strategy games, such as economies, research, and warfare.

Some researchers have addressed real-time individual games. Goodman (1994) applied a projective visualization approach for *Bilestoad*, a personal combat game, to predict actions that would inflict damage on the adversary and/or minimize damage to oneself. Fagan and Cunningham (2003) instead focused on a plan recognition task; they acquire cases (state-action planning sequences) for predicting the next action of a human playing *Space Invaders*™. In

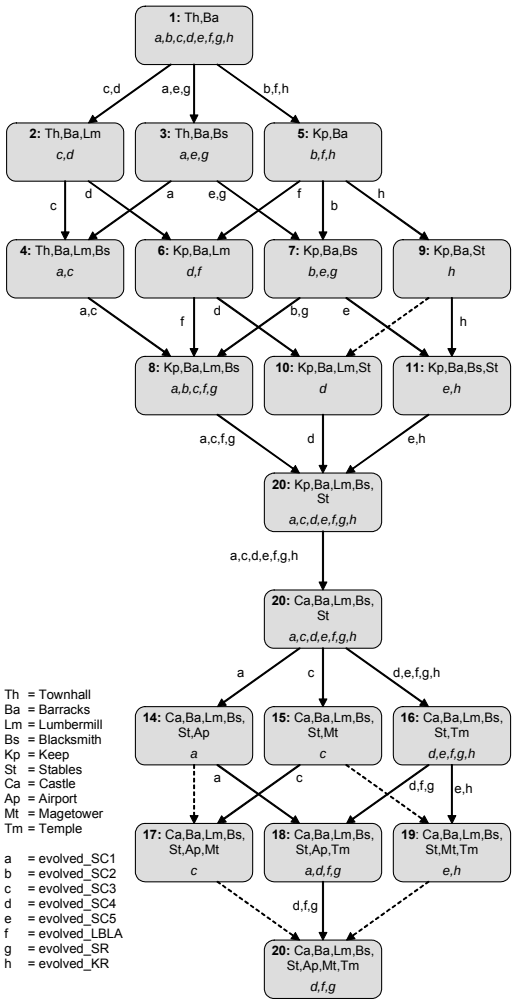


**Figure 1**: A building-specific state lattice for WARGUS, where nodes represent states (defined by a set of completed buildings), and state transitions involve constructing a specific building. Also displayed are the evolved counter-strategies (a-h) that pass through each state.

contrast, CAT does not perform projective modeling, and does not learn to recognize adversarial plans. Instead, it acquires cases concerning the application of a subplan in a given state, learns to select subplans for a given state, and executes them in a more complex gaming environment.

Fasciano's (1996) MAYOR learns from planning failures in SIMCITY™, a real-time city management game with no traditional adversaries. MAYOR monitors planning expectations and employs a causal model to learn how to prevent failure repetitions, where the goal is to improve the ratio of successful plan executions. In contrast, CAT does not employ plan monitoring or causal goal models, and does not adapt retrieved plans. Rather, it simply selects, at each state, a good tactic (i.e., subplan) to retrieve. Also, our gaming environment includes explicit adversaries.

Ulam et al. (2004) described a meta-cognitive approach that performs failure-driven plan adaptation for *Freeciv*, a complex turn-based strategy game. While they employed substantial domain knowledge in the form of task models, it was only enough to address a simple sub-task (defending a city). In contrast, CAT performs no adaptation during reuse, but does perform case acquisition. Also, CAT focuses on winning a game rather than on performing a subtask.

Guestrin et al. (2003) applied relational Markov decision process models for some limited *Wargus* scenarios (e.g., 3x3 combat). They did not address more complex scenarios because their planner's complexity grows exponentially with the number of units. Similarly, Cheng and Thawonmas (2004) proposed a case-based plan recognition approach for assisting *Wargus* players, but only for low-level management tasks. Their state representation is comprehensive and incorporates multiple abstraction levels.

## 3 Domain Knowledge in CAT

CAT employs three sources of domain knowledge to reduce the complexity of WARGUS. Two of these are from Ponsen and Spronck (2004): *a building state lattice*, which abstracts the state space, and *a set of tactics for each state*.

In *Wargus*, certain buildings provide the player with new capabilities, in the form of newly available technologies, units and buildings. Therefore, it makes sense to break the game up into periods when a certain set of buildings exist. We call the time between the construction of one such building to the time the next is built a *building state*. The building state defines the set of actions available to the player at any one time. It is important not to confuse this with the *game state*, which encompasses all of the variables of the game, and changes much more frequently, whether the player takes action or not.

The building state lattice (Figure 1) shows the transitions that are possible from one building state to the next. This was developed in the course of Ponsen and Spronck's research for the purpose of planning. In their research, plans spanned an entire game, which we call *strategies*. These strategies are made up of *tactics*, which are subplans that span a single building state. The tactics are made up of individual actions; using the state lattice, Ponsen and Spronck ensured that all actions used were legal for the corresponding building state, and that the entire plan was therefore legal.

The second source of domain knowledge CAT has access to (i.e., a set of tactics for each state) was acquired using Ponsen and Spronck's genetic algorithm (2004). This was used to evolve chromosomes, representing counter-strategies, against specific opponents. Eight opponent strategies (see Table 1) were available, including some that are publicly available and others that were manually developed. The resulting counter-strategies (i.e., one for each of the eight opponents) were used as a source for automatically acquiring the tactics (Ponsen et al. 2005a) used by CAT. The names of counter-strategies are shown in the lower left of Figure 1. By making reference to the state lattice, CAT is able to determine what tactics are applicable whenever a new building state is entered and choose among them. Figure 1 shows for each building state which of the

**Table 1**: WARGUS opponents used in the experiments.

| Opponent | Description |
|---|---|
| LBLA | This balances offensive actions, defensive actions, and research. |
| SR | Soldier's Rush: This attempts to overwhelm the opponent with cheap offensive units in an early state of the game. |
| KR | Knight's Rush: This attempts to quickly advance technologically, launching large offences as soon as strong units are available. |
| SC1-SC5 | The top 5 scripts created by students, based on a class tournament. |

tactics are applicable to each state. For example, State 1 offers tactics from all evolved counter-strategies, only three tactics apply to state 20, namely those derived from the evolved_SC4, evolved_LBLA and evolved_SR counter-strategies.

In order to choose which tactic to use, CAT employs a *casebase* for each building state which records certain features of the game state, a value indicating which tactic was selected, and an associated performance value (see section 4.3) achieved by using that tactic in gameplay. We will discuss our case-based strategy selection algorithm in more detail in the next Section.

# 4 Case-Based Strategy Selection

Our case-based approach for selecting which tactic to use in each state employs the state lattice and state-specific tactics libraries described in Section 3. By doing this, the decision space (i.e., the number of tactics per state) becomes small, and an attribute-value representation of game situations suffices to select tactics. We define a case *C* as a tuple of four objects:

*C* = <BuildingState, Description, Tactic, Performance>

where *Building State* is an integer node index in the building state lattice, *Description* is a set of features of the current situation (see Table 2), *Tactic* is a counter-strategy's sequence of actions for that building state, and *Performance* is a real value in the range [0,1] that reflects the utility of choosing that tactic for that building state, where higher values indicate higher performance (see Section 4.3). We next use Aamodt and Plaza's (1994) task decomposition model to detail our approach.

**Table 2:** Features of the case description.

| Feature | Description |
|---|---|
| $\Delta Kills_{i-1}$ | Number of opponent combat & worker units killed minus the same for oneself, in the preceding state |
| $\Delta Razings_{i-1}$ | Number of opponent buildings destroyed minus same for onself, in the preceding state |
| $Buildings_{o}$ | Number of opponent buildings ever created |
| $CombatUnits_{o}$ | Number of opponent combat units ever created |
| $Workers_{o}$ | Number of opponent worker units ever created |
| $Buildings_{p,i}$ | Number of own buildings currently existing |
| $CombatUnits_{p,i}$ | Number of own combat units currently existing |
| $Workers_{p,i}$ | Number of own worker units currently existing |

## 4.1 Retrieval

CAT retrieves cases when a new state in the lattice is entered (i.e., at the game's start, and when a transition building is built). At those times, it requests and records values for the eight features shown in Table 2. This set balances information on recent game changes (i.e., the first two features), the opponent's situation (e.g., $Workers_{o}$), and the player's situation (e.g., $Workers_{p,i}$). When games begin, the value of the first two features is 0, while the others have small values. About 50 units are created, per side, in a short game, and a player's limit is 200 at any time.

Cases are grouped by the building state, and at most one case is recorded per building state per game. In our experiments, games lasted an average of 5 minutes, and CAT made a maximum of 8 decisions in that time (see Figure 1); therefore, CAT does not require a fast indexing strategy.

The similarity between a stored case *C* and the current game state *S* is defined as:

$$Sim_{C,S} = (C_{Performance}/dist(C_{Description}, S)) - dist(C_{Description}, S)$$

where dist() is the (unweighted, unnormalized) Euclidean distance between two cases for the eight features. We chose this simple function because it emphasizes distance, and it prefers the higher-performing of two equally distant cases. It is particularly useful for building state 1 (i.e., the start of the game), when all case descriptions are all identical.

CAT uses a modified k-nearest neighbor function to select case Tactics for retrieval. Among the k most similar cases, it retrieves one with the highest Performance. However, to gain experience with all tactics in a state, case retrieval is not performed until each available Tactic at that state is selected *e* times, where *e* is CAT's *exploration* parameter. During exploration, one of the least frequently used tactics is retrieved for reuse. Exploration also takes place whenever the highest performance among the k-nearest neighbors is below 0.5. In our experiments, *e* and *k* were both set to 3; we have not attempted to tune these parameters.

## 4.2 Reuse

CAT's reuse stage is given a retrieved tactic. While adaptation takes place, it is not controlled by CAT, but is instead performed at the level of the action primitives in the context of the *Wargus* game engine (e.g., if an action requests the creation of a building, the game engine decides its location and which workers will construct it, which can differ in each game situation).

## 4.3 Revision

Revision involves executing the reused tactic in *Wargus*, and evaluating the results. No repairs are made to these tactics; they are treated as black boxes.

Evaluation yields the performance of a case's tactic, which is a function of the increase in the player's *Wargus* game score relative to the opponent. The score is measured both when the tactic is selected and at the game's end,

which occurs when one player eliminates all of the opponent's units, or when we terminate a game if no winner has emerged after ten minutes of clock time.

The performance for the tactic of case *C* for building state *b* is computed as:

$$\text{Won}_i = \begin{cases} 1, if\_CaT\_wins \\ 0, if\_CaT\_loses \end{cases}$$

$$C_{Performance} = \sum_{i=1,n} C_{Performance,i} / n$$

$$C_{Performance,i} = 1/3(\Delta\text{Score}_i + \Delta\text{Score}_{i,b} + \text{Won}_i)$$

$$\Delta\text{Score}_i = (\text{Score}_{i,p} - \text{Score}_{i,p,b}) / ((\text{Score}_{i,p} - \text{Score}_{i,p,b}) + (\text{Score}_{i,o} - \text{Score}_{i,o,b}))$$

$$\Delta\text{Score}_{i,b} = (\text{Score}_{i,p,b+1} - \text{Score}_{i,p,b}) / ((\text{Score}_{i,p,b+1} - \text{Score}_{i,p,b}) + (\text{Score}_{i,o,b+1} - \text{Score}_{i,o,b}))$$

$$Sim(C,S) = \frac{C_{\text{Performance}}}{\text{dist}(C_{\text{Description}}, S)} - \text{dist}(C_{\text{Description}}, S) \quad (1)$$

where *n* is the number of games in which *C* was selected, $\text{Score}_{i,p}$ is the player's *Wargus* score at the end of the $i^{th}$ game in which *C* is used, $\text{Score}_{i,p,b}$ is player *p*'s score before *C*'s tactic is executed in game *i*, and $\text{Score}_{i,p,b+1}$ is *p*'s score after *C*'s tactic executes (and the next state begins). Similarly, $\text{Score}_{i,o}$ is the opponent's score at the end of the $i^{th}$ game in which *C* is used, etc. Thus, *C*'s performance is updated after each game in which it is used, and equal weight is given to how well the player performs during its state and throughout the rest of the game.

### 4.4 Retention

During a game, CAT records a description when it enters each building state, along with the score and tactic selected. It also records the scores of each side when the game ends, along with who won (neither player wins a tie). For each building state traversed, CAT checks to see whether a case *C* exists with the same <Description, Tactic> pair. If so, it updates *C*'s value for Performance. Otherwise, CAT creates a new case *C* for that BuildingState, Description, Tactic, and Performance as computed in Section 4.3 (this counts as *C*'s first application). Thus, while duplicate cases are not created, CAT liberally creates new ones, and does not employ any case deletion policy.

## 5 Evaluation and Analysis

Our evaluation focuses on examining the hypothesis that CAT's method for selecting tactics outperforms the best of the eight counter-strategies.

### 5.1 Competitors: WARGUS Players

CAT's performance is compared against the best performing counter-strategies. We would like CAT to outperform the best counter-strategies in terms of achieved score and frequency of victory against the opponents listed in Table 1. CAT has the advantage over the static evolved strategies, in that it can adapt its strategy to the opponents based on the three sources of domain knowledge described in Section 3. However, the best evolved counter-strategies achieve high results against the opponents and are though to outperform.

### 5.2 TIELT Integration

TIELT (Aha & Molineaux, 2004) is a freely available tool (http://nrlsat.ittid.com) that facilitates the integration of decision systems and simulators. We used this tool to perform the experiments described. The Game Model and Game Interface Model came from the Wargus integration led by Lehigh University (Ponsen et al., 2005b). We developed an agent to isolate the decision tasks to be performed and interpret the decisions of the Decision System; we also developed a Decision System Interface Model which describes communications between the CAT system and TIELT. Finally, the experiment methodology describes the data to be collected and the procedure for running the experiment described in the next section.

### 5.3 Empirical Methodology

We compared CAT to the evolved counter-strategies for its ability to win *Wargus* games. We used a fixed initial scenario, on a 128x128 tile map, involving a single opponent. The opponent used one of the eight strategies listed in Table 1. With the exception of the opponent strategies designed by students, these were all used in (Ponsen and Spronck, 2004).

We chose the percentage of the total score as defined by *Wargus*, and the frequency of wins against a test opponent as our dependent variables. Because the environment is non-deterministic (due to communications latencies), we averaged the difference in score over ten trials, and measured the percentage of wins over the same period.

We performed an eight-fold cross validation study, training CAT against seven of the available opponents, and measuring performance against the eighth. For each training trial, the opponent was randomly selected from among the seven training opponents; after every 25 training trials, CAT was tested against the eighth opponent for ten trials.

### 5.4 Results

Figure 2 compares CAT's frequency of victory to the average and best performance of the eight counter-strategies. On average, the eight counter-strategies win 34% of the time. Evolved_SC5 wins 72% of the time; this is the best frequency among the counter-strategies. After 100 trials, CAT wins 79% of the time. We compared the frequency of victory for CAT after 100 games against each opponent with Evolved_SC5's performance against the same opponents, using a paired two sample one-tail t-test for means; the results were not statistically significant. We also compared the average score percentage against each opponent using the same test; the difference for this comparison was significant at the .05 level.

However, Evolved_SC5 was not the best performer in terms of score; Figure 3 compares CAT against Evolved_SR, the best *scoring* of the eight counter-strategies. CAT achieved 65% of the score on average, whereas Evolved_SR achieved 64%. This difference was not shown to be significant. However, Evolved_SR won

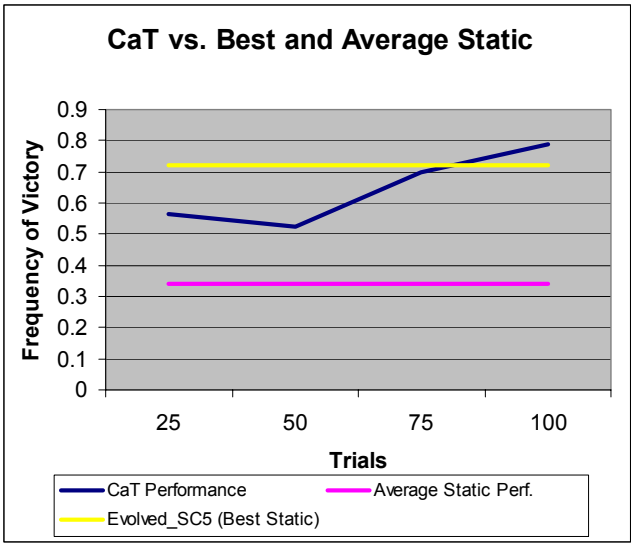


**Figure 2**: Comparison of frequency of CaT victory versus best and average of the counter-strategies.

only 49% of the time (it was the 3rd best on this metric). Here, CAT outperforms Evolved_SR at the .01 level.

This is the first attempt that we know of to learn to defeat an unknown opponent at a real-time strategy game using case-based reasoning. We were not able to significantly outperform all of the static counter-strategies, but we were able to perform at least as well as the best counter-strategies on two separate metrics, which no single counter-strategy could match.

## 6 Discussion

In Section 5, we showed that case-based techniques can be used to learn how to defeat an unknown opponent. The

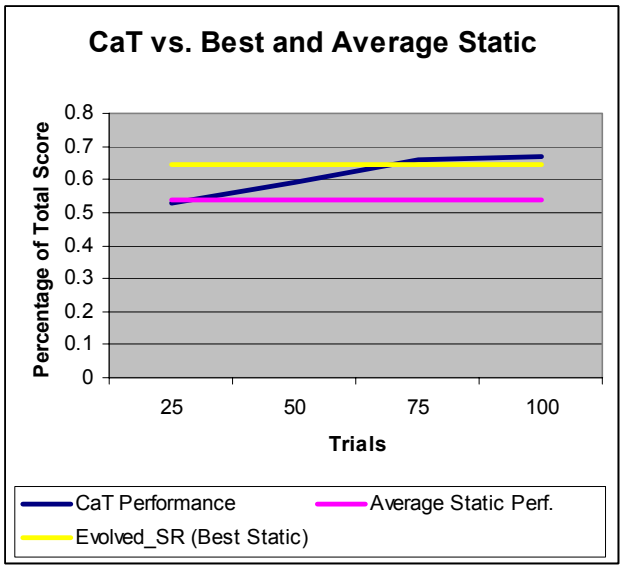


**Figure 3**: Comparison of CaT score versus best and average of the counter-strategies.

static tactics developed using Ponsen and Spronck's genetic algorithm, although evolved to defeat a single opponent in a deterministic setting, were also somewhat flexible in the non-deterministic environment introduced by controlling WARGUS externally. Although they no longer enjoyed a 100% success rate against the opponent that each was evolved to defeat, some of their success rates were high in our experiments. CAT was able to combine the successes of individual counter-strategies, showing that case-based selection of tactics is a viable strategy.

More study could improve CAT's performance in this experiment. One flaw in our methodology is that the various features of the case description are not normalized, giving some features a larger effect on the distance calculation. This should be investigated, and there may be a possibility of improvement.

No attempt has been made to calculate the optimality of the opponents or variance between them. The opponents we used may not cover the space of possible opponents well; some are certainly harder than others, and may require qualitatively different strategies to defeat. This is supported by the fact that some strategies are much more difficult for the CAT system to defeat without training on them directly. More opponents, that better cover the space of possible strategies, may provide better approximations for CAT's learning rate on the entire strategy population by reducing the variance within the opponent pool.

The problem of tactic selection can be viewed as a reinforcement learning problem, and several reinforcement learning approaches have been used with tasks in the game of Stratagus (Guestrin *et al*., 2003; Marthi *et al.*, 2005). Some similarities exist between CAT's selection algorithm and a reinforcement learning approach, in that a performance value is used to guide future actions in similar states. However, CAT's approach disregards optimized long-term rewards, while focusing more heavily on degrees of similarity than traditional reinforcement learning approaches.

## 7 Conclusion

This is the first attempt to use a case-based reasoning system to win real-time strategy (RTS) games against an unknown opponent. Our experiments show that CAT learns to perform as well as or better than the best performing counter-strategy on score and victory frequency metrics.

The TIELT system was able to adapt an earlier experiment (Aha, Molineaux, and Ponsen 2005) quite easily to perform the new experiment we report in this paper. Ongoing research on this system is expected to benefit future work along these lines. Furthermore, our experience with the new experiment highlights key areas for expansion of TIELT's capabilities.

CAT's algorithm has not been tailored for this application; its performance can probably be further improved. Also, many interesting research issues require further attention, such as CAT's applicability to online learning tasks, and transferring learned knowledge to win other games.

**Acknowledgements**

This research was supported by DARPA's Information Processing Technology Office and the Naval Research Laboratory.

## References

Aamodt, A., & Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, **7**, 39-59.

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

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.

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.

Cheng, D.C., & Thawonmas, R. (2004). Case-based plan recognition for real-time strategy games. *Proceedings of the Fifth Game-On International Conference* (pp. 36-40). Reading, UK: Unknown publisher.

Fagan, M., & Cunningham, P. (2003). Case-based plan recognition in computer games. *Proceedings of the Fifth International Conference on Case-Based Reasoning* (pp. 161-170). Trondheim, Norway: Springer.

Fasciano, M.J. (1996). *Everyday-world plan use* (Technical Report TR-96-07). Chicago, Illinois: The University of Chicago, Computer Science Department.

Forbus, K., Mahoney, J., & Dill, K. (2001). How qualitative spatial reasoning can improve strategy game AIs. In J. Laird & M. van Lent (Eds.) *Artificial Intelligence and Interactive Entertainment: Papers from the AAAI Spring Symposium* (Technical Report SS-01-02). Stanford, CA: AAAI Press.

Gabel, T., & Veloso, M. (2001). *Selecting heterogeneous team players by case-based reasoning: A case study in robotic soccer simulation* (Technical Report CMU-CS-01-165). Pittsburgh, PA: Carnegie Mellon University, School of Computer Science.

Goodman, M. (1994). Results on controlling action with projective visualization. *Proceedings of the Twelfth National Conference on AI* (pp. 1245-1250). Seattle, WA: AAAI Press.

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

Karol, A., Nebel, B., Stanton, C., & Williams, M.-A. (2003). Case based game play in the RoboCup four-legged league: Part I the theoretical model. In D. Polani, B. Browning, A. Bonarini, & K. Yoshida (Eds.) *RoboCup 2003: Robot Soccer World Cup VII* (pp. 739-747). Padua, Italy: Springer.

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

Marthi, B., Russell, S., & Latham, D. (2005). Writing Stratagus-playing agents in Concurrent ALisp. To appear in D.W. Aha, H. Muñoz-Avila, & M. van Lent (Eds.) *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop* (Technical Report). Washington, DC: Navy Center for Applied Research in Artificial Intelligence, Washington, DC.

Muñoz-Avila, H., & Aha, D.W. (2004). On the role of explanation for hierarchical case-based planning in real-time strategy games. In P. Gervás & K.M. Gupta (Eds.) *Proceedings of the ECCBR 2004 Workshops* (Technical Report 142-04). Madrid, Spain: Universidad Complutense Madrid, Departamento di Sistemos Informáticos y Programación.

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

Ponsen, M.J.V., Lee-Urban, S., Muñoz-Avila, H., Aha, D.W. & Molineaux, M. (2005b). Stratagus: An open-source game engine for research in real-time strategy games. Paper submitted to this workshop.

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.

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

Ulam, P., Goel, A., & Jones, J. (2004). Reflection in action: Model-based self-adaptation in game playing agents. In D. Fu & J. Orkin (Eds.) *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop* (Technical Report WS-04-04). San Jose, CA: AAAI Press.

Wendler, J., Kaminka, G. A., & Veloso, M. (2001). Automatically improving team cooperation by applying coordination models. In B. Bell & E. Santos (Eds.), *Intent Inference for Collaborative Tasks: Papers from the AAAI Fall Symposium* (Technical Report FS-01-05). Falmouth, MA: AAAI Press.

Wendler, J., & Lenz, M. (1998). CBR for dynamic situation assessment in an agent-oriented setting. In D. W. Aha & J. J. Daniels (Eds.), *Case-Based Reasoning Integrations: Papers from the AAAI Workshop* (Technical Report WS-98-15). Madison, WI: AAAI Press.

# 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; USA; molineau@aic.nrl.navy.mil

**Abstract**

Stratagus is an open-source game engine that can be used for artificial intelligence (AI) research in the complex domain of real-time strategy games. It has already been successfully applied for AI studies by several researchers. The focus of this paper is to highlight Stratatgus' useful features for AI research rather than compare it with other available gaming engines. In particular, we describe Stratagus' 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). The Stratagus engine has already been extensively used by researchers (e.g., Guestrin *et al*., 2003; Ponsen & Spronck, 2004; Ponsen *et al*., 2005; Marthi, Russell, & Latham, 2005; Aha *et al*. 2005) and students (e.g., for class projects on game AI).

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)


**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.

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 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 (2005). 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). For example, by changing simple text-based files one can easily add new units or tweak existing unit characteristics (e.g., increase unit speed), add new buildings, maps and set new triggers (e.g., define what constitutes a win).
- *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 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.

## 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 sent 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

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

| **Game AI Actions** | |
|---|---|
| AiAttackWithForce (forceID)<br>*e.g., AiAttackWithForce(1)* | Command the AI to attack an enemy with all units belonging to a predefined force. |
| AiCheckForce (ForceID)<br>*e.g., AiCheckForce(1)* | Check if a force is ready. |
| AiForce (ForceID, {force})<br>*e.g., AiForce(1,{"unit-grunt", 3})* | Define a force: determine the type and number of units that belong to it. |
| AiForceRole (forceID, role)<br>*e.g.,AiForceRole(1,"defend")* | Define the role of a force: Assign it either an defensive or offensive role. |
| AiNeed (unitType)<br>*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)<br>*e.g., AiMoveTo(1,200,200)* | Command an army to move to a specific location on the map. |
| AiResearch (researchType)<br>*e.g., AiResearch("upgrade-sword")* | Command the AI to pursue a specific research advancement. |
| AiUpgradeTo (unitType)<br>*e.g.,AiUpgradeTo("upgrade-ranger")* | Command the AI to upgrade a specific unit. |
| **Miscellaneous Actions** | |
| GameCycle () | Request the current game cycle. |
| GetPlayerData (playerID, info)<br>*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)<br>*e.g., SetFastFowardCycle(200000)* | Set the fast forward cycle (only for single player games). |
| UseStrategy(strategyID)<br>*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)<br>*e.g., FORCE(1)* | Informs TIELT that all units for an army are combat-ready . |
| GAMECYCLE (gameCycle )<br>*e.g., GameCycle(20000)* | Sent in response to the GameCycle action. It returns the current game cycle. |
| GAMEOVER (result, TieltScore, endCycle)<br>*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)<br>*e.g., INFO(1,"New Peasant ready")* | Informs TIELT about game events (e.g., 'under attack', or 'building or unit is complete'). |
| PLAYERDATA (playerID, info, content)<br>*e.g., PLAYERDATA (1,TotalKills, 10)* | Sent in response to the GetPlayerData action. It returns information for a specific player. |

specifying the number and type of the units belonging to 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).

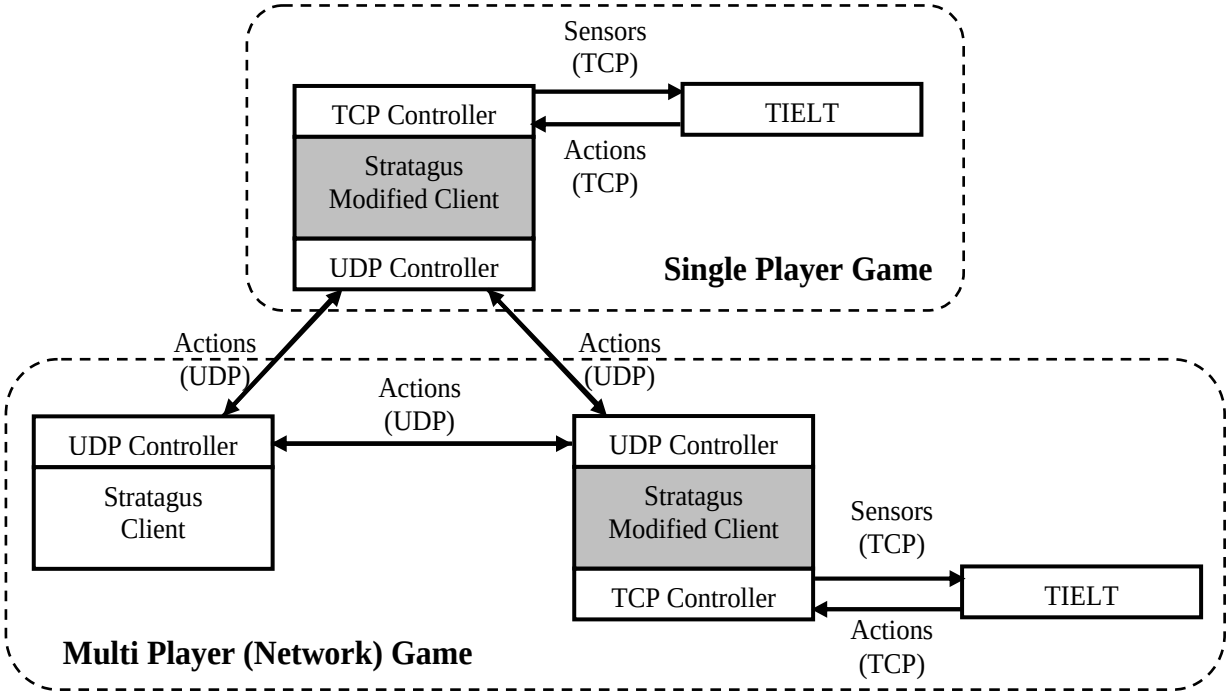


**Figure 2**: Schematic view of the communication between Stratagus clients, modified clients (i.e., those with TIELT integration), and TIELT.

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. In the current network implementation as illustrated in Figure 2, Stratagus does not hide information from clients; instead, clients are connected through a peer-to-peer connection and send each other UDP packets containing information on the action taken during the game by any of the participants.

## 5. Example Application

In this section, we describe an example application of our Stratagus-TIELT integration. We implemented SR, a decision system that employs a 'soldier's rush' strategy in Wargus, 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 training and using an offensive army can be repeated by SR to continuously launch attacks on the enemy.

After a game ends, 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.

## 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 TIELT's sensors.

In our future work we will improve the control a TIELT-integrated decision system has over the AI by adding more game actions (e.g., we will allow TIELT to interface with the local AI). However, the API may be subject to change because different decision systems often require different APIs. Because Stratagus is open-source, researchers can change or extend the API as needed. We will also extend the sensors and provide TIELT with more detailed information on the game state. Greater interaction between Stratagus players and TIELT-integrated decision systems can be aided by adding a TIELT specific "toolbox" panel to the games, which would visually allow human players to dialog with TIELT. Decision systems can then be used as advisors. For example, given 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). 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.

While Stratagus does have limitations in its use for AI research (e.g., some of its behaviors are hardwired, its graphics are not separable from the logic), it provides access to important high-level behaviors, and we expect it will receive increasing attention from AI researchers.

## Acknowledgements

This research was sponsored by 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 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.

Marthi, B., Russell, S., & Latham, D. (2005). Writing Stratagus-playing agents in Concurrent ALisp. To appear in this volume

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.

Stratagus (2005), A Real-Time Strategy Engine, [http://stratagus.sourceforge.net]

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.

# Towards Integrating AI Story Controllers and Game Engines: Reconciling World State Representations

**Mark O. Riedl**
Institute for Creative Technologies
University of Southern California
13274 Fiji Way, Marina Del Rey, CA 90292 USA
riedl@ict.usc.edu

## Abstract

Recently, many AI researchers working on interactive storytelling systems have turned to off-the-shelf game engines for simulation and visualization of virtual 3D graphical worlds. Integrating AI research into game engines can be difficult due to the fact that game engines typically do not use symbolic or declarative representations of characters, settings, or actions. This is particularly true for interactive storytelling applications that use an AI story controller to subtly manipulate a virtual world in order to bring about a structured narrative experience for the user. In this paper, I describe a general technique for translating between an arbitrary game engine's proprietary and procedural world state representation into a declarative form that can be used by an AI story controller. The work is placed in the context of building a narrative-based training simulation.

## 1 Introduction

Interactive storytelling systems are applications in which a story is presented to a user in such a way that the user has the ability to affect the direction and possibly even the outcome of story. The ability of the user to impact the story arc and outcome suggests a branching story structure [Riedl and Young, 2005]. Advanced 3D graphics rendering capabilities, such as those found in modern computer games, makes it possible and even desirable to implement interactive storytelling by situating the user in a 3D graphical story world. In this approach, the user, through her avatar, is a character in the story and is able to interact with the environment and other characters and possibly even play a role in the plot.

Computer games are perhaps the most pervasive example of an interactive storytelling system. However, in computer games, the user's interactivity with the world is typically bounded in such a way that the user's actions do not actually have an impact on the story arc. That is, computer games use story to motivate action but typically have little or no branching.

AI techniques have been applied to the problem of interactive storytelling for entertainment and training. A common technique among AI research in interactive storytelling is to separate the AI story control elements from the graphical, virtual world. An *automated story director* – often referred to as a *drama manager* [Kelso, Weyhrauch, and Bates, 1993] – is responsible for keeping the user and any non-player characters (NPCs) on track for achieving a particular narrative-like experience. An automated story director maintains a representation of the structure that the emergent user experience is expected to conform to and exerts influence over the user, the virtual world, and the NPCs in order to achieve this. Examples of interactive storytelling systems that use some notion of an automated story director are [Weyhrauch, 1997], [Mateas and Stern, 2003], [Szilas, 2003], [Young et al., 2004], and [Magerko et al., 2004]. Some interactive storytelling systems such as [Cavazza, Charles, and Mead, 2002] do not use an explicit story director. Instead, such systems rely on story to *emerge* from the behaviors of the NPCs and the user [Aylett, 2000].

Recently, many AI researchers working on interactive storytelling systems have turned to off-the-shelf game engines for simulation and visualization of virtual 3D graphical worlds (e.g. [Cavazza, Charles, and Mead, 2002], [Seif El-Nasr and Horswill, 2003], [Young et al., 2004], and [Magerko et al., 2004]). Game engines provide sophisticated graphical rendering capabilities with predictable frame rates, physics, and other advantages so that AI researchers do not need to devote resources to "reinventing the wheel."

Integrating AI research into game engines can however be difficult due to the fact that game engines typically do not use symbolic or declarative representations of characters, settings, or actions [Young and Riedl, 2003][1]. Action representations for many game engines such as first-person shooters are expressed as "micro-actions" – mouse clicks, key presses, etc. – and state representations are based on continuous vector positions, rotations, velocities and "flag" variables. AI character or story controllers such as [Cavazza, Charles, and Mead, 2002], [Young et al., 2004], and [Magerko et al., 2004] use declarative, symbolic representations of character, world, and story state. For example, `Walk(agent`$_1$`, loc`$_1$`, loc`$_2$`)` is a discrete action and `(at`

[1] One exception is the commercial game described in [Orkin, 2004], which uses both proprietary and symbolic world state representations.

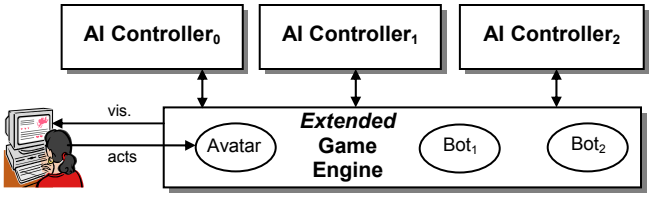


**Figure 1:** Generic architecture for an interactive storytelling system.

$\texttt{agent}_1\ \texttt{loc}_1)$ is a discrete term partially describing the world state.

AI technologies often use declarative and/or symbolic representations of the virtual environment, simplifying the world to only the aspects that are necessary for computation. Declarative representation facilitates robust reasoning about the simulation state such as regressive problem solving (e.g. planning and re-planning), predictive analysis (e.g. predicting plan failure), user goal recognition, agent belief-desire-intention modeling, and others. As far as automated story direction is concerned, [Young, 1999] describes the advantages of using a declarative, partial-order plan representation for narrative: (a) causal dependencies between actions ensure that all events are part of causal chains that lead to the outcome; (b) planning algorithms are general problem-solvers that "solve the problem" of piecing together the events of a narrative that achieves a particular outcome; and (c) story plans can be repaired by replanning to allow interactivity.

For an AI character or story controller to be closely integrated with a proprietary game engine, the AI system must transform the proprietary non-declarative world state in the game engine into a declarative form. For example, Mimesis [Young et al., 2004] overrides the game engine's user input routines in order to detect discrete user actions. The discretized user actions are correlated with plan operators that have declarative preconditions and effects with which to reason about changes to the world wrought by the user. Not all AI controllers use plan operator representations.

The remainder of the paper is laid out as follows. In Section 2, we describe a generic architecture for an interactive storytelling system. In Section 3, we describe a general technique for translating proprietary and procedural world representation from an arbitrary game engine into a declarative form that can be used by AI controllers such as automated story directors and autonomous agents. In Section 4, we briefly describe a narrative-based training simulation that motivates the need for the integration of an automated story director and autonomous agents with an arbitrary game engine.

## 2 A Generic Interactive Storytelling Architecture

A generic architecture for an interactive storytelling system is given in Figure 1. The architecture is based around a game engine and one or more AI controllers. AI controllers can be automated story directors or autonomous agents. Autonomous agents control the decision-making processes of non-player characters (NPCs). Even though a virtual world contains non-player characters, it is not necessarily the case that there must be an AI controller for each NPC. An automated story director, in addition to maintaining a branching narrative model, can be implemented such that it also directs the behaviors of NPCs, as in [Young et al., 2004]. If there is an automated story director, there is typically only one director. There does not necessarily have to be an automated story director for there to be interactive storytelling, as in [Cavazza, Charles, and Mead, 2002].

The game engine can be any game or simulation system that supports or can be extended to support interface to the automated story director and the virtual actors. Figure 1 refers to the game engine as an *extended* game engine because of its support for AI controllers. The game engine extensions are described in further detail in the next section. In general, the game engine is responsible for simulating a virtual world plus graphical presentation of the virtual world to the user who is embodied by an avatar. Each non-player character (NPC) that the trainee will be expected to interact with is represented graphically in the game engine as a *bot*. A bot is a physical manifestation of an NPC based on the proprietary graphical rendering of the character's body in the virtual world. Aside from processes for rendering, animating, and low-level path-planning, there is little or no intelligence in the bot. The higher-level intelligence of an NPC is relegated to one of the AI controllers that receive updates from the virtual world and issues control commands to bots.

It is possible – and sometimes even desirable – for the various AI controllers to communicate with each other to coordinate behaviors and world representations. For the remainder of this paper, we shall assume that all the AI controllers in an interactive storytelling system use the same world state representations and it is only the game engine that does not. Furthermore, we shall assume that there is at least one AI controller that is an automated story director.

## 3 A Middleware Substrate for Integrating a AI Controllers into a Game Engine

In the generic architecture for an interactive storytelling system described in the previous section, the automated story director and any autonomous agents are assumed to use a shared declarative representation of world state. The game engine, however, is not assumed to use a world state representation that is deterministic or shared with the other components. In fact, it is assumed that the game engine does *not* use a declarative representation! However, it is vital that the AI controllers are aware of the state of the simulation in the game engine. An automated story director, in particular, must be aware of the changes to the world state that are caused by the actions of the user. Agents must also be aware of changes in the world state to be able to react appropriately and believably. The solution to reconciling world state representations between an arbitrary game engine and AI controllers described here is motivated by the generic architecture. However, it is our belief that the solu-

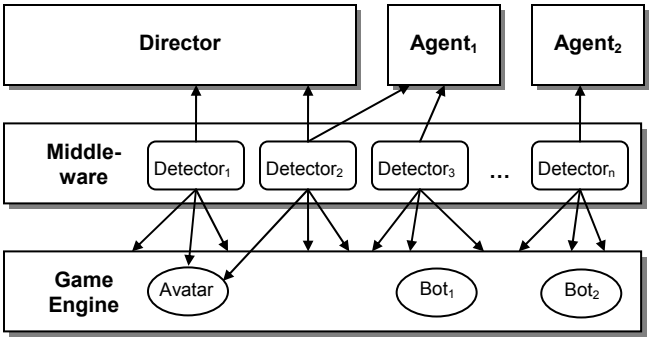


**Figure 2:** Middleware for detecting and translating game engine world state.

tion is general enough to apply to other interactive storytelling systems.

The procedural, non-declarative state representation maintained by the game engine must be translated into a declarative representation shared by the automated story director and the actors. One way to transform the game engine's representation into a declarative form is through a middleware substrate that interfaces directly with the game engine through scripting or through an API such as that proposed in [van Lent, 2004]. A middleware approach may be inappropriate for computer game production where AI is only guaranteed a small portion of a computer's processing time and efficiency is therefore essential. However, AI game research is not necessarily beholden to production constraints. Researchers in automated story direction often resort to a distributed architecture where graphical and simulation processing occurs on one computer while AI processing occurs on one or more other computers. In this case, a middleware solution is favorable because it abstracts away the procedural nature of the game engine and allows AI researchers to focus on theories, algorithms, and cognitively plausible representations of narrative.

The proposed middleware substrate implements *state detectors* and *proprioceptive detectors* that efficiently access the game engine's proprietary state variables (such as object locations, rotations, velocities, flags, etc.) to derive discretized information about the game engine and push that information to any system modules that request updates. Figure 2 shows a conceptualization of the middleware substrate.

## 3.1 State Detectors

State detectors determine if discrete state declarations are true or false. For each atomic, ground sentence used by the automated story director or an autonomous agent to represent some aspect of world state, there must be a detector that can recognize whether it is true or not in the simulation. Note that for efficiency purposes a single state detector can be responsible for more than one fact.

An example of a state detector is one that determines whether `(in-speaking-orientation user ?npc)` is true for some non-player character in the world, meaning the NPC and player are close by, facing each other, etc. When a sentence of this form is true, the player and agents can engage in conversation (either can take the initiative). This world state can be important to agents who need to know if they can engage the user in dialogue and to an automated director if the story requires some conversational exchange between user and another character before the story can continue. Whether a sentence of this form is true or not can be computed from the distance between the user's avatar and the bot and the directional orientation of avatar and bot towards each other. A single detector can be responsible for determining whether the relationship holds or does not hold for all NPCs in the world, as opposed to state detectors for each NPC.

## 3.2 Proprioceptive Detectors

Proprioceptive detectors apply only to the user's avatar and are used to determine if the user has performed certain discrete actions. The purpose of a proprioceptive detector is for the user's avatar to declare to listening AI controllers, "I, the user's avatar, have just performed an action that you might have observed." Bots do not need proprioceptive detectors because their behavior is dictated by an AI controller; success or failure of bot behaviors can be confirmed by comparing the expected world state changes with actual world state changes. Agents can be made aware of each others' observable actions through direct back-channel communication.

An example of a proprioceptive detector is one that determines when the user has moved from one discrete location in the world to another. That is, it determines whether the declarative action `Walk(user, ?loc1, ?loc2)` has been performed. This declaration can be important to agents who observe the user leaving or arriving. This declaration can also be important for an AI controller such as an automated director that needs to know about the effects of the action: `(at user ?loc2)` and $\neg$`(at user ?loc1)`. However, this information can be derived through state detectors as well without concern for how those effects were achieved (e.g. `Walk` versus `Run`).

## 3.3 Detector Integration with the Game Engine

While state changes can be determined from discrete action representations such as those used in planning systems, the purpose of detecting user actions is primarily for sensor input to the autonomous agent AI controllers. When NPCs and the user interact, the agents will need to know the observable actions that the user performs, whether they are physical or discourse acts, instead of inferring them from local world state changes. State detectors, however, are still necessary above and beyond proprioceptive detectors because the user's input into the game engine is through "micro-actions" – mouse clicks, key presses, etc. Many micro-actions string together to produce discrete actions. However, it may be the case that the user performs micro-actions that change the world state but are not aggregated into a recognizable discrete action. Thus, it is possible for the simulation state in the game engine to become out of sync with that of the story director and the actors. One solution is to define discrete action representations at a finer level of

detail. The approach advocated here is to detect high-level actions that are necessary for user-agent interactions and allow state detectors to fill in the rest.

It is possible to integrate symbolic and procedural representations. Orkin [2004] describes a technique for individual AI characters to perform real-time, goal-oriented planning in a game engine using action representations that combine both symbolic preconditions and effects with procedural preconditions and effects. However, it is not clear whether such a technique could be applied to an AI story director since a story director does not directly act in the world as an AI character does. We believe the middleware substrate approach advocated in this paper to be more general and flexible.

# 4 Towards a Narrative-based Training Simulation

In this section, we describe an interactive storytelling system built on top of a game engine that uses a multitude of AI controllers, including an automated story director and several autonomous agents. The various types of AI controllers use different AI technologies and consequently have different declarative world representations. The middleware substrate approach is capable of meeting all of the information requirements of the heterogeneous collection of AI controllers without requiring any to be tightly integrated with the game engine. The following discussion describes the purpose of the system and motivates the necessity of having different types of AI controllers operating simultaneously.

The interactive storytelling system we describe here is a *narrative-based training simulation*. Simulations have been used for training skills and situation awareness. For training tacit knowledge such as the operational and procedural skills required for adaptive military leadership, it is advantageous to situate the trainee in a realistic environment. A virtual reality simulator is a good start. However, it is advantageous that trainees are situated in an environment whose situational evolution is directed. The advantages are that the trainee can be exposed to a larger context, multiple learning objectives can be strung together in a particular order, and the trainee can gain valuable experience in dealing with successions of problems that are interrelated in a lifelike manner (instead of running separate, and thus disjoint, training exercises). Since pure simulations are open-ended, there is no guarantee that the world will evolve in a sustainable manner. That is, the structure of the trainee's experience is not guaranteed to contain certain events or situations after the first few actions. The actions of the trainee and any autonomous agents can cause the world to evolve in a way that is undesirable from the perspective of the trainee being exposed to situations of pedagogical value.

## 4.1 Story Control for Training

Our narrative-based training simulation uses a high-level AI control structure to try to manipulate a simulation such that the world state, at least at a high level of abstraction, evolves in way that corresponds to a given model of narrative. The way in which this is achieved is necessarily different from more entertainment-oriented interactive storytelling systems. One difference between training and entertainment applications is that the trainee must learn about second- and third-order effects of their actions, meaning that it is important for realistic emergence of situation. An entertainment application can ignore the effects on the world that do not contribute to the story. This emergence [Aylett, 2000] must be carefully balanced against the overarching, high-level narrative model of the story director.

A second difference between training and entertainment applications of interactive storytelling is that in systems for training the AI story controller should be relatively resilient to branching. That is, the given high-level narrative model achieves a certain sequence of learning objectives that has pedagogical value. Branching to alternative narrative arcs should be possible, but only when absolutely necessary. Furthermore, any alternative narrative branch should be as similar as possible to the original narrative model and contain the same pedagogical value. Branching story in entertainment applications only require the consequent alternative narrative branches to have entertainment value and can consequently deviate more in order to comply with the apparent desires of the user.

A third difference between training and entertainment applications of interactive storytelling is that in systems for training, the automated story director should not intervene with the actions of the trainee. This is important because one does not want the trainee to learn that certain incorrect or inappropriate actions are okay because they will be caused to fail. It is also important for the trainee to learn from her mistakes, even if it means "game over." This is in contrast to [Young et al., 2004] which describes an entertainment-oriented interactive storytelling system that is capable of subtly intervening with user actions to preserve the content of the narrative. For training, user actions that are not in accordance with the narrative model should either cause an alternative branch to be taken or result in failure with feedback about what was wrong.

## 4.2 Architecture for a Narrative-Based Training Simulation

We believe that we can achieve the nuances of interactive storytelling for training purposes with a combination of automated story direction and semi-autonomous agents. The heterogeneity of AI controllers makes a middleware approach to integration with a game engine desirable. The architecture for the narrative-based training simulation is given in Figure 3.

The three main components to the architecture are: the game engine, the automated story director, and the semi-autonomous virtual actors. The game engine is any game engine or simulation that includes the middleware substrate for interfacing with an automated story director and AI characters. The automated story director is an AI controller that has a branching narrative model and is capable of determining whether the simulation state in the game engine matches – or at least is not contradictory to – the narrative

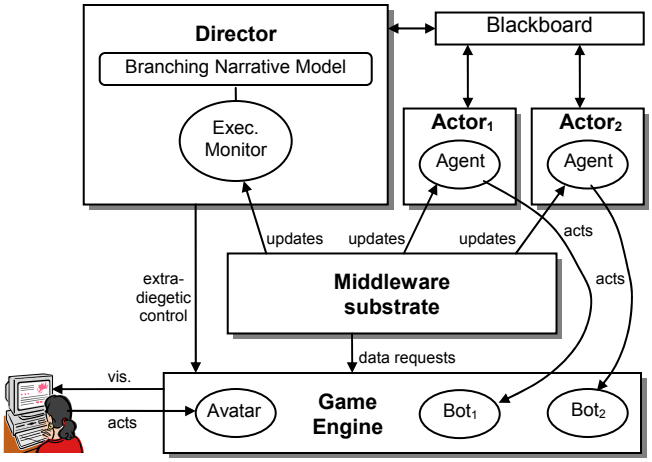


**Figure 3:** Architecture for a narrative-based training simulator.

model. Additionally, the automated story director is capable of manipulating the *extra-diegetic* effects of the game engine as well as the semi-autonomous virtual actors. Extra-diegetic aspects of the game engine are those involving the visualization of the world such as music, cinematography (e.g. [Jhala, 2004]), and lighting (e.g. [Seif El- Nasr and Horswill, 2003]), and not the actual simulation state.

Each non-player character (NPC) that the trainee will be expected to interact with is represented by a pairing of two components: a bot and an AI controller called an *actor*. Bots are described in Section 2. An actor[2] contains within it an autonomous agent decision-making process that has beliefs, desires, and intentions and uses sensors to react to the environment as it attempts to achieve its intentions. Examples of AI character technologies that have been applied to animated, virtual agents are Soar [Rickel et al., 2002], HAP [Loyall, 1997], ABL [Mateas and Stern, 2003], and hierarchical task networks [Cavazza, Charles, and Mead, 2002]. We do not make any commitment to the type of agent technology used in the narrative-based training simulation except that the agent decision-making process is wrapped in additional logic that is aware of the narrative goals of the automated director and is *directable*. A directable agent is one whose behavior and reasoning can be manipulated by an external process [Blumberg and Galyean, 1995; Assanie, 2002]. The actor itself is aware of the narrative goals of the automated director and takes direction from the automated director. Direction from the automated director takes one of two forms:

- Direction to achieve some world state that is desirable to the automated director and moves the plot forward.
- Direction that constrains the internal, reactive decision-making process – which is only aware of its own beliefs, desires, intentions and sensory input from the environment – from choosing actions, behaviors, or dialogue that contradicts or invalidates the automated director's narrative model.

Both types of direction are essential. The first type of direction is the primary mechanism through which the automated director pushes a story forward and is necessary because the actors cannot be relied on to autonomously make decisions that are always favorable to the automated director. The second type of direction is important in any situation where actors do have some autonomy to form and reactively pursue their own goals. Autonomy means that actors can potentially choose actions, behaviors, or dialogue that contradicts the narrative model of the automated director and even make it impossible for the narrative and all of its branches to continue coherently.

The final component in Figure 3 is a blackboard. Rist, André, and Baldes [2003] demonstrate a blackboard to be an effective channel of communication between autonomous agents and story directors. Here, the blackboard serves two purposes. First it contains a specific world state that is shared between the director and the actors. Note that this world state may be different than the world state held by the actor's internal agent processes because the internal agent processes are responsible for reacting to local environmental conditions and should not necessarily be aware of things outside the scope of its senses. Actors only receive state updates and knowledge about user avatar actions that are within range of the bots' senses and necessary for reactivity within the environment. The blackboard, however, contains a global representation of the entire virtual world, including the internal state of all the NPCs. This privileged information is only accessible to the directable processes that wrap the autonomous agent decision-making processes.

The second purpose of the blackboard is a communication channel between the automated story director and the actors. In particular, the director sends directives to the actors so that they will achieve certain world states that are advantageous to the narrative development as well as constraints so that the actors do not perform actions that make it impossible for the plot to advance. Conceivably, actors can also communicate amongst themselves to coordinate their performances.

## 5 Conclusions

In an interactive storytelling system such as the narrative-based training simulator described here, the graphical rendering of the virtual world and story world characters is separate from the AI control processes for story direction and agent decision-making. Game engines notoriously use proprietary and procedural representations for world state whereas AI controllers such as an automated story director often use declarative and/or symbolic world state representations. The approach presented here is a middleware substrate that uses actor and state detectors to produce declarations about the simulation world state and push state changes onto the story director and autonomous actors. While this approach is taken in the context of the architecture for a narrative-based training simulator, the middleware

[2] Gordon and van Lent [2002] lay out the pros and cons of agents that are realistic models of humans versus agents that are actors.

substrate approach is expected to be general enough to be applicable to many interactive storytelling systems.

## Acknowledgements

The project or effort described here has been sponsored by the U.S. Army Research, Development, and Engineering Command (RDECOM). Statements and opinions expressed do not necessarily reflect the position or the policy of the United States Government, and no official endorsement should be inferred.

## References

[Assanie, 2002] Mazin Assanie. Directable synthetic characters. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 2002.

[Aylett, 2000] Ruth Aylett. Emergent narrative, social immersion and "storification." In *Proceedings of the 1st International Workshop on Narrative and Interactive Learning Environments*, 2000.

[Blumberg and Galyean, 1995] Bruce Blumberg and Tinsley Galyean. Multi-level direction of autonomous agents for real-time virtual environments. In *Proceedings of SIGGRAPH,* 1995.

[Cavazza, Charles, and Mead, 2002] Marc Cavazza, Fred Charles, and Steven Mead. Planning characters' behaviour in interactive storytelling. *Journal of Visualization and Computer Animation*, 13: 121-131, 2002.

[Gordon and van Lent, 2002] Andrew Gordon and Michael van Lent. Virtual humans as participants vs. virtual humans as actors. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 2002.

[Kelso, Weyhrauch, and Bates, 1993] Margaret Kelso, Peter Weyhrauch, and Joseph Bates. Dramatic presence. *Presence: The Journal of Teleoperators and Virtual Environments*, 2(1), 1993.

[Jhala, 2004] Arnav Jhala. *An Intelligent Cinematic Camera Planning System for Dynamic Narratives*. Masters Thesis, North Carolina State University.

[Loyall, 1997] Brian Loyall. *Believable Agents: Building Interactive Personalities*. Ph.D. Dissertation, Carnegie Mellon University, 1997.

[Magerko et al., 2004] Brian Magerko, John Laird, Mazin Assanie, Alex Kerfoot, and Devvan Stokes. AI characters and directors for interactive computer games. In *Proceedings of the 16th Innovative Applications of Artificial Intelligence Conference*, 2004.

[Mateas and Stern, 2003] Michael Mateas and Andrew Stern. Integrating plot, character, and natural language processing in the interactive drama Façade. In *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, 2003.

[Orkin, 2004] Jeff Orkin. Symbolic representation of game world state: Towards real-time planning in games. In *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, 2004.

[Rickel et al., 2002] Jeff Rickel, Jon Gratch, Randall Hill, Stacy Marsella, David Traum, and Bill Swartout. Toward a new generation of virtual humans for interactive experiences. *IEEE Intelligent Systems*, July/August 2002.

[Riedl and Young, 2005] Mark Riedl and R. Michael Young. From linear story generation to branching story graphs. In *Proceedings of the 1st Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2005.

[Rist, André, and Baldes, 2003] Thomas Rist, Elisabeth André, and Stephen Baldes. A flexible platform for building applications with life-like characters. In *Proceedings of the 2003 International Conference on Intelligent User Interfaces*, 2003.

[Seif El-Nasr and Horswill, 2003] Magy Seif El-Nasr and Ian Horswill. Real-time lighting design for interactive narrative. In *Proceedings of the 2nd International Conference on Virtual Storytelling*, 2003.

[Szilas, 2003] Nicolas Szilas. IDtension: A narrative engine for interactive drama. In *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, 2003.

[van Lent, 2004] Michael van Lent. Combining gaming and game visualization with traditional simulation systems. Invited talk at the *Serious Games Summit*, 2004.

[Weyhrauch, 1997] Peter Weyhrauch. *Guiding Interactive Fiction*. Ph.D. Dissertation, Carnegie Mellon University.

[Young, 1999] R. Michael Young. Notes on the use of planning structures in the creation of interactive plot. In *Proceedings of the AAAI Fall Symposium on Narrative Intelligence*, 1999.

[Young and Riedl, 2003] R. Michael Young and Mark Riedl. Towards an architecture for intelligent control of narrative in interactive virtual worlds. In *Proceedings of the 2003 International Conference on Intelligent User Interfaces*, 2003.

[Young et al., 2004] R. Michael Young, Mark Riedl, Mark Branly, Arnav Jhala, R.J Martin, and C.J. Saretto. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 1: 51-70, 2004.

# An intelligent decision module based on CBR for C-evo

**Rubén Sánchez-Pelegrín**
CES Felipe II, Aranjuez, Madrid
rsanchez@cesfelipesegundo.com

**Belén Díaz-Agudo**
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
belend@sip.ucm.es

## Abstract

C-evo is a non-commercial free open-source game based on "Civilization", one of the most popular turn-based strategy games. One of the most important goals in the development of C-evo is to support the creation of Artificial Intelligence (AI) modules. Thus everyone can develop his own AI algorithm and watch it play against humans or against other modules. We have developed one of such AI modules. In this paper we describe it and test some techniques based on Case Based Reasoning, a well-known AI technique that has not very commonly been used in video games.

## 1 Introduction

C-evo is a non-commercial game project based on the famous Sid Meier's "Civilization II" by Microprose and has many basic ideas in common with it. C-evo is a turn-based empire building game about the origination of the human civilization, from the antiquity until the future[1].

One of the most important goals in the development of C-evo is to support the creation of AI modules. Thus everyone can develop his own AI algorithm and watch it play against humans, or against other AI modules. C-evo provides an interface with the game engine that enables developers to create an AI module for managing a civilization. AI modules can gather exactly the same information that a human player can, and can make exactly the same actions.

There are another projects similar to C-evo. Probably the most popular is FreeCiv[2][Houk, 2004; Ulam *et al.*, 2004], also an open-source game based on Civilization. We chose C-evo because is more oriented to AI development. There are many different AI modules developed for C-evo, and this makes it a good testbed for evaluating the performance of our module.

Section 2 briefly summarizes the main rules of the game. Section 3 describes our AI module using CBR to select a military unit behavior. Section 4 concludes this paper and describes some lines of future work.

[1]http://www.c-evo.org/

[2]http://www.freeciv.org/

## 2 Game rules

To play this game means "*to lead the nation of your choice through six millenia, from the first settlement until the colonization of space. Your way is full of difficulties. Resources are limited, wild areas take centuries for cultivation. Famine and disorder will be a constant threat for your growing cities. Your foresight is in demand. Expand your empire and improve your cities. Build barracks, universities and power plants, or even some of the famous wonders of the world. Discover new technologies and weapons. Carry on with exploration in order to search for fertile lands and to complete your world maps. Build sails to discover far continents.*"

There are different nations coexisting during each game. The way to win this game is either exterminate all the other civilizations or colonize the stars, i.e, complete a transstellar colony ship.

Each nation is formed by units and cities. Units are the mobile part of the nation. While units are good for exploration, conquest and defense, cities are the source of growth, production and research.

The world map is divided into square tiles. Each tile has a certain terrain type, which defines its properties: resource production, defense bonus and cost of movement for units which move onto it. One tile can not contain more than one city, but an unlimited number of units of one tribe.

A nation can discover several advances. Most of them make special buildings, unit features, unit designs and/or government forms available. To pursue a particular advance, the knowledge of its prerequisites is required.

During its turn a nation could order as many actions as it desires. Each action can be related with cities, units or general actions.

To support the creation of Artificial Intelligence(AI) the game has a special architecture, based on exchangeable competitor modules, that allows the exchange of the AI for every single player. Thus anyone can develop his own AI algorithm in a DLL apart from the game in the language of his choice, and watch it play against humans, against the standard AI or other AI modules[3].

[3]The manual to create an artificial intelligence module for C-evo is found in http://www.c-evo.org/aidev.html

Figure 1: C-evo screenshot

## 3 AI project for C-evo

As we have introduced, C-evo provides an interface with the game engine that enables developers to create an AI module for managing a civilization. AI modules can gather exactly the same information that a human player can and make exactly the same actions. Indeed, the player interface is another module that uses the same interface to communicate with the game core. This way AI modules can't cheat, a very typical way of simulating intelligence in video games.

Our goal is to develop an AI module, using advanced AI techniques. In particular we use Case-Based Reasoning (CBR) [Leake, 1996; Kolodner, 1993] for solving some of the decision problems presented in the management of an empire. There are low-level problems (tactical decisions) and high-level (global strategy of the empire). Examples of low-level problems are choosing next advance to develop, selecting government type, setting tax rates, diplomacy, city production, behavior of a unit.

Even there are different problems that could be solved using CBR, in this first stage we are focusing in a single problem: the selection of a military unit behavior, a tactical problem concerning action selection.

### 3.1 A concrete problem: military unit behavior

We have focused on a low-level problem: the control of military units. We have chosen this as the first problem to work on because we think it have big impact in the result of the game.

At this time all the other decisions are taken by hand written code. We haven't put much effort on developing it, so we can't expect our AI module to be really competitive with others.

For controlling units, our AI module assigns missions to them. These missions take control of the unit behavior, until the mission is completed. When a new unit is created, or an existing one completes its mission, a new one is assigned to the unit, using CBR for making this decision.

We have created three missions for military units:

- Exploring unknown tiles around a given one. The unit goes to the selected tile and moves to the adjacent tile which has more unknown tile around it. If none of its adjacent tiles has unknown adjacent tiles, the mission concludes.
- Defending a given tile. The unit goes to the tile and stay there to defend the position.
- Attacking a given tile. The unit goes to the tile and attacks any enemy defending it.

Our cases are stored in a XML file, and each one is composed of:

- A description of the unit features, and the state of the game relevant to the problem. This description gathers information about the military unit and about its environment. There are features that stores the values for attack, defence, movement and experience of the unit. From the environment, we are interested on information about own cities, enemy cities and unknown tiles. For every city (own or enemy) there are features that stores its distance to the unit, and defence of the unit that defend the city tile. For our own cities, there is also a feature for the number of units in the city tile.[4] For every known tile surrounded by any unknown tile there are a feature that stores its distance to the unit.
- The solution of the problem, that is the mission assigned to the unit. It can be attack one of the enemy cities, defend one of the own cities, or explore from one of the known tiles surrounded by unknown tiles.
- The result of the case; a description of the outcome of the mission, composed by figures extracted from the game engine. It provides features that stores the duration in turns of the mission, number of attacks won, number of defences won, number of cities enemy conquered, number of own cities lost during the mission, number of explored tiles, and whether the unit died or not during the mission.

When a new problem comes up, we select the solution we are going to take basing on the cases stored in the case base.

We follow the next steps:

- We create a new query, which description is that of the problem we are dealing with. We are going to call it query problem. Description features are obtained from the game engine.
- We retrieve from the case base a set of cases relevant for the current problem. In particular, we retrieve the 50 most similar cases to that of the query problem. Our similarity function, as usual, gets two case description, and returns a measure of how similar they are.

  In the last term, we compare individual features and combination of two or more of them. For example, we not only compare attack with attack and defense with defense, but also the proportion between attack and defense. This is because units with similar attack-defense ratios usually are used in a similar way, although the total amount of attack and defense was very different.

[4] The AI interface doesn't provide this figure for enemy cities.

They could be units from different stages of history. Similarity between individual features is the proportion between them:

$$sim(x, y) = \frac{min(x, y)}{max(x, y)}$$

For determining the similarity between two problem descriptions, we calculate two partial similarities, the similarity between unit features $S_u$ and the similarity between world situations $S_s$.

Similarity between unit features is obtained using weighted average of its local features, considering the attack, defense and experience feature less relevant than movement and proportions between attack and defense:

$$S_u = \frac{S_a + S_d + S_e + 2S_m + 2S_{ar} + 2S_{dr}}{9}$$

Where: $S_a$ – similarity between attack feature.
$S_d$ – similarity between defense feature.
$S_e$ – similarity between experience feature.
$S_m$ – similarity between movement feature.
$S_{ar}$ – similarity between attack ratio.
$S_{dr}$ – similarity between defense ratio.

Similarity between world situations $S_s$ is calculated using arithmetic average of several similarities of features and combinations of features.

Finally, the global similarity $S$ is the product of local similarities:

$$S = S_u S_s$$

- We calculate the profit ($P$) obtained in each of the retrieved cases. For this purpose we use a function that aggregates the case result features, returning a single value that is the measure of the success obtained in the mission. The case result features are number of explored tiles ($t$), number of attacks won ($a$), number of defences won ($d$), number of conquered enemy cities ($c_w$), number of own cities lost during the mission ($c_l$), number of turns that the mission took ($t$) and a boolean feature ($e$) that says if the unit died ($e = 1$) or not ($e = 0$) during the mission.

$$P = \frac{(t + 5a + 10d + 50c_w - 100c_l)^2}{1 + 3e}$$

- We select the set of possible missions we can assign to the unit. Initially they are exploring each of the known tiles with an adjacent unknown tile, defending each of own cities, attacking each of enemy cities. Simplifying, we are going to consider only up to five possible missions; exploring the nearest known tile with an adjacent unknown tile, defending the nearest own city, defending the weakest own city, attacking the nearest enemy city or attacking the weakest enemy city. For each of this possible missions, we select among the retrieved cases those which have that mission as its solution. With these cases, we calculate an expected profit $E$ for the mission. It is an average of the profits of those cases, weighted by the similarity of each case with the problem case.

$$E = \frac{\sum S_i P_i}{\sum S_i}$$

We also calculate an uncertainty measure $U$ for each of the possible missions. It depends on the number of cases used for calculating the expected profit, the variance of their profit, and their similarity with the problem case. This way, if we have got similar expected profit from many cases, probably the real profit of the mission will be very close to the expected profit. In this case we have a low amount of uncertainty. On the other hand, if we have calculated the expected profit from few cases, or from cases with very different profits, the uncertainty will be high.

$$U = \frac{\sum S_i (P_i - E)^2}{\sum S_i}$$

- We randomly modify the expected profit of each of the possible missions. The randomness of the modification depends on the uncertainty; the higher it is, the bigger the randomness is. This way we avoid the algorithm to be too much conservative. If we didn't use this random factor, the algorithm would discard some solutions basing on a single case which gave a bad result. Given the complex nature of the game, this bad result could be a bit of bad luck, and it is good to give a second opportunity to solutions, although once they worked badly.

$$E' = E + rand(0, U)$$

- We select the mission with the highest expected profit. We assign it to the unit, and store it as the solution of the problem case. The problem case is not stored yet in the case based. The mission is tracked for obtaining the result of the case until it is completed.
- When the mission is completed, the query problem becomes a full case, with its result obtained from the game engine. Then this new case is stored in the case base.

The success of the algorithm strongly depends on the representation of the description, solution and result of the cases, and on the functions used for calculating similarity, profit, expected profit, uncertainty and the randomness modifying of the expected profit. All these functions are hand coded, written based on our knowledge of the game. In section 3.3 we discuss how learning could be used for improving them.

### 3.2 Evaluation

C-evo allows to play an AI tournament consistent on several games between several AI modules. We tried to confront our module with a version of itself without learning (a AI *dummy* module); all we did was disabling the case storing, so the module had always an empty case base. In all the games played the result was a tie. Although the learning could improve slightly the performance of the AI module, it was not enough to beat the opponent. We state that this is because the problem solved is only a small part of all the decisions that an

AI module must take. The rest of the module was hand coded in a very simple way, so it doesn't play well enough to get a win. Also, the performance of the algorithm can be improved in several ways, some of them discussed in the next section.

The efficiency was good while the size of the case base was not so big to fill RAM memory. When the case base size grows much the efficiency falls. In next subsection we also discuss some solutions to these problem.

### 3.3 Extensions to the basic approach

In the previously described algorithm, some hand tuned functions were used for calculating things as similarity between cases or profit of a result. This way, the success of the algorithm heavily depends on the accuracy of the human expert modifying these functions. It would be better to use some machine learning technique for these.

To learn the profit of a mission, we can associate each mission result with the final result (win, lost or draw) that the AI module got in the game where the mission took place. This information enables machine learning to associate which mission results tend to lead to each final result. For those results which occurs in a victory, the profit function would tend to give higher values. This way we would have a two level reasoning between raw data extracted from the game engine and decisions taken. For this purpose, Genetic Algorithms, Neuronal Nets, Bayesian Nets or CBR could be used to learn the profit function.

Similarity measure could be learned considering that truly similar cases with similar solutions would tend to lead to similar results. Based on this hypothesis we could learn a similarity function that gave high values of similarity for cases with similar results under the same mission, finding which features of cases are more relevant on this similarity, and to what extent. For this purpose could be used some classical learning algorithms, or other techniques, like genetic algorithms.

A very serious trouble of the algorithm is the growth of the case base, that hurts the efficiency. Usually, case-based systems solve this problem storing only new cases different enough from existent cases. However, case-based systems usually only need to recover the most similar case with the problem, and they reason only from one case. This is possible because identical problems should have identical outputs. In our problem this is false; identical decisions taken on identical situations could lead to different results, due to the freedom of choice of the opponent. Randomness would take part too in most games, although in this one it doesn't exist. We need to store somehow the information of all cases produced, or we will lose valuable information.

A system of indexing the cases would lighten this problem very much. This way wouldn't be needed to calculate the similarity measure for all cases in the database, but only those that are solid candidates to be similar enough to the problem. However, it would not be a complete solution, because the case base size would keep growing indefinitely. It only defers the problem.

The solution to the growth of case base would be clustering cases. If we have identical cases, then we have no need of storing it separately; we can store one case, and have a counter for the number of times it have happened. When reasoning, this case would be weighted by this counter for calculating expected profit and uncertainty. However, case descriptions are complex enough for allowing a very high number of different possible cases. So it would be needed to group very similar cases together in a cluster. A general case would represent the cluster, using a counter for the number of cases implied. The description of the case that represented a cluster would have an average measure of the features that distinguish them, instead of the exact values of each one. Some information would be lost on this process, but it is necessary to maintain the computational feasibility of this method.

The use of CBR could be extended to another low level problems of the game. Each problem would have one case base, with a case representation suitable to that problem. However, some problems could share the same case base; for example, the problem of selecting the military unit to be built in a city could use the case base used for the problem of the military unit behavior. Units produced should be those that produce better results for the current situation of the game. The reasoning would be completely different, but the data would be the same.

High level problems could lead to more challenging strategic decisions, where we think CBR could be a very useful technique, given that it usually works well in situations where the environment is complex, and the rules of its behavior are unknown or fuzzy. Examples of such problems are coordinating units action, or selection of the global strategy for the game.

## 4 Conclusions and Related Work

To the author's knowledge CBR has not typically been used as the main underlying AI technique in commercial video games. There are some research papers like [Gabel and Veloso, 2001; Fagan and Cunningham, 2003], but the industry seems not to use CBR in commercial games, although there are several descriptions of similar techniques, without naming it CBR [Rabin, 2002]. Even so, we consider it is a promising technique for AI in computer games. CBR an AI technique close to human thinking. AI for computer games should not only be intelligent, but also believable [Nareyek, 2004]. We think this makes CBR specially suitable to AI for computer games.

CBR is a natural alternative to rule based system. One of the advantages of CBR is the less effort needed to acquire the knowledge. Rule based systems has been used for example in [Champandard, 2003] and Age of Empires: the Conquerors Expansion [AOE, 2004].

There are also some publications about AI for strategy games. [Ulam *et al.*, 2004] address a tactical decision, the defend city task, using learning by model-based reflection and self-adaptation; when the AI loses a game examines its own reasoning process to identify the failure and repair it. [Houk, 2004] describes a whole AI agent for FreeCiv. For some tasks it uses qualitative spatial reasoning, as proposed before in [Forbus *et al.*, 2002].

Though we already have some experimental results, this project is in an early stage. As shown in Section 3.3, there are still several improvements to be developed.

Also, we want to integrate our AI module with TIELT system [TIELT, 2004] in order to standardize the evaluation. This system enables to give a definition of the game model and a model of the interface with the game engine. This two models will be used for gather and store the information needed by the AI module, and for transmitting its decisions to the game engine. Two more models, the decisions system model and the agent description model define the core of the AI module. And a fifth model, the experiment methodology model defines the test for the evaluation. Implementing the system conforming to this models allows evaluating the system in a standard way.

## References


[AOE, 2004] AOE. Age of empires: the conquerors expansion. http://www.microsoft.com/games/aoeexpansion/, 2004.

[Champandard, 2003] Alex J. Champandard. *AI Game Development. Chapters 11 and 12*. New Riders Publishing, 2003.

[Fagan and Cunningham, 2003] Michael Fagan and Padraig Cunningham. Case-based plan recognition in computer games. In *ICCBR*, pages 161–170, 2003.

[Forbus *et al.*, 2002] Kenneth D. Forbus, James V. Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game ais. *IEEE Intelligent Systems*, 17(4):25–30, 2002.

[Gabel and Veloso, 2001] Thomas Gabel and Manuela M. Veloso. Selecting heterogeneous team players by case-based reasoning: A case study in robotic soccer simulation. Technical report CMU-CS-01-165, Computer Science Department, Carnegie Mellon University, 2001.

[Houk, 2004] Phillip A. Houk. A strategic game playing agent for freeciv. Technical report NWU-CS-04-29, Evanston, IL: Northwestern University, Department of Computer Science, 2004.

[Kolodner, 1993] J.L. Kolodner. *Case-based reasoning*. Morgan Kaufmann, Calif., US., 1993.

[Leake, 1996] D. Leake. *Case-Based Reasoning: Experiences, Lessons, & Future Directions*. AAAI Press / The MIT Press. ISBN 0-262-62110-X, 1996.

[Nareyek, 2004] Alexander Nareyek. Ai in computer games. *ACM Queue*, 1(10):58–65, 2004.

[Rabin, 2002] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc., Rockland, MA, USA, 2002.

[TIELT, 2004] TIELT. http://nrlsat.ittid.com/, 2004.

[Ulam *et al.*, 2004] Patrick Ulam, Ashok Goel, and Joshua Jones. Reflection in action: Model-based self-adaptation in game playing agents. In Dan Fu and Jeff Orkin, editors, *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop*. San Jose, CA: AAAI Press, 2004.

# A Model for Reliable Adaptive Game Intelligence

**Pieter Spronck**
Universiteit Maastricht
Institute for Knowledge and Agent Technology
P.O. Box 616, 6200 MD Maastricht, The Netherlands
{p.spronck}@cs.unimaas.nl

## Abstract

Adaptive game AI aims at enhancing computer-controlled game-playing agents with the ability to self-correct mistakes, and with creativity in responding to new situations. Before game publishers will allow the use of adaptive game AI in their games, they must be convinced of its reliability. In this paper we introduce a model for Reliable Adaptive Game Intelligence (RAGI). The purpose of the model is to provide a conceptual framework for the implementation of reliable adaptive game AI. We discuss requirements for reliable adaptive game AI, the RAGI model's characteristics, and possible implementations of the model.

## 1 Introduction

The behaviour of computer-controlled agents in modern computer games is determined by so-called 'game AI'. For artificial intelligence research, game AI of complex modern games (henceforth called 'games') is a truly challenging application. We offer four arguments for this statement: (1) Games are widely available, thus subject to the scrutiny of hundreds of thousands of human players [Laird and van Lent, 2001; Sawyer, 2002]; (2) Games reflect the real world, and thus game AI may capture features of real-world behaviour [Sawyer, 2002; Graepel *et al.*, 2004]; (3) Games require human-like (realistic, believable) intelligence, and thus are ideally suited to pursue the fundamental goal of AI, i.e., to understand and develop systems with human-like capabilities [Laird and van Lent, 2001; Sawyer, 2002]; and (4) Games place highly-constricting requirements on implemented game AI solutions [Laird and van Lent, 2001; Nareyek, 2002; Charles and Livingstone, 2004; Spronck *et al.*, 2004b].

We define 'adaptive game AI' as game AI that employs unsupervised online learning ('online' meaning 'during gameplay'). Adaptive game AI has two main objectives, namely (1) to enhance the agents with the ability to learn from their mistakes, to avoid such mistakes in future play (self-correction), and (2) to enhance the agents with the ability to devise new behaviour in response to previously unconsidered situations, such as new tactics used by the human player (creativity). Although academic researchers have achieved successful results in their exploration of adaptive game AI in recent research (e.g., [Demasi and Cruz, 2002; Spronck *et al.*, 2004b; Graepel *et al.*, 2004]), game publishers are still reluctant to release games with online-learning capabilities [Funge, 2004]. Their main fear is that the agents learn inferior behaviour [Woodcock, 2002; Charles and Livingstone, 2004]. Therefore, the few games that contain online adaptation, only do so in a severely limited sense, in order to run as little risk as possible [Charles and Livingstone, 2004].

Regardless of the usefulness of adaptive game AI, to convince game publishers to allow it in a game, the *reliability* of the adaptive game AI should be guaranteed, even against human players that deliberately try to exploit the adaptation process to elicit inferior game AI. Reliability of adaptive game AI can be demonstrated by showing that it meets eight requirements [Spronck, 2005], which are discussed in Section 2. However, meeting the requirements is easier said than done, because they tend to be in conflict with each other.

In this paper, we propose a model for Reliable Adaptive Game Intelligence (RAGI). The purpose of the model is to provide a conceptual framework for the implementation of reliable adaptive game AI. The model makes explicit two concepts which, in our view, are necessary for the design of reliable adaptive game AI, namely a knowledge base, and an adaptive opponent model.

The outline of this paper is as follows. In Section 2 we discuss requirements for the creation of reliable adaptive game AI. In Section 3 we discuss domain knowledge and opponent models for adaptive game AI. The RAGI model is introduced in Section 4. In section 5 we argue that the proposed model is a suitable framework for implementing reliable adaptive game AI. Section 6 describes possible implementations of the model. Finally, Section 7 concludes and looks at future work.

## 2 Requirements for Reliability

We define 'reliable adaptive game AI' as adaptive game AI that meets the eight requirements for online learning of game AI specified by Spronck [2005], who indicated that adaptive game AI that meets these eight requirements will go a long way in convincing game publishers to adopt it. The eight requirements are divided into four computational requirements and four functional requirements. The computational requirements are necessities: failure of adaptive game AI to meet the computational requirements makes it useless in practice. The functional requirements are not so much necessities, as strong

preferences by game developers: failure of adaptive game AI to meet the functional requirements means that game developers will be unwilling to include it in their games, even when it yields good results (e.g., improves the effectiveness of agent behaviour) and meets all four computational requirements.

The four computational requirements are the following.

**Speed:** Adaptive game AI must be computationally fast, since learning takes place during game-play [Laird and van Lent, 2001; Nareyek, 2002; Charles and Livingstone, 2004; Funge, 2004].

**Effectiveness:** Adaptive game AI must be effective during the whole learning process, to avoid it becoming inferior to manually-designed game AI, thus diminishing the entertainment value for the human player [Charles and Livingstone, 2004; Funge, 2004]. Usually, the occasional occurrence of non-challenging game AI is permissible, since the player will attribute an occasional easy win to luck.

**Robustness:** Adaptive game AI has to be robust with respect to the randomness inherent in most games [Chan *et al.*, 2004; Funge, 2004].

**Efficiency:** Adaptive game AI must be efficient with respect to the number of trials needed to achieve successful game AI, since in a single game, only a limited number of occurrences happen of a particular situation which the adaptive game AI attempts to learn successful behaviour for. Note that the level of intelligence of the adaptive game AI determines how many trials can still be considered efficient adaptation; on an operational level of intelligence (as in the work by Graepel *et al.* [2004]), usually many more trials are available for learning than on a tactical or strategic level of intelligence (as in the work by Spronck *et al.* [2004b] and the work by Ponsen *et al.* [2005]).

The four functional requirements are the following.

**Clarity:** Adaptive game AI must produce easily interpretable results, because game developers distrust learning techniques of which the results are hard to understand.

**Variety:** Adaptive game AI must produce a variety of different behaviours, because agents that exhibit predictable behaviour are less entertaining than agents that exhibit unpredictable behaviour.

**Consistency:** The average number of trials needed for adaptive game AI to produce successful results should have a high consistency, i.e., a low variance, to ensure that it is rare that learning in a game takes exceptionally long.

**Scalability:** Adaptive game AI must be able to scale the effectiveness of its results to match the playing skills of the human player [Lidén, 2004]. This last functional requirement may be considered optional: without it, adaptive game AI aims to be as strong as possible; with it, adaptive game AI aims to be an appropriate match for the human player.

We observe that there are conflicts between several of these requirements. For instance, the requirements of speed and efficiency are in conflict with the requirements of robustness and consistency, because in a non-deterministic learning environment, robustness and consistency are typically acquired by always basing the learning on several repetitions of each test, which is costly in computation time and required number of trials. Also, the requirement of effectiveness is in conflict with the requirement of variety, because, in general, enforced variations on game AI make it less effective.

The core problem for online learning, especially in a non-deterministic, complex environment, is finding the right balance between exploitation and exploration [Carmel and Markovitch, 1997]. During exploitation, adaptive game AI does not learn, but deploys its learned knowledge to elicit successful agent behaviour in the game. During exploration, adaptive game AI attempts to learn new behaviour. If there is insufficient exploration, the adaptive game AI learns slowly, and may remain stuck in a local or even a false optimum, and thus fails to meet the requirement of efficiency. If there is too much exploration, the adaptive game AI will often generate inferior agent behaviour, and thus fails to meet the requirement of effectiveness. A possible solution for this issue is to automatically tune the amount of exploration to the observed results of the agent behaviour: good results require a low degree of exploration, while unsatisfying results require a higher degree of exploration. However, note that due to the non-determinism of most game environments, unsatisfying results may be the effect of a string of chance runs, in which case these results preferably should not lead to a higher degree of exploration [Spronck, 2005].

## 3 Necessary Concepts for Adaptive Game AI

In the few published games that contain online adaptation, changes made by the adaptive game AI are almost always limited to updating a small number of in-game parameters, such as the agents' strength and health. In the rare cases where a published game allows an agent's behaviour to be influenced, it is either through supervised learning (i.e., the human player actively training the agent to exhibit certain behaviour, as in BLACK & WHITE [Evans, 2001]), or through choosing between a few pre-programmed behaviours, such as different formations of enemy groups. Most academics will hesitate to call this 'adaptive game AI', since the agents do not design new behaviour autonomously (professional game developers might disagree, but they interpret the term 'artificial intelligence' much broader than academics [Tomlinson, 2003]).

In academic research of adaptive game AI, it is typically implemented as a direct feedback loop (cf., [Demasi and Cruz, 2002; Bakkes, 2003; Graepel *et al.*, 2004; Spronck *et al.*, 2004b]). In a direct feedback loop for agent control in a game (illustrated in Figure 1), the agent interacts with a game world. The agent's actions are determined by game AI. The agent feeds the game AI with data on its current situation, and with the observed results of its actions. The game AI adapts by processing the observed results, and generates actions in response to the agent's current situation.

Adaptive game AI is necessarily based on two concepts.

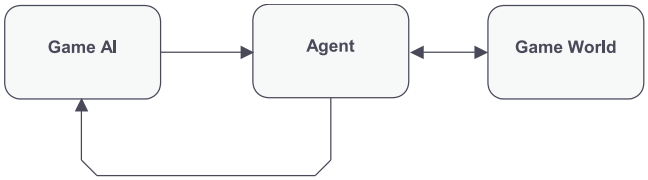


Figure 1: Game AI feedback loop.

The first concept is *domain knowledge* of the game environment. The reasoning behind this concept is that, to meet the four computational requirements, adaptive game AI must be of 'high performance'. According to Michalewicz and Fogel [2000], the two main factors of importance when attempting to achieve high performance for a learning mechanism are the exclusion of randomness and the addition of domain-specific knowledge. Since randomness is inherent in most games, it cannot be excluded. Therefore, it is imperative that the learning process is based on domain-specific knowledge [Manslow, 2002].

The second concept is an *opponent model*. The task of an opponent model is to understand and mimic the opponent's behaviour, to assist the game AI in choosing successful actions against this opponent. Without an opponent model, the game AI is unable to adapt adequately to human player behaviour.

The opponent model can be either *explicit* or *implicit*. An opponent model is explicit in game AI when a specification of the opponent's attributes exists separately from the decision-making process. An opponent model is implicit in game AI when the game AI is fine-tuned to a specific (type of) opponent, without the game AI actually referring that opponent's attributes [van den Herik *et al.*, 2005]. With an implicit opponent model, the adaptive game AI basically is a process that updates its opponent model by improving its decision making capabilities against particular human-player behaviour.

In most, if not all published research on adaptive game AI, the opponent model is implicit. However, in the comparable research field of adaptive multi-agent systems, Carmel and Markovitch [1997] have shown that adaptive agents that use an explicit opponent model are more effective than adaptive agents that use an implicit opponent model. Furthermore, the use of explicit opponent models is considered a necessary requirement for successful game-play in the research of such classical games as ROSHAMBO [Egnor, 2000] and POKER [Billings *et al.*, 2000], which have many features in common with modern commercial games. Therefore, we feel that there are sufficient reasons to suppose that an explicit opponent model is highly desired for adaptive game AI.

Figure 2 presents the feedback loop of Figure 1, enhanced with a data store of domain knowledge and an explicit opponent model. Examples are given, derived from a Computer RolePlaying Game (CRPG), of (1) a piece of domain knowledge, (2) an attribute of the opponent model, and (3) a rule of the game AI which takes the domain knowledge and opponent model into account. Note that, by simply removing the explicit opponent model from the figure, the opponent model becomes implicit in the game AI. Under the condition that we know that the game AI is effective, we can derive the explicit opponent model to a great extent by analysing the game AI.

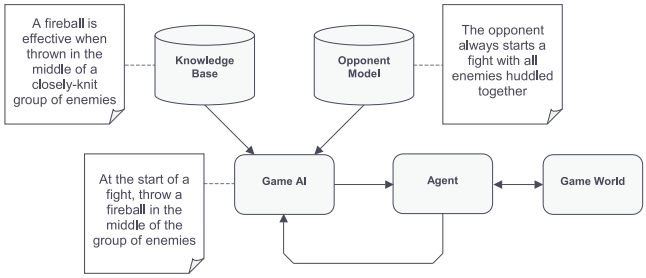


Figure 2: Game AI with domain knowledge and an explicit opponent model.

Typically, opponent models of human players are not implemented statically, but are learned from observed behaviour [Carmel and Markovitch, 1997]. In contrast, domain knowledge for game AI is typically manually designed by game developers, usually by programming static game AI. However, as Ponsen *et al.* [2005] show, it is possible to generate domain knowledge automatically from game-play data.

In conclusion, we propose that successful adaptive game AI should incorporate a knowledge base of domain knowledge, and an adaptive opponent model of the human player (preferably explicit).

## 4 The Model

In this section we present our model for Reliable Adaptive Game Intelligence (RAGI). The RAGI model is illustrated in Figure 3. It is described below.

Basically, the RAGI model implements a feedback loop, as represented in Figure 1. The two differences between the feedback loop of Figure 1, and the RAGI model of Figure 3, are that (1) the RAGI model extends the feedback loop with explicit processing of observations distinguished from the game AI, and (2) the RAGI model also allows the use of game world attributes which are not directly observed by the agent (e.g., observations concerning different agents).

The RAGI model collects agent observations and game world observations, and extracts from those a 'case base'. The case base contains all observations relevant for the adaptive game AI, without redundancies, time-stamped, and structured in a standard format for easy access. A case consists of a description of a game-play situation, comprising selected features and actions undertaken by agents in that situation. All cases in the case base contain an identification of the particular agents involved, whether controlled by the computer or by a human player. In the case of multi-player games, we may expect the case base to expand rather fast. In the case of single-player games, the case base will probably expand slowly. Consequently, the RAGI model is most applicable to multi-player games, although under certain conditions it may be applicable to single-player games, too.

The case base has two uses. The first use is to build an opponent model. The second use is to generate domain knowledge.

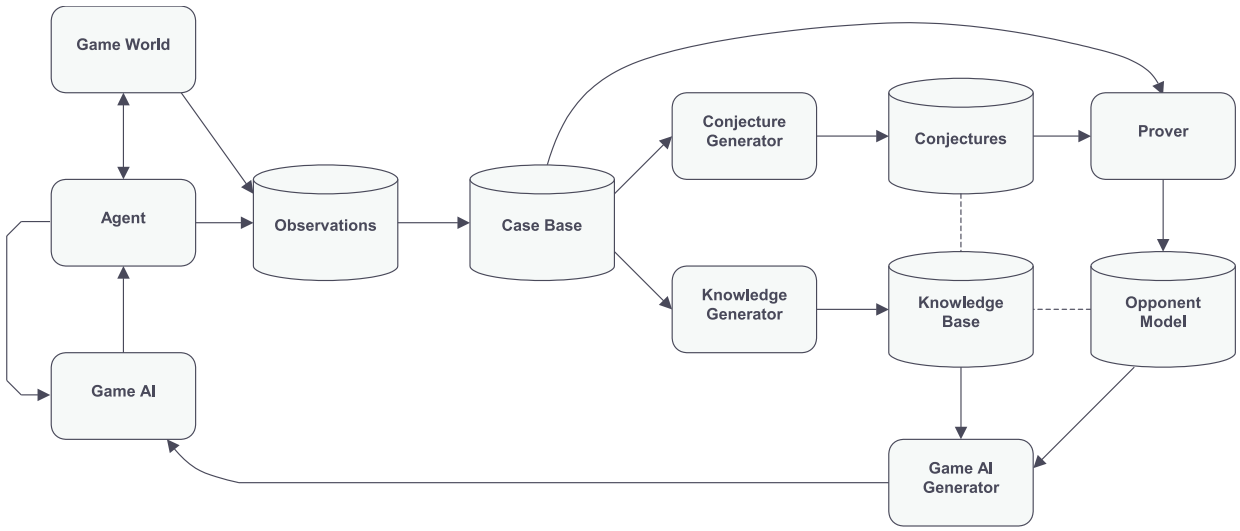


Figure 3: The RAGI model.

To build an opponent model, a 'conjecture generator' creates conjectures (i.e., statements and observations) on the way human players interact with the game world and with computer-controlled agents. These conjectures may not be generally applicable for all human players. However, a 'prover' (not to be confused with a theorem prover) selects those conjectures that might be of interest in building an opponent model of a specific human player, and uses the case base to attach a degree of confidence to the selected conjectures in this respect. Conjectures with a sufficiently high degree of confidence are stored as the opponent model of the human player. The representation of the conjectures depends on the application: for example, it might be in the form of first-order logic, or simply in the form of a collection of values for certain variables.

To generate domain knowledge, a 'knowledge generator', which can be considered a data mining process, analyses the case base and extracts relevant statements and rules. These are stored in a knowledge base. As with the opponent model, the case base is used to attach a degree of confidence to each statement in the knowledge base.

The opponent model and the knowledge base are used by a 'game AI generator' to create new game AI. Depending on the contents of the knowledge base, the game AI generator can be used to imitate the play of successful agents (for instance, those that are controlled by expert human players), or to design completely new tactics and strategies.

Through changes in the case base, changes might be caused in the opponent model and/or the knowledge base, which will automatically generate new game AI. For instance, if the human player changes his behaviour, the prover may assign a lower confidence to certain statements in the opponent model of this human player, which will influence the game AI generator to update the game AI.

Usually, there are connections between 'conjectures' and the 'knowledge base'. For instance, a conjecture might state that the human player has a preference for certain actions, while the knowledge base specifies a good defence against these actions. It is a good idea for implementations of the RAGI model to make these connections explicit. In Figure 3, this is represented by a dotted line between the conjectures and the knowledge base. Since the opponent model consists of a subset of the conjectures (enhanced with a degree of confidence), the same connections exist between the opponent model and the knowledge base.

## 5 Reliability

Why do we expect the RAGI model to be a good starting point for the creation of reliable adaptive game AI?

Besides the fact that the RAGI model encompasses an explicit opponent model and explicit domain knowledge, which we argued in Section 3 to be necessary for successful adaptive game AI, the RAGI model may meet the requirements specified in Section 2 as follows.

- The speed of the adaptive game AI relies, of course, on the speed of its components. In the past, authors have investigated speedy implementations of several of the components (e.g., for the knowledge generator [Ponsen *et al.*, 2005], and for the game AI generator [Spronck *et al.*, 2004b]). However, even if some components require too much processing time, since the model uses a case base the adaptive game AI may learn on a computer separate from the computer used to play the game, or in a separate thread, or on down-time of the game-playing computer (admittedly, in the last case this would amount to offline learning). This may allow the RAGI model to meet the requirement of speed, even when the processing itself is computationally intensive.
- Inferior behaviour on the part of any agent will automatically be translated into instances in the case base, that are processed into the opponent model or the knowledge base, to generate new game AI. This allows the RAGI model to meet the requirement of effectiveness.

- A lower limit to the required degree of confidence can be set so that the quality of the domain knowledge and of the opponent model is at an appropriate level. This allows the RAGI model to meet the requirement of robustness.
- The adaptive game AI does not learn only from experiences of the agent it controls, but also from the experiences of all other agents in the game world, whether controlled by the human or by the computer. It is even possible, in the case of single-player games, to collect cases from games played on different computers through the internet. Therefore, for the RAGI model the requirement of efficiency is simply not an issue.
- The use of an explicit opponent model and explicit domain knowledge helps the RAGI model in meeting the requirement of clarity.
- By varying over the domain knowledge used, the RAGI model meets the requirement of variety.
- Since the case base can be shared between all players of a game (whether in single-player or multi-player mode), all instances of the adaptive game AI learn at the same rate. This allows the RAGI model to meet the requirement of consistency.
- By using statements in the opponent model or the knowledge base with a lower confidence, or by excluding high-confidence domain knowledge, the generated game AI may function at an arbitrary level of skill [Spronck *et al.*, 2004a]. This allows the RAGI model to meet the requirement of scalability.

Depending on the implementation of the various processes, arguably the RAGI model may be too complex, and thus too computationally intensive, to be used for online learning. This issue holds in particular for single-player games, when only a single computer is available. It has less impact on multi-player games, where the case base is preferably situated on the game server, and domain knowledge and conjectures are generated centrally, so that they can be shared amongst all players. On the client computer, at maximum only two processes need to be executed, namely (1) the maintenance of the opponent model of the human player that uses the computer, and (2) the generation of new game AI on the basis of the opponent model and the centralised knowledge base. In general, opponent models do not change quickly. Furthermore, if connections between the conjectures and the domain knowledge (i.e., the dotted lines in Figure 3) are maintained centrally, the generation of new game AI can be fast.

## 6 Implementations

When implementing adaptive game AI according to the RAGI model, many findings of previous research can be incorporated. For instance, Spronck *et al.* [2004b] designed 'dynamic scripting', an adaptive-game-AI technique that makes use of a rulebase, which is equivalent to a knowledge base with domain knowledge. Ponsen *et al.* [2005]) investigated the automatic generation of domain knowledge for adaptive game AI, i.e., a knowledge generator. There is plenty of research available on the generation of opponent models (cf., [Fürnkranz, 1996; Carmel and Markovitch, 1997; Davison and Hirsh, 1998; Billings *et al.*, 2000; Egnor, 2000]), even in the area of commercial games (cf., [Alexander, 2002; McGlinchey, 2003]).

An interesting aspect of the RAGI model is that it can be implemented in stages. An easy implementation would use a static data store of manually-designed conjectures, and a static knowledge base of manually-designed knowledge, with the connections between the conjectures and the knowledge also programmed manually. Only the 'prover' would need to use the case base to constitute the opponent model, by selecting the conjectures that are most likely to be true. Depending on how the knowledge is formulated, the game AI generator would be trivial, because it only would need to select from the knowledge that is connected with the opponent model.

At a moderate level of difficulty for the implementation of the model, the connections between the conjectures and the knowledge base could be generated automatically. And at a high level of difficulty, a knowledge generator and conjecture generator could be implemented.

The possibility to start an implementation of the RAGI model at an easy level, gradually expanding it to become more complex, makes the model ideal for explorative research. The RAGI model can also be combined easily with the TIELT architecture [Aha and Molineaux, 2004], since TIELT has been designed to work with a task model (i.e., game AI), a player model (i.e., an opponent model), and a game model (i.e., a case base).

## 7 Conclusions and Future Work

In this paper we argued that reliable adaptive game AI needs to meet eight requirements, namely the requirements of (1) speed, (2) effectiveness, (3) robustness, (4) efficiency, (5) clarity, (6) variety, (7) consistency, and (8) scalability. Furthermore, we argued that successful adaptive game AI is necessarily based on domain knowledge and on an adaptive opponent model. We proposed a model for Reliable Adaptive Game Intelligence (RAGI), that is indeed based on domain knowledge and an explicit adaptive opponent model, and that may meet the eight specified requirements (at least for multi-player games).

Of course, the RAGI model must still undergo the proof of the pudding. In future work, we intend to structure our research into adaptive game AI around the RAGI model, and to explore to what extent elements of the model can learn while the adaptive game AI as a whole remains a reliable process.

## Acknowledgments

This research is supported by a grant from the Dutch Organisation for Scientific Research (NWO grant No. 612.066.406).

## References

[Aha and Molineaux, 2004] D.W. Aha and M. Molineaux. Integrating learning in interactive gaming simulators. In D. Fu, S. Henke, and J. Orkin, editors, *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence*, pages 49–53, 2004. AAAI Press.

[Alexander, 2002] T. Alexander. GoCap: Game observation capture. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 579–585, 2002. Charles River Media, Inc.

[Bakkes, 2003] S. Bakkes. *Learning to Play as a Team: Designing an Adaptive Mechanism for Team-Oriented Artificial Intelligence*. M.Sc. thesis. Universiteit Maastricht, Maastricht, The Netherlands, 2003.

[Billings *et al.*, 2000] D. Billings, A. Davidson, J. Schaeffer, and S. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1–2):201–240, 2000.

[Carmel and Markovitch, 1997] D. Carmel and S. Markovitch. Exploration and adaptation in multiagent systems: A model-based approach. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 606–611, 1997. Morgan Kaufmann.

[Chan *et al.*, 2004] B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan. Evolutionary behavior testing of commercial computer games. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 125–132, 2004. IEEE Press.

[Charles and Livingstone, 2004] D. Charles and D. Livingstone. AI: The missing link in game interface design. In M. Rauterberg, editor, *Entertainment Computing – ICEC 2004*, Lecture Notes in Computer Science 3166, pages 351–354, 2004. Springer-Verlag.

[Davison and Hirsh, 1998] B.D. Davison and H. Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time-Series Problems*, pages 5–12, 1998. AAAI Press. Proceedings of AAAI-98/ICML-98 Workshop, published as Technical Report WS-98-07.

[Demasi and Cruz, 2002] P. Demasi and A.J. de O. Cruz. Online coevolution for action games. *International Journal of Intelligent Games and Simulation*, 2(2):80–88, 2002.

[Egnor, 2000] D. Egnor. Iocaine powder. *ICGA Journal*, 23(1):33–35, 2000.

[Evans, 2001] R. Evans. The future of game AI: A personal view. *Game Developer Magazine*, 8(8):46–49, 2001.

[Funge, 2004] J.D. Funge. *Artificial Intelligence for Computer Games*. A K Peters, Ltd., Wellesley, MA, 2004.

[Fürnkranz, 1996] J. Fürnkranz. Machine learning in computer chess: The next generation. *ICCA Journal*, 19(3):147–161, 1996.

[Graepel *et al.*, 2004] T. Graepel, R. Herbrich, and J. Gold. Learning to fight. In Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass, editors, *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)*, pages 193–200, 2004. University of Wolverhampton.

[Laird and van Lent, 2001] J.E. Laird and M. van Lent. Human-level's AI killer application: Interactive computer games. *Artificial Intelligence Magazine*, 22(2):15–26, 2001.

[Lidén, 2004] L. Lidén. Artificial stupidity: The art of making intentional mistakes. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 41–48, 2004. Charles River Media, Inc.

[Manslow, 2002] J. Manslow. Learning and adaptation. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 557–566, 2002. Charles River Media, Inc.

[McGlinchey, 2003] S.J. McGlinchey. Learning of AI players from game observation data. In Q. Mehdi, N. Gough, and S. Natkin, editors, *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)*, pages 106–110, 2003. EUROSIS.

[Michalewicz and Fogel, 2000] Z. Michalewicz and D.B. Fogel. *How To Solve It: Modern Heuristics*. Springer-Verlag, Berlin, Germany, 2000.

[Nareyek, 2002] A. Nareyek. Intelligent agents for computer games. In T.A. Marsland and I. Frank, editors, *Computers and Games, Second International Conference, CG 2000*, volume 2063 of *Lecture Notes in Computer Science*, pages 414–422, 2002. Springer-Verlag.

[Ponsen *et al.*, 2005] M.J.V. Ponsen, H. Muñoz-Avila, P. Spronck, and D.W. Aha. Acquiring adaptive real-time strategy game opponents using evolutionary learning. In *Proceedings of the IAAI-05*, 2005. Accepted for publication.

[Sawyer, 2002] B. Sawyer. *Serious Games: Improving Public Policy through Game-based Learning and Simulation*. Foresight & Governance Project, Woodrow Wilson International Center for Scolars, Washington, DC, 2002.

[Spronck *et al.*, 2004a] P.H.M. Spronck, I.G. Sprinkhuizen-Kuyper, and E.O. Postma. Difficulty scaling of game AI. In A. El Rhalibi and D. Van Welden, editors, *GAME-ON 2004 5th International Conference on Intelligent Games and Simulation*, pages 33–37, 2004. EUROSIS.

[Spronck *et al.*, 2004b] P.H.M. Spronck, I.G. Sprinkhuizen-Kuyper, and E.O. Postma. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation*, 3(1):45–53, 2004.

[Spronck, 2005] Pieter Spronck. *Adaptive Game AI*. Ph.D. thesis, Universiteit Maastricht. Universitaire Pers Maastricht, Maastricht, The Netherlands, 2005.

[Tomlinson, 2003] S.L. Tomlinson. Working at thinking about playing or a year in the life of a games AI programmer. In Q. Mehdi, N. Gough, and S. Natkin, editors, *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)*, pages 5–12, 2003. EUROSIS.

[van den Herik *et al.*, 2005] H.J. van den Herik, H.H.L.M. Donkers, and P.H.M. Spronck. Opponent modelling and commercial games. In G. Kendall and S. Lucas, editors, *IEEE 2005 Symposium on Computational Intelligence and Games*, pages 15–25, 2005.

[Woodcock, 2002] S. Woodcock. AI roundtable moderator's report, 2002. www.gameai.com/cgdc02notes.html.

# Knowledge-Intensive Similarity-based Opponent Modeling

**Timo Steffens**
Institute of Cognitive Science
Kolpingstr. 7, 49074 Osnabrueck, Germany
tsteffen@uos.de

## Abstract

Similarity-based opponent modeling predicts the actions of an agent based on previous observations about the agent's behavior. The assumption is that the opponent maps similar situations to similar actions. Since agents are likely to use domain knowledge to reason about situations, it is reasonable to incorporate domain knowledge into the modeling system. We propose a knowledge-intensive similarity-based approach for opponent modeling in multi-agent systems. The classification accuracy is increased by adding domain knowledge to the similarity measure. The main contribution is to propose a hierarchy of knowledge types that can be incorporated into similarity measures. The approach has been implemented and evaluated in the domain of simulated soccer.

## 1 Introduction

An important factor effecting the behavior of agents in multi-agent systems (MAS) is the knowledge which they have about each other [Carmel and Markovich, 1996]. Predicting the actions of other agents has been shown to positively influence the behavior of an agent (e. g. [Denzinger and Hamdan, 2004]). However, in many MAS predicting agent actions is difficult due to the fact that many MAS (most prominently RoboCup [Kitano *et al.*, 1997], and also GameBots [Adobbati *et al.*, 2001]) are open systems where a variety of agents are encountered which are not known to an agent or its designers beforehand. In this paper we present an approach to opponent modeling which relies on the observation that the agents and their behavior are constrained by the properties of the environment. These properties are part of the domain knowledge and we show how such domain knowledge can be incorporated into similarity-based opponent modeling.

The remainder of this paper is organized as follows. The next section describes similarity-based opponent modelling and identifies issues that have to be coped with if the approach is applied to multi-agent systems. Section 3 introduces the domain of simulated soccer, a multi-agent system which serves as the evaluation domain. In section 4 we present our taxonomy of knowledge types and show how the knowledge can be incorporated into similarity measures. Experimental results are presented in section 5. Section 6 discusses related work, and finally the last section concludes.

## 2 Similarity-based Opponent Modeling

Case-based reasoning (CBR) is a common method for opponent modeling in MAS (e. g. [Ahmadi *et al.*, 2003; Wendler, 2004; Denzinger and Hamdan, 2004]). In a MAS several autonomous agents are active. From a CBR view, the classification goal is to predict the action of an opponent agent in a situation $S$. The CBR system compares $S$ to a case-base of previously observed situations, selects the situation $S'$ that is most similar to $S$, and returns the action in $S'$.

The classification- or prediction-accuracy of CBR depends on the quality of the similarity measure. Unfortunately, implementing a similarity measure for opponent modeling is not trivial due to a number of issues:

Context: Similarity is not an absolute quantity but depends on the context. The similarity must be adapted to the game situation and role of the agent whose action is to be predicted. Consider situations in a soccer game: The positions of team B's defenders will be rather irrelevant if the classification goal is the action of team A's goalie, but rather relevant if the classification goal is the action of team A's forwards. For these two classification goals, the similarity measure must weight attributes (i. e. player positions) differently.

Sparseness of data: Often, in CBR it is assumed that there is an abundance of data. However, in many opponent modeling domains this is not the case, since cases enter the case-base over time as observations are made. For example, in an open multi-agent system such as RoboCup [Kitano *et al.*, 1997], new opponents may be encountered without the possibility to gather data about them beforehand. Thus, a requirement for similarity-based opponent modeling is that it has to perform well with sparse data. Fortunately, the lack of knowledge in the case knowledge container can be compensated by moving additional knowledge into the similarity measure knowledge container [Richter, 1995]. In this paper we explore how different forms of knowledge can be incorporated into similarity measures.

Attribute matching: In most CBR applications, matching attributes is straight-forward, as equally named attributes or attributes at the same vector position are matched. But when applying CBR to opponent modeling in MAS, matching of attributes is not trivial. The agents of the two situations have to be matched in a situation-specific way, so that their properties (e. g., their positions and velocities) can be compared. This matching has to take the agents' roles and goals into account.

In this paper we tackle the first two issues by incorporating domain knowledge into the similarity measure.

## 3 An Example MAS: RoboCup

The RoboCup domain is a typical MAS where opponent modeling is crucial for successfully counteracting adversary agents [Kitano *et al.*, 1997]. Two teams of autonomous agents connect to a server and play simulated soccer against each other. Each player is an autonomous process. This is a challenge for opponent modeling, since the behavior of each opponent player has to be approximated individually.

Decision making is done in near real-time, that is, in discrete time steps. Every 100ms the agents can execute a primitive action and the world-state changes based on the actions of all players. Basically, the action primitives are *dash*, *turn*, *kick*, which must be combined in consecutive time steps in order to form high-level actions such as passes or marking. The agents act on incomplete and uncertain information: Their visual input consists of noisy information about objects in their limited field of vision. There is an additional privileged agent, the online coach, which receives noise-free and complete visual input of the playing field. The interface for online coaches has been included into the server for opponent modeling purposes [Chen *et al.*, 2001]. Every 100 ms it receives information about the position and velocity of all objects on the playing field (22 players and the ball). The agents' actions cannot be observed directly, but can be inferred from the differences between consecutive world-states. E. g., in our implementation the coach assumes that the player controlling the ball executed a kick, if the ball's velocity increases.

Cases for our CBR system are generated from the observations of the coach. A case is represented in two parts: 46 attributes (23 positions and 23 velocities) specifying the situation, and 22 attributes storing the actions. In a prediction task, only the situation is known and one of the actions serves as the classification goal; the other actions are ignored.

RoboCup is an ideal domain for evaluating our approach, because it is complex enough to use the various knowledge types we introduce in the next section.

## 4 Knowledge Types for Similarity Measures

This section discusses which types of knowledge can be incorporated into similarity measures for similarity-based opponent modeling. Knowledge types will be grouped into a hierarchy (see figure 1) based on the integration methods that are applicable to each type. Incorporation methods applicable to a type are also applicable to its subtypes. This way, several knowledge types that were previously researched in isolation in knowledge-intensive CBR can now be integrated into a common framework.

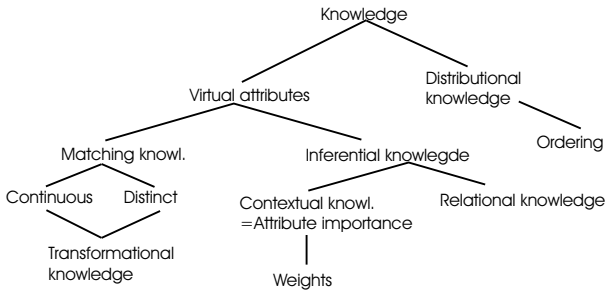


Figure 1: Hierarchy of knowledge types.

We group the types by the way they can be used and incorporated into similarity measures. It should be noted that we do not start from the knowledge types as proposed by knowledge representation work, such as frames, scripts, or semantic networks. These mechanisms are a mixture of the types that we discuss in the following.

For the examples, we use the following notation: $C_1, C_2, C_3 \in \mathbb{R}$ are continuous attributes. $D_1, D_2 \in \mathbb{Z}$ are discrete attributes. $P(x)$ is a binary concept applicable to instance $x$. $C_i(x)$ or $D_i(x)$ denote the value of instance $x$ for attribute $C_i$ or $D_i$. $w \in \mathbb{R}$ is a weight.

At the most general level, we distinguish *virtual attributes* [Richter, 2003] from *distributional knowledge*. The latter includes knowledge about the range and distribution of attributes and their values. Knowledge about the range of an attribute is commonly used to normalize the attribute similarity to [0,1]. Since this type of knowledge is widely used in CBR, we focus on the less researched type of knowledge that can be formalized as virtual attributes.

Virtual attributes are not directly represented in the cases but can be inferred from other attributes [Richter, 2003]. They are common in database research. In CBR, virtual attributes are useful if the similarity does not depend on the case attributes themselves, but on their relation to each other. For example, if $C_1$ is the position of player A and $C_2$ is the position of player B, then a virtual attribute $C_3(x) = C_1(x) - C_2(x)$ could be the distance between A and B.

We further distinguish between *inferential* and *matching knowledge*. Discrete matching knowledge states that two values of an attribute are equivalent. The famous PROTOS system made extensive use of this type of knowledge [Porter *et al.*, 1990]. Also taxonomies are instantiations of matching knowledge and were used in CBR [Bergmann, 1998]. Continuous matching knowledge defines intervals in an attribute stating that the exact value of the attribute in the interval is irrelevant. Examples: $C_1(x) > 30 \wedge C_1(x) < 50$ (continuous) and $D_1(x) \equiv D_1(y)$ (discrete). This can be easily formulated as virtual attribute, stating that an attribute is a member of the interval or is identical to one of the equivalent values.

Matching knowledge can be used to match syntactically different attributes that are semantically equivalent. For example, in our opponent modelling approach two different players will be treated as equivalent if they have the same

role (such as defender).

*Transformational knowledge* is a specialization of matching knowledge where geometric operations are used to map a point in the instance-space to another point. For example, transformational knowledge has been used to establish identity despite geometric rotation (e. g. [Schaaf, 1994]). Example: $C_1(x) = rotate(C_1(y), 30)$. In our RoboCup implementation, transformational knowledge is used to match local scenes from one wing to the other. That is, the similarity of two attributes is maximal if they can be transformed into each other or if they are identical.

*Inferential knowledge* specifies the value of an attribute that is inferrable from other attributes' values. This type of knowledge has been used in explanation-based CBR (e. g., [Aamodt, 1994]). Example: $P(x) \leftarrow C_1(x) > 30 \wedge C_1(x) < 50$ Note that the condition makes use of matching knowledge. An example from RoboCup is to define offside as virtual attribute over the player and ball positions.

*Contextual knowledge* is a special form of inferential knowledge. It states that some feature is important given some other features. Knowledge about contextual features has been shown to improve classification accuracy [Aha and Goldstone, 1992]. Example: $important(P(x)) \leftarrow C_1(x) > 30 \wedge C_1(x) < 50$ We use contextual knowledge to code that a team's defenders' positions are irrelevant if the team's forward controls the ball close to the opponent goal.

In our hierarchy, *weights* are a special form of contextual knowledge. They allow us to express the importance of a feature on a continuous scale. We can express feature weights in a global way $important(P(x), w) \leftarrow TRUE$, or in a local way $important(P(x), w) \leftarrow C_1(x) > 30 \wedge C_1(x) < 50$.

In other words, contextual knowledge and weights can be called "attribute importance" knowledge.

*Relations* are a subtype of inferential knowledge. The condition part uses at least two different attributes. Relational knowledge for similarity is prominent in computational modelling of human categorization [Medin *et al.*, 1993]. Example: $P(x) \leftarrow C_1(x) > C_2(x)$. Note that relations usually make use of matching knowledge in the condition part, as they define regions in which the relation holds. Relations are for example necessary to code wether a certain opponent is between the ball and the goal.

*Ordering of nominal feature values* is distributional knowledge. It establishes a dimension in the instance space. In [Surma, 1994] it was shown that knowledge of the ordering of discrete feature values can increase classification accuracy.

According to the knowledge container approach [Richter, 1995], knowledge in the similarity measure can be simulated by moving knowledge into the case representation. However, moving contextual knowledge into the case-representation is not straight-forward (if possible at all) and has to dynamically interact with the similarity measure, since it is decided at classification time which attributes are used.

# 5 Evaluation

The following experiments tested whether the prediction accuracy for player actions increases if the similarity measure is extended with domain knowledge. Both the standard (knowledge-poor) and extended (knowledge-rich) similarity measures are tested on the same case-base and test cases. The test domain is RoboCup. We used 21 publicly available logfiles of recorded games between 19 different teams. The logfiles were taken from different rounds of the RoboCup-2002 tournament, so that they have a mixture of weak and good teams. Logfiles contain the same data that the online coach (refer to section 3) would receive. For each game, the first 1000,2000,3000 or 4000 cycles of the match were recorded into the case-base. A complete game lasts 6000 time steps. The test cases were drawn from the remaining time steps at fixed intervals of 50 time steps. The classification goal was the action of the ball owner.

## 5.1 The similarity measures

The non-extended similarity measure is defined as follows:

$$\begin{aligned} &sim(S_1, S_2) = \qquad (1)\\ &\sum_{i=1}^{22} [\omega_i * \Delta(p(i, S_1), p(i, S_2)) + \\ &\quad \omega'_i * \Delta(v(i, S_1), v(i, S_2))] + \\ &\quad \omega_0 * \Delta(bp(S_1), bp(S_2)) + \omega'_0 * \Delta(bv(S_1), bv(S_2)) \end{aligned}$$

where $S_1$ and $S_2$ are the two situations in comparison, $p(i, S_j)$ and $v(i, S_j)$ are the position and velocity of player $i$ in situation $S_j$, respectively, $bp(S_j)$ and $bv(S_j)$ are the ball-position and ball-velocity in $S_j$, respectively. $\Delta(A, B)$ is the Euclidean distance between $A$ and $B$, and $\omega_k$ and $\omega'_k$ with $\sum_{k=0}^{22}(\omega_k + \omega'_k) = 1$ are weights for positions and velocities, respectively.

The non-extended similarity measure uses some distributional knowledge for normalizing positions and velocities.

In comparison, the extended similarity measure is defined as follows: In line with the well-known local-global principle [Bergmann, 2002], we compute the similarity between two situations as the weighted average aggregation of the attributes' local similarities:

$$sim(S_1, S_2) = \sum_{i=1}^{n}(\omega_i * s_i)$$

where $s_i$ are the local similarity values (i. e., $s_i$ is the similarity for attribute $i$), and the $\omega_i$ are the corresponding weights.

$$\begin{aligned} sim(S_1, S_2) = {} & \omega_1 * 1(role(S_1), role(S_2)) + \\ & \omega_2 * 1(region(S_1), region(S_2)) + \\ & \omega_3 * 1(offside(S_1), offside(S_2)) + \\ & \omega_4 * 1(pressing(S_1), pressing(S_2)) + \\ & \omega_5 * \sum_{i=1}^{22} 1(free(S_1, i), free(S_2, i)) + \\ & \omega_6 * 1(ahead(S_1), ahead(S_2)) + \\ & \omega_7 * \Delta(positions(S_1), positions(S_2)) + \\ & \omega_8 * \Delta(velocities(S_1), velocities(S_2)) \end{aligned} \qquad (2)$$

where $1(X, Y) = 1$ iff $X = Y$, and 0 otherwise.

The attributes $role$ and $region$ make use of matching knowledge as they define hyper-planes in the instance-space. $role(S) \in \{forward, defender, midfielder\}$ denotes the role of the ball owner. $region(S) \in \{inFrontOfGoal, penaltyArea, corner, wing, midfield\}$ denotes the region the ball is in. Note that no distinction is made between left and right wing, and between the four corners, which is achieved by transformational knowledge about mirroring and rotating.

The attributes $offside$, $pressing$, and $free$ make use of inferential knowledge. $offside(S)$ is a binary predicate that checks whether the situation is offside. $pressing(S)$ checks whether pressing is performed in the situation, that is, whether the opponent attacks the ball owner with two or more players. $free(S, i)$ checks whether player $i$ stands free (i. e., no player of the opponent is within a certain distance).

The predicate $ahead(S)$ makes use of relational knowledge. It denotes the number of opponents that are between the ball and the goal.

$positions(S)$ and $velocities(S)$ are examples for contextual knowledge. They denote the positions and velocities of the players that are relevant in the given situation. Relevance of a player is computed by its role and the role of the ball owner. If the ball owner is a forward, its own defenders and the opponent's forwards are deemed irrelevant. If the ball owner is a defender, its own forwards and the opponent's defenders are deemed irrelevant.

The attribute weights of both similarity measures are relearned for each game with RELIEF [Kira and Rendell, 1992], using the case-base as training data.

### 5.2 Focus on high-level actions

In a complex domain such as RoboCup it is infeasible to predict an agent's behavior in terms of primitive actions. For individual skills (e. g., dribbling), primitive actions are often combined by neural networks. These are trained using amounts of training instances that are typically one or two levels of magnitude greater than the amount of observations available for opponent modelling. Hence, it is infeasible to predict an agent's primitive actions. Rather, in our experiments we predicted the high-level action *shoot on goal*. We assume that for taking countermeasures it is sufficient to anticipate high-level actions within a certain time window. For example, if a defender knows that within the next 20 time steps an opponent will shoot on the goal, it can position itself accordingly (and maybe even inhibit the shot by doing so. Therefore, in our prediction experiments the agents do not use the predictive information. This way they do not interfere with the prediction accuracy.) For both the standard and the extended similarity measures, the prediction of an action is counted as correct if the action occurs within 20 time steps after the prediction.

### 5.3 Results

The mean prediction accuracy of both similarity measures are shown in figure 2. For small case-bases, the prediction accuracies are almost the same. The extended similarity measure predicts better than the standard similarity measure for medium case-base sizes. If more data is available, the standard measure outperforms the extended measure. However, opponent modeling in open multi-agent systems does not have plenty of data, that is, the system can not wait until late in the game. Thus, a method that performs well with few and medium amounts of data should be preferred.

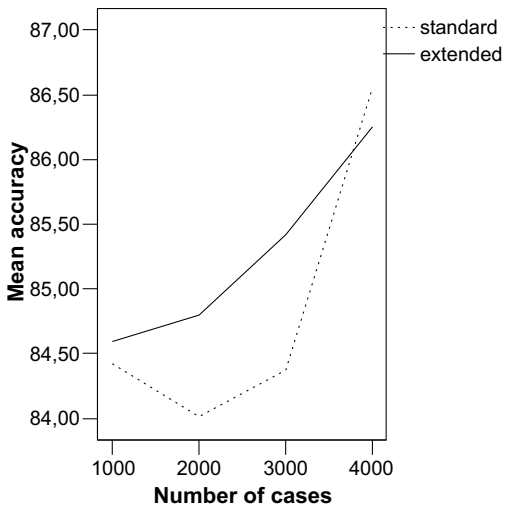


Figure 2: Mean accuracies of the standard and the extended similarity measures for different sizes of the case-base.

The accuracy difference between the two measures is small. Our analysis suggest that the small impact of the extended similarity measure is due to the fact that any increase of prediction accuracy is difficult, since the behaviors of the player agents are implemented as many different methods. Player implementations range from simple decision trees [Buttinger *et al.*, 2001], over probabilistic approaches [de Boer *et al.*, 2002] to neural networks [Riedmiller *et al.*, 2003]. Particularly if behaviors are learned, the partitioning of the situation space can be highly irregular and complex. Furthermore, it is very unlikely that the opponent players used the same domain knowledge. Hence, their situation space will be different from the situation space of our case-base. Considering this, we believe that the accuracy increase is substantial.

The most surprising result, though, is that the accuracy of the knowledge-poor (i. e. standard) similarity measure does not increase monotonically as the size of the case-base increases. We also observed that the accuracy of the knowledge-poor similarity measure did not always benefit from learning weights, that is, the unweighted measure performed as well or slightly better than the measure with learnt weights. While this needs more analysis, we believe that both observations can be explained by overfitting to the case-base.

## 6 Related Work

We already gave references to several knowledge-intensive CBR approaches in section 4. In addition to such approaches that incorporated domain knowledge for case retrieval, much effort has been done to use domain knowledge for adapting retrieved solutions (e. g. [Bergmann and Wilke, 1998; Wilke and Bergmann, 1996]). In knowledge-intensive CBR,

knowledge has typically not been regarded in terms of types, but rather each approach used knowledge in its own ad-hoc and domain-specific way (cf. [Aamodt, 2001]).

Explanation-based CBR (EBCBR) uses inference rules to create so-called explanations describing why a solution is appropriate for a given case [Cain *et al.*, 1991]. If an attribute was not used in the explanation, it is regarded as irrelevant and ignored in future similarity assessment. Another branch of EBCBR uses explanations to "explain away" differences that are either irrelevant or only syntactical rather than semantical [Aamodt, 1994]. Similarly, while not regarded as EBCBR, the famous PROTOS system [Porter *et al.*, 1990] uses domain-knowledge for matching syntactically different features, too. The main difference to our approach is that those methods process only attributes that are already explicitly represented in the cases and do not infer additional ones.

Generating additional features has been researched in constructive induction (CI) [Matheus, 1991]. While CI focussed mainly on rule-based expert systems, there are also CI approaches for instance-based learning showing that learning additional features can improve classification accuracy [Aha, 1991] even if the knowledge is vague [Steffens, 2004].

Apart from the similarity-based approach, there are several different approaches to opponent modeling. However, not all of them are well-suited for a continuous, real-time multi-agent game such as RoboCup.

In game theory there are approaches to learn opponent models from action sequences [Carmel and Markovich, 1996]. Usually a payoff-matrix is necessary, but for predicting the opponent's actions this requirement does not hold [Rogowski, 2004]. Unfortunately, these learning techniques assume that the opponent strategy can be described by a deterministic finite automaton, which might not always be the case in a complex domain. Most importantly, game theory can describe game states only as a history of actions, which is infeasible in complex games such as RoboCup, where subsequent game states are not only determined by player actions but also by the game physics.

Predicting opponent actions can also be done via plan-recognition [Kaminka and Avrahami, 2004], but reactive agents cannot be modeled. When classifying opponents into classes and selecting appropriate counter strategies [Steffens, 2002; Riley and Veloso, 2000] domain knowledge in the form of pre-defined model libraries is a non-optional requirement. In contrast, CBR only requires a set of observations, adding domain knowledge is optional. However, up to now our approach does not provide an appropriate counter-action, but only predicts the opponent's actions.

An approach that avoids predicting the opponent's actions is to adapt probability weights of action rules by reinforcement [Spronck *et al.*, 2003]. Instead of choosing actions that counter-act the opponent's next move, the own behavior is adapted to the opponent based on rewards and penalties.

## 7 Conclusion

We presented a hierarchy of knowledge types that can be incorporated into similarity measures for opponent modeling. Knowledge types from knowledge-intensive CBR that were previously seen as unrelated have been integrated into a framework. An implementation for simulated soccer suggests that predictive accuracy can benefit from domain knowledge.

## References


[Aamodt, 1994] Agnar Aamodt. Explanation-driven case-based reasoning. In Stefan Wess, Klaus-Dieter Althoff, and Michael M. Richter, editors, *Topics in Case-Based Reasoning*, pages 274–288. Springer, 1994.

[Aamodt, 2001] Agnar Aamodt. Modeling the knowledge contents of CBR systems. In Agnar Aamodt, David Patterson, and Barry Smyth, editors, *Proc. of the Workshop Program at ICCBR 2001*, pages 32–37, 2001.

[Adobbati *et al.*, 2001] R. Adobbati, Andrew N. Marshall, Andrew Scholer, S. Tejada, Gal A. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3D virtual world test-bed for multi-agent research. In Tom Wagner, editor, *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, pages 47–52. ACM Press, 2001.

[Aha and Goldstone, 1992] David W. Aha and Robert L. Goldstone. Concept learning and flexible weighting. In *Proc. of the 14th Annual Conf. of the Cognitive Science Society*, pages 534–539, Hillsdale, New Jersey, 1992. Lawrence Erlbaum.

[Aha, 1991] David W. Aha. Incremental constructive induction: An instance-based approach. In Lawrence Birnbaum and Gregg Collins, editors, *Proceedings of the 8th Intern. Workshop on Machine Learning*, pages 117–121, Evanston, IL, 1991. Morgan Kaufmann.

[Ahmadi *et al.*, 2003] M. Ahmadi, A. K. Lamjiri, M. M. Nevisi, J. Habibi, and K. Badie. Using two-layered case-based reasoning for prediction in soccer coach. In Hamid R. Arabnia and Elena B. Kozerenko, editors, *Proc. Intern. Conf. of Machine Learning; Models, Technologies and Applications*, pages 181–185. CSREA Press, 2003.

[Bergmann and Wilke, 1998] Ralph Bergmann and Wolfgang Wilke. Towards a new formal model of transformational adaptation in case-based reasoning. In Henri Prade, editor, *ECAI98, Thirteenth European Conference on Artificial Intelligence*, pages 53–57. John Wiley and Sons, Chichester, 1998.

[Bergmann, 1998] Ralph Bergmann. On the use of taxonomies for representing case features and local similarity measures. In Lothar Gierl and Mario Lenz, editors, *Proceedings of the Sixth German Workshop on CBR*, pages 23–32, 1998.

[Bergmann, 2002] Ralph Bergmann. *Experience Management*. Springer, Berlin, 2002.

[Buttinger *et al.*, 2001] Sean Buttinger, Marco Diedrich, Leonhard Hennig, Angelika Hoenemann, Philipp Huegelmeyer, Andreas Nie, Andres Pegam, Collin Rogowski, Claus Rollinger, Timo Steffens, and Wilfried Teiken. Orca project report. Technical report, University of Osnabrueck, 2001.

[Cain *et al.*, 1991] Timothy Cain, Michael J. Pazzani, and Glenn Silverstein. Using domain knowledge to influence similarity judgements. In *Proceedings of the Case-Based Reasoning Workshop*, pages 191–198, Washington D.C., U.S.A., 1991.

[Carmel and Markovich, 1996] David Carmel and Shaul Markovich. Learning models of intelligent agents. In Howard Shrobe and Ted Senator, editors, *Proceedings of AAAI-96 and the 8th IAAI, Vol. 2*, pages 62–67, Menlo Park, California, 1996. AAAI Press.

[Chen *et al.*, 2001] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Patrick Riley, Timo Steffens, Yi Wang, and Xiang Yin. Soccerserver manual v7, 2001.

[de Boer *et al.*, 2002] Remco de Boer, Jelle Kok, and Frans C. A. Groen. Uva trilearn 2001 team description. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V. Lecture Notes in Computer Science 2377*, pages 551–554, Berlin, 2002. Springer.

[Denzinger and Hamdan, 2004] Joerg Denzinger and Jasmine Hamdan. Improving modeling of other agents using stereotypes and compactification of observations. In *Proceedings of AAMAS 2004*, pages 1414–1415. IEEE Computer Society, 2004.

[Kaminka and Avrahami, 2004] Gal A. Kaminka and Dorit Avrahami. Symbolic behavior-recognition. In Mathias Bauer, Piotr Gmytrasiewicz, Gal A. Kaminka, and David V. Pynadath, editors, *AAMAS-Workshop on Modeling Other Agents from Observations*, pages 73–80, 2004.

[Kira and Rendell, 1992] Kenji Kira and Larry A. Rendell. A practical approach to feature selection. In Derek H. Sleeman and Peter Edwards, editors, *Proceedings of the Ninth International Workshop on Machine Learning*, pages 249–256. Morgan Kaufmann Publishers Inc., 1992.

[Kitano *et al.*, 1997] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The robocup synthetic agent challenge,97. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 24–29, San Francisco, CA, 1997. Morgan Kaufmann.

[Matheus, 1991] Christopher J. Matheus. The need for constructive induction. In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Proc. of the 8th Intern. Workshop on Machine Learning*, pages 173–177. Morgan Kaufmann, 1991.

[Medin *et al.*, 1993] Douglas L. Medin, Robert L. Goldstone, and Dedre Gentner. Respects for similarity. *Psychological Review*, 100(2):254–278, 1993.

[Porter *et al.*, 1990] Bruce W. Porter, Ray Bareiss, and Robert C. Holte. Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence*, 45(1-2):229–263, 1990.

[Richter, 1995] Michael Richter. The knowledge contained in similarity measures. Invited talk at ICCBR-95, 1995.

[Richter, 2003] Michael M. Richter. Fallbasiertes Schliessen. *Informatik Spektrum*, 3(26):180–190, 2003.

[Riedmiller *et al.*, 2003] Martin Riedmiller, Arthur Merke, W. Nowak, M. Nickschas, and Daniel Withopf. Brainstormers 2003 - team description. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *Pre-Proceedings of RoboCup 2003*, 2003.

[Riley and Veloso, 2000] Patrick Riley and Manuela Veloso. On behavior classification in adversarial environments. In Lynne E. Parker, George Bekey, and Jacob Barhen, editors, *Distributed Autonomous Robotic Systems 4*, pages 371–380. Springer, 2000.

[Rogowski, 2004] Collin Rogowski. Model-based opponent-modelling in domains beyond the prisoner's dilemma. In Mathias Bauer, Piotr Gmytrasiewicz, Gal A. Kaminka, and David V. Pynadath, editors, *Workshop on Modeling Other Agents from Observations at AAMAS 2004*, pages 41–48, 2004.

[Schaaf, 1994] Joerg W. Schaaf. Detecting gestalts in CAD-plans to be used as indices. In Angi Voss, editor, *FABEL - Similarity concepts and retrieval methods*, pages 73–84. GMD, Sankt Augustin, 1994.

[Spronck *et al.*, 2003] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. Online adaptation of game opponent AI in simulation and in practice. In Quasim Mehdi and Norman Gough, editors, *Proc. of the 4th Inter. Conf. on Intelligent Games and Simulation (GAME-ON'03)*, pages 93–100, Belgium, 2003. EUROSIS.

[Steffens, 2002] Timo Steffens. Feature-based declarative opponent-modelling in multi-agent systems. Master's thesis, Institute of Cognitive Science Osnabrueck, 2002.

[Steffens, 2004] Timo Steffens. Virtual attributes from imperfect domain theories. In Brian Lees, editor, *Proceedings of the 9th UK Workshop on Case-Based Reasoning at AI-2004*, pages 21–29, 2004.

[Surma, 1994] Jerzy Surma. Enhancing similarity measure with domain specific knowledge. In *Proceedings of the Second European Conference on Case-Based Reasoning*, pages 365–371, Paris, 1994. AcknoSoft Press.

[Wendler, 2004] Jan Wendler. Recognizing and predicting agent behavior with case-based reasoning. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII, Lecture Notes in Artificial Intelligence*, pages 729–738, Berlin, 2004. Springer.

[Wilke and Bergmann, 1996] Wolfgang Wilke and Ralph Bergmann. Adaptation with the INRECA system. In Angi Voss, editor, *Proceedings of the ECAI 96 Workshop: Adaptation in CBR*, 1996.

# Using Model-Based Reflection to Guide Reinforcement Learning

**Patrick Ulam**[1], **Ashok Goel**[1], **Joshua Jones**[1], **and William Murdock**[2]
1. College of Computing, Georgia Institute of Technology, Atlanta, USA 30332
2. IBM Watson Research Center, 19 Skyline Drive, Hawthorne, USA 10532
pulam, goel, jkj@cc.gatech.edu, murdockj@us.ibm.com

## Abstract

In model-based reflection, an agent contains a model of its own reasoning processes organized via the tasks the agents must accomplish and the knowledge and methods required to accomplish these tasks. Utilizing this self-model, as well as traces of execution, the agent is able to localize failures in its reasoning process and modify its knowledge and reasoning accordingly. We apply this technique to a reinforcement learning problem and show how model-based reflection can be used to locate the portions of the state space over which learning should occur. We describe an experimental investigation of model-based reflection and self-adaptation for an agent performing a specific task (defending a city) in a computer war strategy game called FreeCiv. Our results indicate that in the task examined, model-based reflection coupled with reinforcement learning enables the agent to learn the task with effectiveness matching that of hand coded agents and with speed exceeding that of non-augmented reinforcement learning.

## 1 Introduction

In model-based reflection/introspection, an agent is endowed with a self-model, i.e., a model of its own knowledge and reasoning. When the agent fails to accomplish a given task, the agent uses its self-model, possibly in conjunction with traces of its reasoning on the task, to assign blame for the failure(s) and modify its knowledge and reasoning accordingly. Such techniques have been used in domains ranging from game playing [B. Krulwich and Collins, 1992], to route planning [Fox and Leake, 1995; Stroulia and Goel, 1994; 1996], to assembly and disassembly planning [Murdock and Goel, 2001; 2003]. It has proved useful for learning new domain concepts [B. Krulwich and Collins, 1992], improving knowledge indexing [Fox and Leake, 1995], reorganizing domain knowledge and reconfiguring the task structure [Stroulia and Goel, 1994; 1996], and adapting and transferring the domain knowledge and the task structure to new problems [Murdock and Goel, 2001; 2003].

However, [Murdock and Goel, 2001; 2003] also showed in some cases model-based reflection can only *localize* the causes for its failures to specific portions of its task structure, but not necessarily *identify* the precise causes or the modifications needed to address them. They used reinforcement learning to complete the partial solutions generated by model-based reflection: first, the agent used its self-model to localize the needed modifications to specific portions of its task structure, and then it used Q-learning within the identified parts of the task structure to precisely identify the needed modifications.

In this paper, we evaluate the inverse hypothesis, viz., model-based reflection may be useful for focusing reinforcement learning. The learning space represented by combinations of all possible modifications to an agent's reasoning and knowledge can be extremely large for reinforcement learning to work efficiently. If, however, the agent's self-model *partitions* the learning space into much smaller subspaces and model-based reflection *localizes* the search to specific subspaces, then reinforcement learning can be expedient. We evaluate this hypothesis in the context of game playing in a highly complex, extremely large, non-deterministic, partially-observable environment. This paper extends our earlier work reported in [Ulam *et al.*, 2004] which used only model-based reflection.

## 2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique in which an agent learns through trial and error to maximize rewards received for taking particular actions in particular states over an extended period of time [Kaelbling *et al.*, 1996]. Given a set of environmental states $\mathcal{S}$, and a set of agent actions $\mathcal{A}$, the agent learns a policy, $\pi$, which maps the current state of the world $s \in \mathcal{S}$, to an action $a \in \mathcal{A}$, such that the sum of the reinforcement signals $r$ are maximized over a period of time. One popular technique is Q-Learning. In Q-Learning, the agent calculates Q-Values, the expected value of taking a particular action in a particular state. The Q-Learning update rule can be described as $\mathcal{Q}(s,a) = \mathcal{Q}(s,a) + \alpha(r + \gamma \max_{a'} \mathcal{Q}^*(s,a') - \mathcal{Q}(s,a))$, where $r$ is the reward received for taking the action, $\max_{a'} \mathcal{Q}^*(s,a')$ is the reward that would be received by taking the optimal action after that, $\alpha$ is a parameter to control the learning rate, and $\gamma$ is a parameter to control discounting.

### 2.1 Hierarchical Reinforcement Learning

Although reinforcement learning is very popular and has been successful in many domains (e.g., [Tesauro, 1994]), its use is limited in some domains because of the so-called *curse of dimensionality*: the exponential growth of the state space required to represent additional state variables. In many domains, this prevents the use of reinforcement learning without significant abstraction of the state space. To overcome this limitation, much research has investigated the use of hierarchical methods of reinforcement learning. There are many variants of hierarchical reinforcement learning most of which are rooted the the theory of Semi-Markov decision processes [Barto and Mahadevan, 2003]. Hierarchical reinforcement learning techniques such as MAXQ value decomposition [Dietterich, 1998] rely on domain knowledge in order to determine the hierarchy of tasks that must be accomplished by the agent, as does our approach. However, in our approach, the agent uses model-based reflection to determine the portion of the task structure over which the reward should be applied after task execution. Furthermore, many hierarchical methods focus on abstractions of temporally extended actions for the hierarchy [Sutton *et al.*, 1999]; our approach uses the hierarchy to delimit natural partitions in non-temporally extended tasks.

## 3 The FreeCiv Game

The domain for our experimental investigation is a popular computer war strategy game called FreeCiv. FreeCiv is a multiple-player game in which a player competes either against several software agents that come with the game or against other human players. Each player controls a civilization that becomes increasingly modern as the game progresses. As the game progresses, each player explores the world, learns more about it, and encounters other players. Each player can make alliances with other players, attack the other players, and defend their own assets from them. In the course of a game (that can take a few hours to play) each player makes a large number of decisions for his civilization ranging from when and where to build cities on the playing field, to what sort of infrastructure to build within the cities and between the civilizations' cities, to how to defend the civilization. FreeCiv provides a highly complex, extremely large, non-deterministic, partially-observable domain in which the agent must operate.

Due the highly complex nature of the FreeCiv game, our work so far has addressed only two separate tasks in the domain: *Locate-City* and *Defend-City*. Due to limitations of space, in this paper we describe only the *Defend-City* task. This task pertains to the defense of one of the agent's cities from enemy civilizations. This task is important to the creation of a general-purpose FreeCiv game playing agent in that the player's cities are the cornerstone in the player's civilization. This task is also common enough such that the agent must make numerous decisions concerning the proper defense of the city during the time span of a particular game.

## 4 Agent Model

We built a simple agent (that we describe below) for the *Defend-City* task. The agent was then modeled in a variant of a knowledge-based shell called REM [Murdock, 2001] using a version of a knowledge representation called Task-Method-Knowledge Language (TMKL). REM agents written in TMKL are divided into tasks, methods, and knowledge. A task is a unit of computation; a task specifies *what* is done by some computation. A method is another unit of computation; a method specifies *how* some computation is done. The knowledge portion of the model describes the different concepts and relations that tasks and methods in the model can use and affect as well as logical axioms and other inferencing knowledge involving those concepts and relations. Formally, a TMKL model consists of a tuple $(T, M, K)$ in which $T$ is a set of tasks, $M$ is a set of methods, and $K$ is a knowledge base. The representation of knowledge ($K$) in TMKL is done using using Loom, an off-the-shelf knowledge representation (KR) framework. Loom provides not only all of the KR capabilities found in typical AI planning system (the ability to assert logical atoms, to query whether a logical expression holds in the current state, etc.), but also an enormous variety of more advanced features (logical axioms, truth maintenance, multiple inheritance, etc.). In addition, Loom provides a top-level ontology for reflective knowledge. Through the use of a formal framework such as this, dependencies between the knowledge used by tasks as well as dependencies between tasks themselves can be described in such a way that an agent will be able to reason about the structure of the tasks. A thorough discussion of TMKL can be found in [Murdock and Goel, 1998].

Table 1 describes the functional model of the *Defend-City* task as used by model-based reflection. The overall *Defend-City* task is decomposed into two sub-tasks by the *Evaluate-then-Defend* method. These subtasks are the evaluation of the defense needs for a city and the building of a particular structure or unit at that city. One of the subtasks, *Evaluate-Defense-Needs*, can be further decomposed through the *Evaluate-Defense* method into two additional subtasks: a task to check internal factors in the city for defensive requirements and a task to check for factors external to the immediate vicinity of the city for defensive requirements. These subtasks are then implemented at the procedural level for execution as described below.

The *Defend-City* task is executed each turn that the agent is not building a defensive unit in a particular city in order to determine if production should be switched to a defensive unit. It is also executed each turn that a defensive unit has finished production in a particular city. The internal evaluation task utilizes knowledge concerning the current number of troops that are positioned in and around a particular city to determine if the city has an adequate number of defenders barring any extraneous circumstances. This is implemented as a relation in the form of the evaluation of the linear expression: $allies(r) + d \geq t$ where $allies(r)$ is the number of allies within radius $r$, $d$ is the number of defenders in the city and $t$ is a threshold value. The external evaluation of a city's defenses examines the area within a specified radius around a

Table 1: TMKL Model of Defend-City Task

| **TMKL Model of the Defend-City Task** | |
|---|---|
| **Task** | Defend-City |
| *by* | Evaluate-Then-Build |
| *makes* | City-Defended |
| **Method** *transitions:* | Evaluate-Then-Build |
| state: *s1* | Evaluate-Defense-Needs |
| *success* | s2 |
| state: *s2* | Build-Defense |
| *success* | success |
| *additional-result* | City-Defended, Unit-Built Wealth-Built |
| **Task** | Evaluate-Defense-Needs |
| *input* | External/Internal-Defense-Advice |
| *output* | Build-Order |
| *by* | UseDefenseAdviceProcedure |
| *makes* | DefenseCalculated |
| **Method** *transitions:* | Evaluate-Defense-Needs |
| state: *s1* | Evaluate-Internal |
| *success* | s2 |
| state: *s2* | Evaluate-External |
| *success* | success |
| *additional-result* | Citizens-Happy, Enemies-Accounted Allies-Accounted |
| **Task** | Evaluate-Internal |
| *input* | Defense-State-Info |
| *output* | Internal-Defense-Advice |
| *by* | InternalEvalProcedure |
| *makes* | Allies-Accounted, Citizens-Happy |
| **Task** | Evaluate-External |
| *input* | Defense-State-Info |
| *output* | External-Defense-Advice |
| *by* | ExternalEvalProcedure |
| *makes* | Enemies-Accounted |
| **Task** | Build-Defense |
| *input* | BuildOrder |
| *by* | BuildUnitWealthProcedure |
| *makes* | Unit-Built, Wealth-Built |

city for nearby enemy combat units. It utilizes the knowledge of the number of units, their distance from the city, and the number of units currently allocated to defend the city in order to provide an evaluation of the need for additional defense. This is also implemented as a relation in the form of the linear expression $enemies(r) + e_t \leq d$ where $enemies(r)$ is the number of enemies in radius $r$ of the city, $e_t$ is a threshold value, and $d$ is the number of defenders in the city. These tasks produce knowledge states in the form of defense recommendations that are then utilized by the task that builds the appropriate item at the city. The *Build-Defense* task utilizes the knowledge states generated by the evaluation subtasks, knowledge concerning the current status of the build queue, and the technology currently available to the agent to determine what should be built for a given iteration of the task. The Build Defense task will then proceed to build a defensive unit, either a warrior or a phalanx based on the technology level achieved by the agent at that particular point in the game, or wealth to keep the citizens of the city happy. The goal of the *Defend-City* task is to provide for the defense of a city for a certain number of years. The task is considered successful if the city has not been conquered by opponents by the end of this time span. If the enemy takes control of the city the task is considered a failure. In addition, if the city enters civil unrest, a state in which the city revolts because of unhappiness, the task is considered failed. Civil unrest is usually due the neglect of infrastructure in a particular city that can be partially alleviated by producing wealth instead of additional troops.

## 5 Experimental Setup

We compared four variations of the *Defend-City* agent to determine the effectiveness of model-based reflection in guiding reinforcement learning. These were a control agent, a pure model-based reflection agent, a pure reinforcement learning agent, and a reflection-guided RL agent. The agents are described in detail below.

Each experiment was composed of 100 trials and each trial was set to run for one hundred turns at the hardest difficulty level in FreeCiv against eight opponents on the smallest game map available. This was to ensure that the *Defend-City* task would be required by the agent. The same random seed was utilized in all the trials to ensure that the same map was used. The random seed selected did not fix the outcome of the combats, however. The *Defend-City* task is considered successful if the city neither revolted nor was defeated. If the task was successful no adaptation of the agent occurred. If the agent's city is conquered or the city's citizens revolt, the *Defend-City* task is considered failed. Execution of the task is halted and adaptation appropriate to the type of agent is initiated. The metrics measured in these trials include the number of successful trials in which the city was neither defeated nor did the city revolt. In addition, the number of attacks successfully defended per game was measured under the assumption that the more successful the agent in defending the city, the more attacks it will be able to successfully defend against. The final metric measured was the number of trials run between failures of the task. This was included as a means of determining how quickly the agent was able to learn the task and is included under the assumption that an agent with longer periods between task failures indicate that the task has been learned more effectively.

### 5.1 Control Agent

The control agent was set to follow the initial model of the *Defend-City* task and was not provided with any means of adaptation. The initial *Defend-City* model used in all agents executes the *Evaluate-External* only looking for enemy units one tile away from the city. The initial *Evaluate-Internal* task only looks for defending troops in the immediate vicinity of the city and if there are none will build a single defensive unit. The control agent will not change this behavior over the lifetime of the agent.

Table 2: State variables for RL Based Agents

| Pure RL State Variables | Additional State Variables | Associated Sub-Task |
|---|---|---|
| $\leq$ 1 Allies in City | | *Evaluate-Internal* |
| $\leq$ 3 Allies in City | | *Evaluate-Internal* |
| $\leq$ 6 Allies in City | | *Evaluate-Internal* |
| $\leq$ 1 Allies Nearby | | *Evaluate-Internal* |
| $\leq$ 2 Allies Nearby | | *Evaluate-Internal* |
| $\leq$ 4 Allies Nearby | | *Evaluate-Internal* |
| $\leq$ 1 Enemies Nearby | | *Evaluate-External* |
| $\leq$ 3 Enemies Nearby | | *Evaluate-External* |
| $\leq$ 6 Enemies Nearby | | *Evaluate-External* |
| | Internal Recommend | *Evaluate-Defense* |
| | External Recommend | *Evaluate-Defense* |

Table 3: Failure types used in the *Defend-City task*

| **Model Location** (task) | **Types of Failures** |
|---|---|
| Defend-City | *Unit-Build-Error, Wealth-Build-Error, Citizen-Unrest-Miseval, Defense-Present-Miseval, Proximity-Miseval, Threat-Level-Miseval, None* |
| Build-Defense | *Unit-Build-Error, Wealth-Build-Error, None* |
| Evaluate-Internal | *Citizen-Unrest-Miseval, Defense-Present-Miseval, None* |
| Evaluate-External | *Proximity-Miseval, Threat-Level-Miseval, None* |

### 5.2 Pure Model-Based Reflection Agent

The second agent was provided capabilities of adaption based purely on model-based reflection. Upon failure of the *Defend-City* task, the agent used an execution trace of the last twenty executions of the task, and in conjunction with the current model, it performed failure-driven model-based adaptation. The first step is the localization of the error through the use of feedback in the form of the type of failure, and the model of the failed task. Using the feedback, the model is analyzed to determine in which task the failure has occurred. For example, if the the *Defend-City* task fails due to citizen revolt the algorithm would take as input: the *Defend-City* model, the traces of the last twenty executions of the task, and feedback indicating that the failure was a result of a citizen revolt in the city. The failure localization algorithm would take the model as well as the feedback as input. As a city revolt is caused by unhappy citizens, this information can be utilized to help localize where in the model the failure may have occurred. This algorithm will go through the model, looking for methods or tasks that result in knowledge states concerning the citizens' happiness. It will first locate the method *Evaluate-Defense-Need* and find that this method should result in the assertion *Citizens-Happy*. It will continue searching the sub-tasks of this method in order to find if any sub-task makes the assertion *Citizens-Happy*. If not, then the error can be localized to the *Evaluate-Defense-Need* task and all sub-tasks below it. In this case, the *Evaluate-Internal* task makes the assertion *Citizens-Happy* and the failure can be localized to that particular task. An extensive discussion on failure localization in model-based reflection can be found in [Murdock, 2001]. Given the location in the model from which the failure is suspected to arise, the agent then analyzes the execution traces available to it to determine to the best of its ability what the type of error occurred in the task execution through the use of domain knowledge. For this agent, this is implemented through the use of a failure library containing common failure conditions found within the *Defend-City* task. An example of a failure library used in this task is shown in Table 3. If a failure has been determined to have occurred, it is then used to index into a library of adaptation strategies that will modify the task in the manner indicated by the library. These adaptations consist of small modifications to the subtasks in the defend city tasks, such as changing the *Evaluate-External* subtask to look for enemies slightly further away. This is a slight variation on fixed value production repair [Murdock, 2001], as instead of adding a special case for the failed task, the agent replaces the procedure with a slightly more general version. If multiple errors are found with this procedure, a single error is chosen stochastically so as to minimize the chance of over-adaptation of the agent.

### 5.3 Pure Reinforcement Learning Agent

The third agent used a pure reinforcement learning strategy for adaptation implemented via Q-Learning. The state space encoding used by this agent is a set of nine binary variables as seen in Table 2. This allows a state space of 512 distinct states. It should be noted, however, that not all states are reachable in practice. The set of actions available to the agent were: *Build Wealth, Build Military Unit*. The agent received a reward of -1 when the the *Defend-City* task failed and a reward of 0 otherwise. In all trials alpha was kept constant at 0.8 and gamma was set to 0.9.

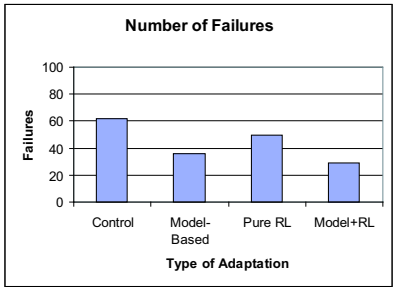


Figure 1: Number of Failures

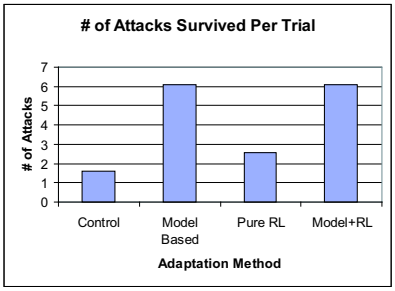


Figure 2: Average Attacks Resisted

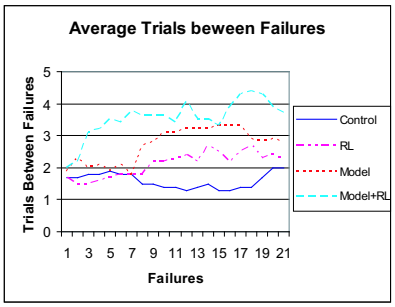


Figure 3: Average Number of Trials Between Failures

### 5.4 Reflection-Guided RL Agent

The final agent utilized model-based reflection in conjunction with reinforcement learning. The *Defend-City* task model was augmented with reinforcement learning by partitioning the state space utilized by the pure reinforcement learning agent into three distinct state spaces that are then associated with the appropriate sub-tasks of the *Defend-City* task. This essentially makes several smaller reinforcement learning problems. Table 2 shows the states that are associated with each sub-task. The *Evaluate-External* task is associated with three binary state variables. Its actions are the equivalent of the knowledge state produced via the *Evaluate-External* relation in the pure model-based agent, namely a binary value indicating if the evaluation procedure recommends that defensive units be built. In a similar manner, *Evaluate-Internal* is associated with six binary state variables as shown Table 2. The actions are also a binary value representing the relation used in the pure model-based agent. There are two additional state variables in this agent that are associated with the *Evaluate-Defenses* sub-task. The state space for this particular portion of the model are the outputs of the *Evaluate-External* and *Evaluate-Internal* tasks and is hence two binary variables. The actions for this RL task is also a binary value indicating a yes or no decision on whether defensive units should be built. It should be noted that while the actions of the individual sub-tasks are different from the pure reinforcement learning agent, the overall execution of the *Defend-City* task results in two possible actions for all agents, namely an order to build wealth or to build a defensive unit. Upon a failure in the task execution, the agent initiates reflection in a manner identical to the pure model-based reflection agent. Utilizing a trace of the last twenty executions of the *Defend-City* task as well as its internal model of the *Defend-City* task, the agent localizes the failure to a particular portion of the model as described in section 5.2. If an error in the task execution is detected, instead of utilizing adaptation libraries to modify the model of the task as in the pure model-based reflection agent, the agent applies a reward of -1 to the sub-task's reinforcement learner as indicated via reflection. The reward is used to update the Q-values of the sub-task via Q-Learning at which point the adaptation for that trial is over. If no error is found, then a reward of 0 is given to the appropriate reinforcement learner. In all trials alpha was kept constant at 0.8 and gamma was set to 0.9.

## 6 Results and Discussion

Figure 1 depicts the number of trials in which a failure occurred out of the one hundred trials run for each agent. The more successful adaptation methods should have a lower failure rate. As can be seen from the results, the reflection-guided RL agent proved most effective at learning the *Defend-City* task, with a success rate of around twice that of the control agent. The pure model-based reflection agent with the hand designed adaptation library proved to be successful also with a failure rate slightly higher then that of the reflection-guided RL agent. The pure RL agent's performance did not match either of the other two agents in this metric, indicating that most likely the agent had not had enough trials to successful learn the *Defend-City* task. The pure reinforcement learning agent's failure rate did improve over that of the control, however, indicating that some learning did take place, but not at the rate of either the pure model-based reflection agent or the reflection-guided RL agent.

The second metric measured was the number of attacks successfully defended by the agent in its city. This serves as another means of determining how effectively the agent has been able to perform the *Defend-City* task. The more attacks that the agent was able to defend, the more successfully the agent had learned to perform the task. The results from this metric can be seen in Figure 2. Both the pure model-based reflection and reflection-guided RL agent were able to defend against an equal number of attacks per trial indicating that both methods learned the task to an approximately equal degree of effectiveness. The pure RL based agent performed around twice as well as the control but was less then half as effective as the model-based methods, once again lend-

ing support to the conclusion that the pure RL based agent is hampered by its slow convergence times. This result, coupled with the number of failures, provide significant evidence that the model-based methods learned to perform the task with a significant degree of precision. They not only reduced the number of failures when compared to the control and pure RL based agent, but were also able to defend the city from more than twice as many attacks per trial.

Figure 3 depicts the average number of trials between failures for the first twenty-five failures of each agent averaged over a five trial window for smoothing purposes. This metric provides a means of measuring the speed of convergence of each of the adaptation methods. As can be seen, the reflection-guided RL agent shows the fastest convergence speed followed by the non-augmented model-based reflection. The pure RL did not appear to improve the task's execution until around the twelfth failed trial. After this point the control and the pure RL inter-trial failure rate begin to deviate slowly. Though not depicted in the figure, the performance of the pure RL based agent never exceeded a inter-trial failure rate of three even after all trials were run. This lends further evidence to the hypothesis that pure RL cannot learn an appropriate solution to this problem in the alloted number of trials though it should be noted that the performance of this agent did slightly outperform that of the control, indicating that some learning did occur. Surprisingly, the reflection-guided RL agent outperformed the pure model-based agent in this metric.

## 7 Conclusions

This work describes how model-based reflection may guide reinforcement learning. In the experiments described, this has been shown to have two benefits. The first is a reduction in learning time as compared to an agent that learns the task via pure reinforcement learning. The model-guided RL agent learned the task described, and did so faster then the pure RL based agent. In fact, the pure RL based agent did not converge to a solution that equaled that of either the pure model-based reflection agent or the reflection-guided RL agent within the alloted number of trials. Secondly, the reflection-guided RL agent shows benefits over the pure model-based reflection agent, matching the performance of that agent in the metrics measured in addition to converging to a solution in a fewer number of trials. In addition, the augmented agent eliminates the need for an explicit adaptation library such as is used in the pure-model based agent and thus reduces the knowledge engineering burden on the designer significantly. This work has only looked at an agent that can play a small subset of FreeCiv. Future work will focus largely on scaling up this method to include other aspects of the game and hence larger models and larger state spaces.

## References


[B. Krulwich and Collins, 1992] L. Birnbaum B. Krulwich and G. Collins. Learning several lessons from one experience. In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, pages 242–247, 1992.

[Barto and Mahadevan, 2003] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[Dietterich, 1998] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126, 1998.

[Fox and Leake, 1995] Susan Fox and David B. Leake. Using introspective reasoning to refine indexing. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1995.

[Kaelbling *et al.*, 1996] Leslie P. Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Murdock and Goel, 1998] J. William Murdock and Ashok K. Goel. A functional modeling architecture for reflective agents. In *AAAI-98 workshop on Functional Modeling and Teleological Reasoning*, 1998.

[Murdock and Goel, 2001] W. Murdock and A. K. Goel. Meta-case-based reasoning: Using functional models to adapt case-based agents. In *Proceedings of the 4th International Conference on Case-Based Reasoning*, 2001.

[Murdock and Goel, 2003] W. Murdock and A. K. Goel. Localizing planning with functional process models. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, 2003.

[Murdock, 2001] J. W. Murdock. *Self-Improvement Through Self-Understanding: Model-Based Reflection for Agent Adaptation*. PhD thesis, Georgia Institute of Technology, 2001.

[Stroulia and Goel, 1994] E. Stroulia and A. K. Goel. Learning problem solving concepts by reflecting on problem solving. In *Proceedings of the 1994 Euopean Conference on Machine Learning*, 1994.

[Stroulia and Goel, 1996] E. Stroulia and A. K. Goel. A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. In *Proceedings of AAAI'96*, pages 959–965, 1996.

[Sutton *et al.*, 1999] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[Tesauro, 1994] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[Ulam *et al.*, 2004] P. Ulam, A. Goel, and J. Jones. Reflection in action: Model-based self-adaptation in game playing agents. In *AAAI Challanges in Game AI Workshop*, 2004.

# The Design Space of Control Options for AIs in Computer Games

**Robert E. Wray[1], Michael van Lent[2], Jonathan Beard[1], Paul Brobst[2]**

[1]Soar Technology
3600 Green Road Suite 600
Ann Arbor, MI 48105 USA
{wray,beard}@soartech.com

[2]Institute for Creative Technologies
University of Southern California
13274 Fiji Way
Marina del Rey, CA 90292 USA
{vanlent,brobst}@ict.usc.edu

## Abstract

The goal of this paper is to define a design space for the control of multiple intelligent entities in computer games and to begin to evaluate options in this design space. While both finite state machine and agent technologies have been used in games, these approaches alone are usually insufficient to meet the competing requirements of game AI. We introduce new control options, based on heterogeneous application of existing technologies, entity aggregation, and different controller points-of-view. We then provide a preliminary evaluation of these options against believability, multi-entity scalability, knowledge scalability, and systems engineering criteria. Several options that potentially reduce the limitations of the individual approaches and enhance their strengths are identified.

## 1 Introduction

Game players crave both intelligent entities to play with and against and realistic worlds populated by large numbers of diverse, intelligent characters. These desires are inherently in conflict since, with fixed processing power, designers must choose between fewer, but more complex entities or greater numbers of less complex entities. The goal of this paper is to examine the requirements and enabling technology to increase the number of intelligent, synthetic entities that can be fielded in desk-top computer games while maintaining some degree of complex, intelligent entity behavior.

Typical game behavior technology often requires guidance from human players (or "cheating") to provide believable performance. For example, in real-time strategy (RTS) games, human players intervene and micro-manage individual entities when they reach tactically important decision points (such as engagements with other entities) because the entities generally lack the situation awareness and tactical knowledge to act successfully themselves.

More advanced technology exists for behavior modeling that can provide greater believability. For example, TacAir-Soar, a virtual pilot, has flown with and against human pilots in distributed simulation with full autonomy and believable interaction (via simulated speech) [1]. The primary limitation of this technology is its high computational overhead, relative to the behavior technology typically used in games. To fly hundreds of simulated TacAir-Soar entities, one usually needs tens of workstations with current technology. This requirement may be reasonable in distributed simulation, but will not provide large numbers of realistic entities in desk-top games.

Moore's Law and specialized hardware may one day solve this problem, just as these advances have reduced the computational challenges of three-dimensional, real-time rendering in computer games. However, it may be possible today to construct more believable systems by using combinations and hybridizations of existing behavior technology. This paper introduces a number of these options for the control of individual entities in games. We further outline the technologies for controlling entities in games as well as factors impacting their evaluation. We then describe a number of additional options based on heterogeneous use of the technologies, entity aggregation, and different controller points-of-view and discuss a preliminary evaluation of these options against the evaluation criteria.

## 2. Technologies for behavior representation

There are two general classes of technologies that have been deployed to represent behaviors in computer games: simpler approaches, such as finite state machines (FSMs), and agent architectures.

### 2.1 Finite State Machines

FSMs are a frequently used control structure in computer games for many reasons including: ease of implementation, efficiency, familiarity of technology, predictability, and player control. Ease of implementation allows rapid prototyping and enables more development iterations, producing a more polished product. Efficiency is always important in real-time systems; FSMs are lightweight and fast. Developer familiarity also helps with rapid prototyping; learning to use FSMs is relatively easy. Predictability helps quality assurance. Compared to other control schemes FSMs are easier to test and validate. Player control leads to 'gaming

the system.' Once players learn reactions of the entities they can use that information to their advantage. Some designers use this aspect of FSMs deliberately in design making "gaming the AI" part of the game experience.

Individual FSMs scale poorly as additional behaviors are added. Further, coordination between the entities is often desirable for increased believability. Entity coordination increases the behavioral complexity greatly in FSMs and, as the complexity of an FSM increases, the difficulty of authoring, maintaining, and augmenting these systems increases at a faster rate. Hierarchical state machines can be used to alleviate poor scalability, but do not solve the problem. Due to the scalability challenge, entity behaviors based on FSMs are usually fairly simple, which leads to greater predictability. In entertainment, predictability can cause boredom. In training uses of games, predictable entities can again allow a trainee to 'game the system' in ways that negatively impact learning.

## 2.2 Agent Approaches

There are many different agent architectures; most share some common functional capabilities that provide more run-time flexibility and autonomy than FSM approaches:

**Context-based reasoning/action**: Control and decision making in agent systems are based on the current situation, which includes both the external situation and internally recorded features (such as a history). Unlike FSMs, the allowable transitions between "states" in agent-based systems are not prescribed at design time. The agent interprets its situation from moment to moment and decides what actions should be considered based on that interpretation.

**Least commitment/run-time decision making**: Decisions (commitments) in agent systems are postponed as long as possible. Run-time decision-making gives the agent the ability to collect and interpret its situation before making a decision. Least commitment provides agent systems with significant flexibility in behavior, and avoids the predictability of behaviors that often results with FSMs.

**Knowledge-based conflict resolution**: An agent must be able to choose among any available options. Conflict resolution mechanisms provide the means to make choices among competing options. Critically, choices must be dependent on the context as well. Thus, agent knowledge will not only identify particular options, but will also provide help when evaluating them.

**Scalable knowledge bases**: Run-time decision making and knowledge-based conflict resolution generally require more explicit agent knowledge than FSM approaches. The additional knowledge requirements then force a requirement on the agent system to support large knowledge bases.

**Learning**: At least some agent architectures support learning. Learning can be used to improve reactivity (by compiling reasoning), acquire new knowledge, and adapt to new situations (new entity tactics).

**Table 1: A comparison of FSM and agent technology.**

| Evaluation Dimensions | FSMs | Agents |
|---|---|---|
| **Believability** | Low | High |
| **Multi-entity scalability** | High | Low |
| **Knowledge scalability** | Low | High |
| **Simplicity of systems engineering** | High | Low |

## 2.3 Evaluation dimensions for entity control

Both FSM and agent technology are important in game environments. However, they each bring individual strengths and limitations to the representation of entity behavior in those games. Table 1 lists four dimensions that are important for computer games and a qualitative, comparative evaluation of the technologies introduced above against those dimensions. The dimensions are:

**Believability**: This dimension refers to the quality of behavior generated by the entities. Believability improves the player's immersion in the game experience and the likelihood players will play against AIs repeatedly. In general, agent technology offers the potential for greater believability than FSMs, because the agent technology provides more flexibility and variability for behavior representation.

**Multi-entity scalability**: This dimension measures the maximum number of entities that can be represented in a desktop game. The greater the scalability, the richer the potential population of entities and interactions among them and game players. FSMs, because of their relatively light computational overhead, can support many more individual entities than most agent systems.

**Knowledge scalability**: This dimension refers to the ability of the technology to support an increasingly large knowledge store. Knowledge scalability includes both the ability to engineer increasingly large knowledge bases as well as the ability to execute them efficiently. Games increasingly require complex, on-going interactions among entities and players. These interactions, in order to offer variable, unpredictable, and believable experiences, require much greater knowledge resources, including deeper and broader domain knowledge and the ability to communicate and coordinate with other entities and players. Agent technology is generally designed to gracefully handle increasingly large stores of knowledge and it often includes many encapsulation and abstraction mechanisms to aid behavior developers in managing larger knowledge stores. While FSMs do not necessarily slow with increasing knowledge stores, they become significantly more complex with large knowledge stores. Most FSMs approaches offer only the state as a design abstraction, which can lead to combinatorial numbers of states in behavior representation.

**Simplicity of systems engineering**: This dimension refers to the ease of using a behavior system and integrating it with a game environment. C0mplex integrations can signifi-

cantly slow software development and increase development costs. FSMs generally provide simple integration and some FSM developers provide tools to explicitly support integration and behavior development in game environments [2]. Agent technologies generally do not yet provide interfaces geared specifically for games (one exception is Gamebots [3]), but increasingly offer tool suites to simplify integration and support behavior development.

## 3. Control architectures for game AIs

The evaluation in Table 1 assumed that a single technology was used for all entities in a game and each entity was represented with a single instance of each technology. However, there are a number of additional options that could be considered when creating AIs for a game. In particular, aggregating entities and/or using a combination of FSM and agent technologies might offer a way to capitalize on the strengths of the individual approaches and minimize their weaknesses. We consider the following three dimensions in the design space:

**Implementation technology**: The technology used to implement behavior in the environment. For this analysis, we assume the implementation technology is either *FSMs* or an *agent* architecture or a *heterogeneous* mix of both technologies (other approaches to behavior representation could be considered in future work).

**Grain-size of the behavior representation**: A single instance of a behavior representation technology can be used to control an individual entity or groups of entities. Aggregated representations are common for vehicles (e.g., a tank crew is typically represented as one agent or FSM). However, it might also be possible to group the control of related entities together under a single control process, even when the behavior of the single entities is distinct. We consider two values in this analysis: *single* entity and *aggregated* entities. We assume the level of aggregation is small (e.g., aggregation of a squad of 10 vs. the individual members of a company of 100's).

**Controller point-of-view**: Thus far, the point-of-view of the control process was assumed to be that of an individual *entity*. However, another common point-of-view in games (especially RTS games) is that of the player or "*commander*." Typically, the commander perspective offers a more global view of the game.

Table 2 lists the control options generated from the combination of the values for the dimensions proposed above. The first two options correspond to the approaches to entity representation in Sections 2.1 and 2.2. Other options are described in Section 3, as noted in the table.

Some additional options are not discussed in the paper. Option 4, in which an FSM is used to control an aggregation of entities is not necessary because FSMs generally have acceptable performance at the individual entity level. Because there is no advantage of aggregation for FSMs, option 6 is effectively identical to option 5, in which agent-controlled entities are aggregated.

**Table 2: Options for game AI control architecture.**

| | Point of View | Grain-size | Tech. | Additional Notes |
|---|---|---|---|---|
| 1 | Entity | Single | FSM | Typical approach; see Section 2.1 |
| 2 | Entity | Single | Agent | Section 2.2 |
| 3 | Entity | Single | Hetero | Multiple controllers/entity; Sec.3.1 |
| 4 | Entity | Aggregate | FSM | Not needed; single entity control is ok |
| 5 | Entity | Aggregate | Agent | Multiple entities/agent controller; Sec. 3.2 |
| 6 | Entity | Aggregate | Hetero | Not considered; identical to #5 |
| 7 | Command | Single | FSM | Game player controllers; Sec. 3.3 |
| 8 | Command | Single | Agent | Game player controllers; Sec. 3.3 |
| 9 | Command | Single | Hetero | Game player controllers; Sec. 3.3 |
| 10 | Command | Aggregate | FSM | Not considered; single commander |
| 11 | Command | Aggregate | Agent | Not considered; single commander |
| 12 | Command | Aggregate | Hetero | Not considered; single commander |
| 13 | Mixed | (either) | Hetero | Commander + individual entity control; Section 3.4 |

Options 10-12 (Command, Aggregate, *) are also not discussed. For this analysis, we assume that there is a single player or commander for the entities in the game. While it might be beneficial to populate the game with multiple commanders (as discussed in Section 3.4), in most games, there is only a single overall commander or player and thus aggregation at the level of the commander is meaningless.

Finally, we have added a 13th option representing both a mixed point of view (player and individual entities) and a heterogeneous mix of implementation technologies. While this mixed point-of-view could be considered with homogeneous implementation technologies, we focus on this option because FSM and agent technologies are particularly well-suited to the entity and commander points-of-view respectively (detailed in Section 3.4).

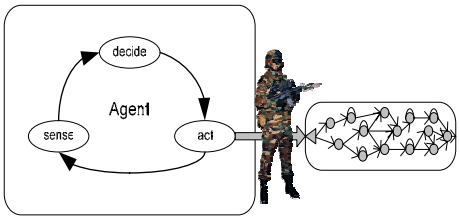


**Figure 1: Using multiple controllers/entity (Opt. 3a).**

### 3.1 Multiple controllers per entity

Option 3 in Table 2 combines both FSM and agent technology for the control of individual entities. There are two different ways in which this heterogeneous option could be approached. Option 3a is to use agents and FSMs together to control individual entities. Option 3b is to dynamically switch between control technologies.

In option 3a, illustrated in Figure 1, agent reasoning could focus on high level actions such as tactical behavior and then execute this behavior by triggering specific FSMs designed to accomplish the action. This approach is roughly equivalent to the sequencing and reactive layers of 3T(iered) architectures [4].

The option 3a approach does match both FSMs and agents to appropriate levels of representation. For example, individual FSMs could be used to execute low-level behaviors such as navigation but need not be connected together to form a complete set of entity behaviors. Agent execution can focus on high level behaviors without decomposing them to very fine-grained behaviors, which reduces the overall representation requirements for the agent.

The primary drawback of this approach is that it requires the computational overhead of *both* FSMs and agent technology for each entity. Thus, it is unlikely to scale to games where many entities are needed. To address this drawback, one could design the system so that when the FSM was executing, the agent was "paused" and not consuming system resources. This option may be worth additional exploration. However, it requires solutions to the problem of maintaining situation awareness when the agent component is suspended. For example, if the agent initiated a move-to-x FSM in a situation where the entity was not under fire, if the entity comes under fire in the course of execution, the agent needs to be aware of this new situation, in order to possibly reconsider or interrupt the executing FSM.

Option 3b is to switch control of the entity from one type of controller to another. In RTS games, human players intervene and micro-manage individual entities when they reach tactically important decision points (such as engagements with other entities). FSMs entities often lack the situation awareness and tactical knowledge to manage the situation believably themselves. However, in other situations, the human player is content to allow the built-in behaviors to execute with little oversight, because the lack of believability does not impact game outcomes. If an agent could be triggered when the situation required more powerful reasoning, then an improved level of believability might be realized when it really mattered. At any point in time, only a subset of the total number of entities would need to be controlled by agents, making the approach more scalable than agents alone. This option is different than 3a because the agent knowledge would cover all behavior when it was "activated." While not necessarily a drawback, it is not known how and when to switch controllers and how computationally demanding switching might be. In particular, the FSM controller will need to save situation interpretation information from the FSM and then communicate it to the newly initiated agent, which could be resource intensive.

### 3.2 Multiple entities per agent controller

Figure  illustrates option 5 from Table 2. There is a single instance of an agent but the single instance controls multiple entities simultaneously. Knowledge is represented much as it is in the baseline option (2.2), so that the agent is reasoning about each individual entity in the system.

The main advantage of this approach is that it amortizes the relatively high overhead cost of the agent architecture over multiple entities. For example, a small number of entities (~10) might be implemented with about the same performance cost as a single agent. This hypothesis assumes that the entity knowledge representations are somewhat homogeneous across entities (a squad of infantry).

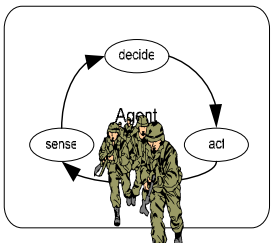


**Figure 2: Using a single agent to control multiple entities.**

This option would simplify some knowledge representation (e.g., behavior coordination between aggregated entities). The agent would maintain awareness of the state of a collection of entities and generate coordinated behaviors within the unit.

One challenge to this approach is to develop the knowledge for organizing and controlling the team, because this knowledge is often implicit in human systems and must be made explicit in the knowledge encodings for the entities.

This option imposes some additional requirements on the agent system. In particular, it must be capable of representing and pursuing multiple, simultaneous goals in parallel. Without this capability present in the architecture, control and task-switching algorithms between the entities must be created by hand, using the knowledge representations of the architecture. Without careful design, such as hand-shaking protocols, it may be possible to enter deadlock situations where two entities controlled by the architecture are each awaiting actions from the other. It may also be possible to create a general solution to manage multiple goals in single goal systems. For example, Soar Technology has developed a behavioral design pattern [5] for individual Soar agents that allows an agent using declared (non-architectural) goals

to create and manage multiple hierarchical decompositions of goals in agent memory.

### 3.3 Game player controllers

This section reviews options 7-9 from Table 2. Control is initiated from a commander's point of view, comparable to the role and actions a human player takes in RTS games

Categorical examples of this approach are chess playing programs. A chess program takes the global battlefield view and (a good program) takes into account the specific situations of all the individual entities when making decisions. Individual entities are fully controlled by the player, who dictates if, when, and where an entity will act.

In chess, however, individual entities (pieces) do not need any local autonomy. The environmental complexity and richness of most current computer games, however, makes the computational demands of determining actions for each entity and the communicating those commands to the game infeasible from just the point-of-view of the commander. These options are similar to the aggregation options in Section 3.2, but the level of aggregation is over all entities.

### 3.4 Commander + individual entity control

Option 13 in Table 2 represents a combination of both the commander and single entity points-of-view. Just like a human RTS player, the commander in this option employs other controllers to execute behavior at the entity level. We assume the commander is implemented with agent technology and the individual entities as FSMs. This approach capitalizes on the knowledge scalability strengths of agent technology, which is important for the commander as it interprets the overall situation and considers tactical decisions, and the multi-entity scalability of FSMs, which control the many individual entities in the game. This approach has been used at ICT in the Adaptive Opponent project [6] and at Soar Technology in a command and control simulation [7]. In the Soar Tech simulation, 5 Soar agents lead a company of about 100 FSM entities. This approach is also comparable to the approach to coordination and teamwork in the JACK agent architecture [8].

Likely limitations of this approach are increased communication bandwidth and possibly a lower level of believability of individual entities. The commander agent will issue many orders to the individual FSM entities, especially at times when a high degree of coordination or interaction is necessary. The communication bandwidth requirements can then become the limiting factor in performance. For example in the command and control application mentioned above, the additional cost of communication limited the addition of more FSM entities. Communication latencies and the limited capacity of the commander to respond immediately to every entity also may limit the believability of the overall system. Without commander input, believability of individual FSM entities is generally low (Table 1). The open question is whether the addition of the commander can significantly improve believability by "filling in" some of the missing capabilities of the FSM and in, doing so, whether the overall scalability of the system is maintained.

## 4. Initial evaluation of new control options

Figure 3 summarizes each option in comparison to the technology evaluation criteria introduced in Section 2. Each option is cast relative to FSM and agent technologies, based on the discussion of possible strengths and limitations in Section 3. These evaluations are preliminary and further analysis is planned.

**Believability**: Both options 3 and 5 should provide increased believability over FSMs, because they employ the more flexible agent technology at the entity level. We suggest option 3 may provide increased believability over option 5 because the control algorithms needed for option 5 may force some serialization in behavior. Option 13 can only be placed into the believability spectrum with empirical analysis. The key question is the extent to which the commander's oversight can lessen the believability limitations of the individual entity FSM controllers. However, regardless of the impact of the commander, it is doubtful that believability comparable to single entity, single agent controllers could be achieved.

**Multi-entity scalability**: Both options 3 and 5 are likely to scale only marginally better than agent controllers alone, because they depend on individual (or aggregate) agent controllers. Option 13, which introduces a single (or a few) commander agents along with FSM entity controllers should scale very well, comparable to FSM technology alone.

**Knowledge scalability**: Options 3 and 13 should both scale well in terms of the richness of individual entity knowledge representations. Option 3 uses agent systems to represent high level tactical knowledge including coordination, communication and teamwork. Option 13 leverages the additional commander agent(s) to improve overall knowledge scalability of the system. Individual FSMs remain relatively simple while increasing overall behavioral sophistication. Option 5 should scale reasonably well, but an unknown is the complexity of the algorithms used for switching between agents and the extent to which underlying agent architectures can naturally support these algorithms.

**Simplicity of systems engineering**: Both options 3 and 5 are likely to be **more** complex (and thus lower on the simplicity scale) than individual agent controllers. Option 3 requires one to develop a solution that integrates agent and FSM controllers. Option 5 requires the development of the switching algorithms and interfaces to each individual entity in the aggregate. In contrast to these approaches, Option 13 is relatively simple. It does add agent controllers, but only one (or a few). There is a natural knowledge representation for these entities, corresponding to knowledge a player uses when playing the game. Option 13 does require the addition of communication infrastructure between the commander and the FSM controlled entities.

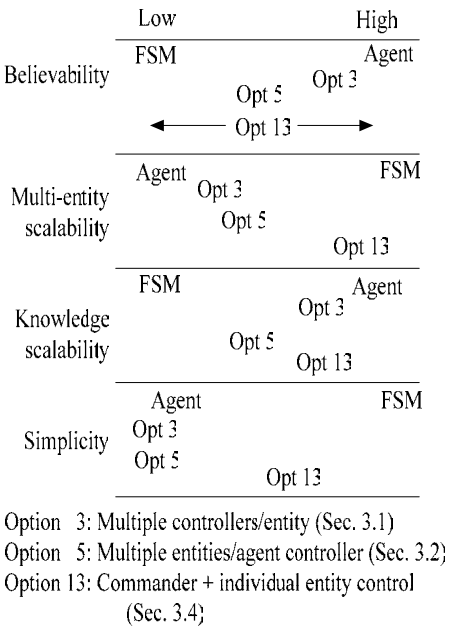


**Figure 3: Prospective summary of options in comparison to evaluation dimensions in Table 1.**

Based on this analysis, the commander + individual entity control (Option 13) appears to be the most promising option to explore, because it is the only option to appear on the "high side" of the evaluation dimensions at least three times. Developing multiple, heterogeneous controllers/individual entity (Option 3) may also be a reasonable option to pursue. If general, reusable interfaces between FSMs and agent technologies can be developed, this would lessen the negative impact of simplicity in its evaluation. Option 5 does not appear very promising, as the increased complexity of the system does not appear to offer clear benefits elsewhere. However, if the switching algorithms turned out to be relatively simple and the agent architecture directly supported multiple, simultaneous goals, then both believability and knowledge scalability might be more akin to those of individual agent controllers and make this option more viable.

## 5. Conclusions

This paper has introduced a number of different options for achieving large numbers of believable, knowledge rich agents using technologies that are relatively easy to integrate with game environments. We provided a preliminary comparative evaluation of the options. Empirical evaluation is needed to determine the exact properties of each option but the evaluation here prioritizes the different options and suggests questions to ask in an empirical evaluation.

The evaluation dimensions are concerned with the effect on gameplay and game production. Different or additional criteria may be appropriate for different uses of games. For example, in training environments, rather than believability (a subjective, vague concept), fidelity with human behavior might be more important than other dimensions.

Finally, we evaluated only two technologies, the finite state machines typical of modern computer game AIs and agent architectures. Other technologies could also be evaluated against the dimensions proposed here and, importantly, new options generated via the combination of promising technologies with FSMs and agents.

## Acknowledgements

The project or effort depicted is sponsored by the U.S. Army Research, Development, and Engineering Command (RDECOM), and that the content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

## References

[1] R. M. Jones, J. E. Laird, P. E. Nielsen, K. J. Coulter, P. G. Kenny, and F. V. Koss, "Automated Intelligent Pilots for Combat Flight Simulation," *AI Magazine*, vol. 20, pp. 27-42, 1999.

[2] R. Houlette, D. Fu, and R. Jensen, "Creating an AI Modeling Application for Designers and Developers," presented at AeroSense-2003 (Proceedings SPIE Vol. 5091), Orlando, 2003.

[3] R. Adobbati, A. N. Marshall, G. Kaminka, A. Scholer, and S. Tejada, "Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research," presented at 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal, Canada, 2001.

[4] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, pp. 237-256, 1997.

[5] G. Taylor and R. E. Wray, "Behavior Design Patterns: Engineering Human Behavior Models," presented at 2004 Behavioral Representation in Modeling and Simulation Conference, Arlington, VA, 2004.

[6] M. van Lent, M. Riedl, P. Carpenter, R. McAlinden, and P. Brobst, "Increasing Replayability with Deliberative and Reactive Planning," presented at Artificial Intelligence and Interactive Digital Entertainment Conference, Marina del Rey, CA, 2005.

[7] S. Wood, J. Zaientz, J. Beard, R. Frederiksen, and M. Huber, "CIANC3: An Agent-Based Intelligent Interface for the Future Combat System," presented at 2003 Conference on Behavior Representation in Modeling and Simulation (BRIMS), Scottsdale, Arizona, 2003.

[8] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas, "JACK: Summary of an Agent Infrastructure," presented at Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the Fifth International Conference on Autonomous Agents, Montreal, Canada, 2001.

# A Scheme for Creating Digital Entertainment with Substance

**Georgios N. Yannakakis and John Hallam**
Mærsk Mc-Kinney Møller Institute
for Production Technology
University of Southern Denmark
Campusvej 55, DK-5230, Odense M
{georgios;john}@mip.sdu.dk

## Abstract

Computer games constitute a major branch of the entertainment industry nowadays. The financial and research potentials of making games more appealing (or else more interesting) are more than impressive. Interactive and cooperative characters can generate more realism in games and satisfaction for the player. Moreover, on-line (while play) machine learning techniques are able to produce characters with intelligent capabilities useful to any game's context. On that basis, richer human-machine interaction through real-time entertainment, player and emotional modeling may provide means for effective adjustment of the non-player characters' behavior in order to obtain games of substantial entertainment. This paper introduces a research scheme for creating NPCs that generate entertaining games which is based interdisciplinary on the aforementioned areas of research and is foundationally supported by several pilot studies on testbed games. Previous work and recent results are presented within this framework.

## 1 Introduction

While game development has been concentrated primarily on the graphical representation of the game worlds, minor focus has been given to non-player characters' (NPCs') behavior. Simple scripted rules and finite-state or fuzzy-state machines are still used to control NPCs in the majority of games [Woodcock, 2001; Cass, 2002]. The increasing number of multi-player online games (among others) is an indication that humans seek more intelligent opponents and richer interactivity. Advanced artificial intelligence techniques are able to improve gaming experience by generating intelligent interactive characters and furthermore cover this human demand [Funge, 2004]. Moreover, computational power may bring expensive innovative AI techniques such as machine learning to meet a game application in the near future.

Game players seek continually for more enjoyable games as they spent 3.7 days per week playing an average of 2.01 hours per day [Rep, 2002], as stressed in [Fogel *et al.*, 2004], and this interest should somehow be maintained. It is only very recently that game industry has begun to realize the great (financial) importance of stronger AI in their products. Boon [2002] stresses that the most common complaint that gamers have is that the game is too short. However, as Boon claims, rather than making games longer game developers should focus on making games more interesting and appealing to both hard-core and soft-core gamers.

Intelligent interactive opponents can provide more enjoyment to a vast gaming community of constant demand for more realistic, challenging and meaningful entertainment [Fogel *et al.*, 2004]. However, given the current state-of-the-art in AI in games, it is unclear which features of any game contribute to the satisfaction of its players, and thus it is also uncertain how to develop enjoyable games. Because of this lack of knowledge, most commercial and academic research in this area is fundamentally incomplete.

In this paper, a general scheme for obtaining digital entertainment of richer interactivity and higher satisfaction is presented. On that basis, previous achievements within this framework are outlined and new promising results that reinforce our intentions are portrayed. The challenges we consider within the proposed scheme are to provide qualitative means for distinguishing a game's enjoyment value and to develop efficient tools to automatically generate entertainment for the player. In that sense, investigation of the factors/criteria that map to real-time enjoyment for the player as well as the mechanisms that are capable of generating highly entertaining games constitute our primary aims. The levels of human-game interaction that we consider in this work are focused on the player's actions and basic emotions and their impact to the behavior of the NPCs.

## 2 Related Work

This section presents a literature review on the interdisciplinary research fields that the proposed scheme attempts to combine. These include entertainment metrics for computer games; on-line adaptive learning approaches; user modeling and player's emotional flow analysis in computer games.

### 2.1 Entertainment metrics

Researchers in the AI in computer games field are based on several empirical assumptions about human cognition and human-machine interaction. Their primary hypothesis is that by generating human-like opponents [Freed *et al.*,

2000], computer games become more appealing and enjoyable. While there are indications to support such a hypothesis (e.g. the vast number of multi-player on-line games played daily on the web) and recent research endeavors to investigate the correlation between believability of NPCs and satisfaction of the player [Taatgen *et al.*, 2003], there has been no evidence that a specific opponent behavior generates more or less interesting games.

Iida's work on measures of entertainment in board games was the first attempt in this area. He introduced a general metric of entertainment for variants of chess games depending on average game length and possible moves [Iida *et al.*, 2003]. On that basis, some endeavors towards the criteria that collectively make simple online games appealing are discussed in [Crispini, 2003]. The human survey-based outcome of this work presents challenge, diversity and unpredictability as primary criteria for enjoyable opponent behaviors.

### 2.2 On-Line Adaptive Learning

There is a long debate on which form of learning is the most appropriate and feasible for a computer game application. In between off-line and on-line learning, the latter can be slow and lead to undesired and unrealistic behavior but it can demonstrate adaptive behaviors. Off-line learning is more reliable but it generally generates predictable behaviors [Manslow, 2002; Champandard, 2004]. However, researchers have shown that on-line learning in computer games is feasible through careful design and efficient learning methodology [Demasi and de O. Cruz, 2002; Johnson, 2004; Ponsen and Spronck, 2004; Stanley *et al.*, 2005].

### 2.3 User Modeling in Computer Games

Player modeling in computer games and its beneficial outcomes have recently attracted the interest of a small but growing community of researchers and game developers. Houlette's [2004] and Charles' and Black's [2004] work on dynamic player modeling and its adaptive abilities in video games constitute representative examples in the field. According to [Houlette, 2004], the primary reason why player modeling is necessary in computer games is in order to recognize the type of player and allow the game to adapt to the needs of the player. Many researchers have recently applied such probabilistic network techniques for player modeling on card [Korb *et al.*, 1999] or board games ([Vomlel, 2004] among others) in order to obtain adaptive opponent behaviors.

### 2.4 Real-time Emotional Flow

For modeling the player's emotions real-time, we gain inspiration primarily from the work of Kaiser et al. [1998]. They attempted to analyze emotional episodes, facial expressions and feelings — according to the Facial Coding Action System [Eckman, 1979] — of humans playing a predator/prey computer game similar to Pac-Man [Kaiser and Wehrle, 1996].

## 3 The Scheme

As previously mentioned, given the current state-of-the-art in AI in games, it is unclear which features of any game contribute to the enjoyment of its players, and thus it is also doubtful how to generate enjoyable games. Research endeavors aiming to do this without clear understanding of what factors yield enjoyable gaming experience will inevitably be unsuccessful.

In order to bridge the current gap between human designation of entertainment and interest generated by computer games and to find efficient and robust paths in obtaining appealing games, there is a need for an intensive and interdisciplinary research within the areas of AI, human-computer interaction and emotional and cognitive psychology. We therefore aim at exploring the novel directions opened by previous work on introducing entertainment measurements and adaptive on-line learning tools for generating interesting computer games [Yannakakis and Hallam, 2004a; 2005b]. The long-term target of this work is to reveal the direct correlation between the player's perceived entertainment $(I)$, his/her playing strategy (style) $(U)$ and his/her emotional state $(E)$ — see Figure 1. Such a perspective will give insights into how a game should adapt to and interact with humans, given their emotional state and playing skills, in order to generate high entertainment. Towards this purpose, an innovative computer game will be developed, based on an interactive system that will allow one to study the ongoing processes of situated game state, the user's playing style and emotional flow.

The steps towards meeting our objectives are described in the following sections (see Figure 1). Previous and recent research achievements at each part of the proposed scheme are also presented.

## 4 Enjoyable Cooperative Opponents

Cooperation among multiple game characters portrays intelligent behavior and exhibits more enjoyment for the player. From that perspective, teamwork is a desired gaming opponent behavior. We therefore experiment with games of multiple opponents where cooperative behaviors could be generated and studied.

A set of games that collectively embodies all the above-mentioned environment features is the predator/prey genre. We choose predator/prey games as the initial genre of our game research [Yannakakis *et al.*, 2004; Yannakakis and Hallam, 2004a] since, given our aims, they provide us with unique properties. In such games we can deliberately abstract the environment and concentrate on the characters' behavior. The examined behavior is cooperative since cooperation is a prerequisite for effective hunting behaviors. Furthermore, we are able to easily control a learning process through on-line interaction. In other words, predator/prey games offer a well-suited arena for initial steps in studying cooperative behaviors generated by interactive on-line learning mechanisms. Other genres of game (e.g. first person shooters) offer similar properties and will be studied later so as to demonstrate the methodology's generality over different genres of game.

### 4.1 Interest Metric

In order to find an objective measure of real-time entertainment (i.e. interest) in computer games we first need to empirically define the criteria that make a specific game interesting. Then, second, we need to quantify and combine all

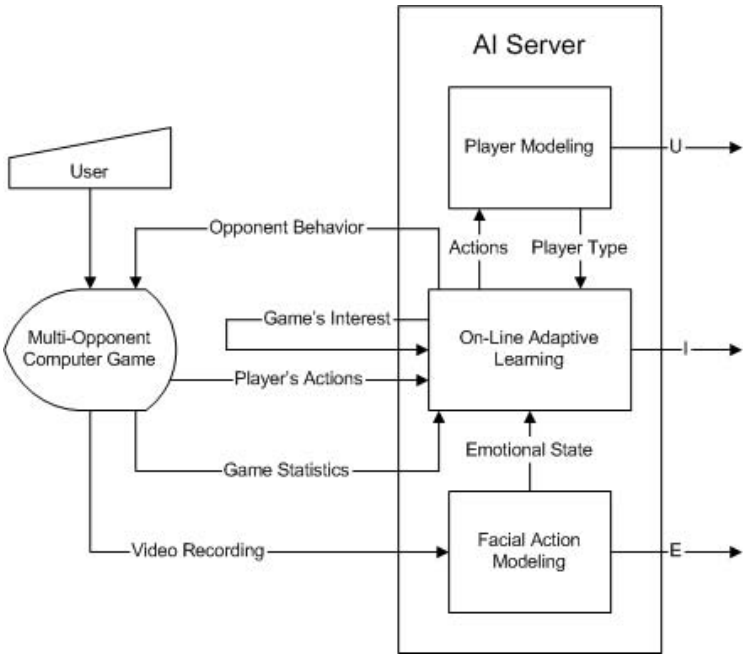


Figure 1: The proposed scheme.

these criteria in a mathematical formula. Subsequently, test-bed games should be tested by human players to have this formulation of interest cross-validated against the interest the game produces in real conditions. To simplify this procedure we will ignore the graphics' and the sound effects' contributions to the interest of the game and we will concentrate on the opponents' behaviors. That is because, we believe, the computer-guided opponent character contributes the vast majority of qualitative features that make a computer game interesting. The player, however, may contribute to its entertainment through its interaction with the opponents of the game and, therefore, it is implicitly included in the interest formulation presented here.

In [Yannakakis and Hallam, 2004a], we introduced the criteria that collectively define entertainment in predator /prey games. According to these criteria a game is interesting when: a) it is neither easy nor difficult to play; b) the opponents' behavior is diverse over the game and c) the opponents appear as they attempt to accomplish their predator task via uniformly covering the game environment. The metrics for the above-mentioned criteria are given by $T$ (difference between maximum and average player's lifetime over N games — N is 50), $S$ (standard deviation of player's lifetime over N games) and $E\{H_n\}$ (stage grid-cell visit average entropy of the opponents over N games) respectively. All three criteria are combined linearly (1)

$$I = \frac{\gamma T + \delta S + \epsilon E\{H_n\}}{\gamma + \delta + \epsilon} \quad (1)$$

where $I$ is the interest value of the predator/prey game; $\gamma, \delta$ and $\epsilon$ are criterion weight parameters.

By using a predator/prey game as a test-bed, the interest value computed by (1) proved to be consistent with the judgement of human players [Yannakakis and Hallam, 2005b]. In fact, human player's notion of interestingness seems to highly correlate with the $I$ value. Moreover, given each subject's performance (i.e. score), it is demonstrated that humans agreeing with the interest metric do not judge interest by their performance. Or else, humans disagreeing with the interest metric judge interest by their score or based on other criteria like game control and graphics.

The metric (1) can be applied effectively to any predator/prey computer game because it is based on generic features of this category of games. These features include the time required to kill the prey as well as the predators' entropy throughout the game field. We therefore believe that this metric — or a similar measure of the same concepts — constitutes a generic interest approximation of predator/prey computer games. Evidence demonstrating the interest metric's generality are reported in [Yannakakis and Hallam, 2004b; 2005a] through experiments on dissimilar predator/prey games. Moreover, given the two first interest criteria previously defined, the approach's generality is expandable to all computer games. Indeed, no player likes any computer game that is too hard or too easy to play and, furthermore, any player would enjoy diversity throughout the play of any game. The third interest criterion is applicable to games where spatial diversity is important which, apart from predator/prey games, may also include action, strategy and team sports games according to the computer game genre classification of Laird and van Lent [2000].

## 4.2 On-Line Learning

The next step we consider in our approach is to enhance the entertainment value of the examined computer game players based on the above-mentioned interest metric. This is

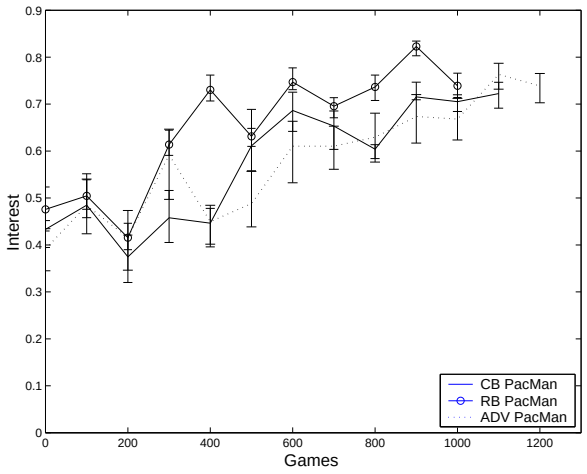


Figure 2: Game interest (average and confidence interval values) over the number of OLL games against different types of Pac-Man player. For reasons of computational effort, the OLL procedure continues for a number of games, large enough to illustrate the mechanism's behavior, after a game of high interest ($I \geq 0.7$) is found.

achieved collectively through evolutionary learning opponents via on-line interaction. We use an evolutionary machine learning mechanism which is based on the idea of heterogeneous cooperative opponents that learn while they are playing against the player (i.e. on-line). The mechanism is initialized with some well-behaved opponents trained off-line and its purpose is to improve the entertainment perceived by the player. More comprehensively, at each generation of the algorithm:

**Step 1:** Each opponent is evaluated periodically via an individual reward function, while the game is played.

**Step 2:** A pure elitism selection method is used where only a small percentage of the fittest solutions is able to breed. The fittest parents clone a number of offspring.

**Step 3:** Offspring are mutated.

**Step 4:** The mutated offspring are evaluated briefly in off-line mode, that is, by replacing the least-fit members of the population and playing a short off-line game against a selected computer-programmed opponent. The fitness values of the mutated offspring and the least-fit members are compared and the better ones are kept for the next generation.

The algorithm is terminated when a predetermined number of generations has elapsed or when the $I$ value has reached high values. Successful applications of this algorithm have demonstrated its generality over predator/prey variants [Yannakakis and Hallam, 2004b]; game complexity, game environment topology, playing strategy and initial opponent behavior [Yannakakis and Hallam, 2005a]. Figure 2 illustrates an example of the adaptability and robustness that on-line learning (OLL) demonstrates in a predator/prey game (a modified version of Pac-Man).

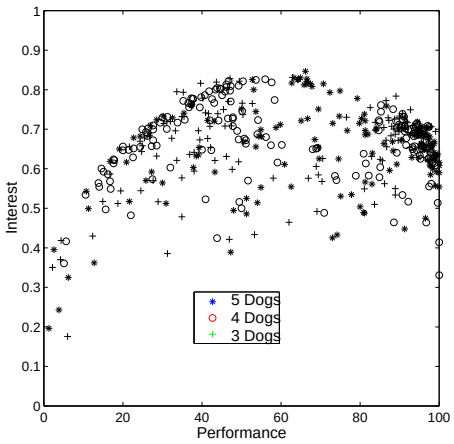


Figure 3: Scatter plot of $I$ and opponent (*Dog*) performance ($P$) value instances for variants of the Dead End predator/prey game.

For justifiable and realistic character behaviors we follow the animat approach as far as the control of the opponents is concerned. Consequently, we require opponents that demonstrate emergent cooperative behaviors whose intercommunication is based on indirect (implicit) and non-global (partial) information of their game environment.

Figure 3 illustrates a test-bed game application of OLL where the above-mentioned goal is achieved. Given the interest criteria, behaviors of high performance ought to be sacrificed for the sake of highly entertaining games. Consequently, there has to be a compromise between $P$ (performance) and $I$ values. However, as seen in Figure 3, teamwork features within the opponents (*Dogs* in the game of 'Dead End' as introduced in [Park, 2003] and [Yannakakis *et al.*, 2004]) behavior are maintained when interesting games emerge through the on-line learning mechanism. It appears that the most interesting games require a performance ($50 < P < 70$ approximately) which is not achievable without cooperation. Thus, teamwork is present during on-line learning and it furthermore contributes to the emergence of highly interesting games.

## 5 Player Modeling

The subsequent step to take is to study the players' contribution to their emergence of entertaining games as well as to investigate the relation between the player's type and the generated interest. We do that by investigating Player Modeling (PM) mechanisms' impact on the game's interest when it is combined with on-line adaptive learning procedures. By recording players' actions real-time we dynamically model the player and classify him/her into a player type $U$ (see Figure 1), which will determine features of the AI adaptive process (e.g. on-line learning mechanism).

More specifically, we use Bayesian Networks (BN), trained on computer-guided player data, as a tool for inferring appropriate parameter values for the chosen OLL mechanism. On-line learning is based on the idea of opponents that learn while they are playing against the player which, as previously

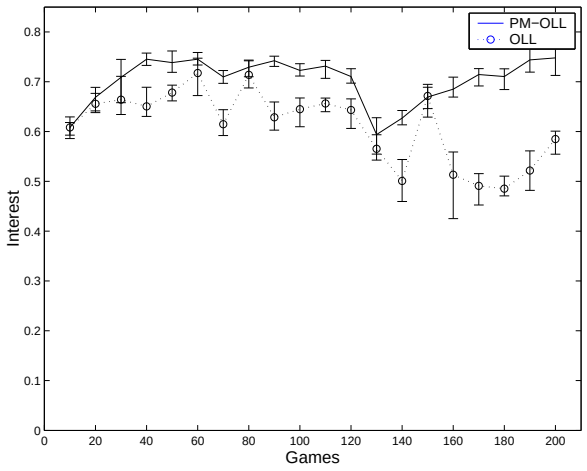


Figure 4: Adaptability test in the Pac-Man game: Random selection of hand-programmed strategies for the player every 10 games.

noted, leads to games of high interest. However, there are a number of parameters and a set of machine learning variants that strongly influence the performance of the on-line learning mechanism. Naive selection of the method and its parameter values may result in disruptive phenomena on the opponents' controllers and lead to unrealistic behavior.

A BN-based mechanism designed to lead to more careful OLL parameter value selection and furthermore to an increasingly interesting game is presented in [Yannakakis and Maragoudakis, 2005]. Results obtained show that PM positively affects the OLL mechanism to generate more entertaining games for the player. In addition, this PM-OLL combination, in comparison to OLL alone, demonstrates faster adaptation to challenging scenarios such as frequent changes of playing strategies (see Figure 4).

## 6 Player's Emotional Flow

The proposed scheme's part of future work includes the recording of players' behavior real-time by visual means in order to model their dynamical emotional flow $E$ through facial coding procedures. Real-time video data obtained from that environment will be analyzed through automatic face detection and the derived facial expressions will be classified to map basic emotions through algorithms based on the Facial Action Coding System [Eckman, 1979]. This procedure will effectively expose the relation of the player's emotional flow and his/her real-time entertainment which defines one of the objectives of this work. Accordingly, given this relation, the linkage between basic human emotions and features of the on-line adaptive learning will be revealed.

## 7 AI server and Game test-beds

There is already much evidence for the effectiveness, robustness and adaptability of both OLL and PM mechanisms. However, more complex cooperative games will demonstrate the generality of the methodology over different genres of games. Open source platform, multi-opponent, popularity and on-line gaming potential are the basic game selection criteria, which leave space for FPS and/or real-time strategy (RTS) games.

To cope with the high computational effort, an on-line web server that will host all AI processes (e.g. on-line adaptive learning) needs to be constructed (see Figure 1). The server will monitor game features (e.g. player's actions), reinforce the AI with its real-time generated interest $I$ and adjust the opponent behavior back to the game.

## 8 Conclusions & Discussion

In this paper we portrayed a scheme for obtaining computer games of richer interactivity and higher entertainment by focusing on the real-time adjustment of the opponent's controller. New unreported results on predator/prey game test-beds demonstrate the effectiveness and fast adaptability of the method in its endeavor to increase the entertainment value of the game and provide further evidence for its successful application to the other genres of game. However, in order for the proposed scheme to be complete, there are still steps that need to be taken towards the automatic recognition of basic emotions of players and their interactivity with the on-line adaptive learning procedures.

The potential of the proposed scheme lies in its innovative endeavor to bring emotional psychology, human-machine interaction and advanced AI techniques to meet upon computer game platforms. As soon as experiments with statistically significant numbers of human subjects are held, this work's outcome will provide important insights to spot the features of computer games — that map to specific emotions — that make them appealing to most humans. Moreover, the playing strategy and emotional features that generate entertainment in games will be exposed contributing important input for the game AI research community.

## References

[Boon, 2002] R. Boon. The 40 hour millstone. *Computer Trade Weekly*, (877), February 2002.

[Cass, 2002] S. Cass. Mind games. *IEEE Spectrum*, pages 40–44, 2002.

[Champandard, 2004] Alex J. Champandard. *AI Game Development*. New Riders Publishing, 2004.

[Charles and Black, 2004] D. Charles and M. Black. Dynamic player modelling: A framework for player-centric digital games. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 29–35, 2004.

[Crispini, 2003] Nick Crispini. Considering the growing popularity of online games: What contibutes to making an online game attractive, addictive and compeling. Dissertation, SAE Institute, London, October 2003.

[Demasi and de O. Cruz, 2002] Pedro Demasi and Adriano J. de O. Cruz. On-line coevolution for action games. In *Proceedings of the 3rd Interational Conference on Intelligent Games and Simulation (GAME-ON)*, pages 113–120, 2002.

[Eckman, 1979] P. Eckman. Facial expressions of emotions. *Annual Review of Psychology*, 20:527–554, 1979.

[Fogel *et al.*, 2004] David B. Fogel, Timothy J. Hays, and Douglas R Johnson. A platform for evolving characters in competitive games. In *Proceedings of the Congress on Evolutionary Computation (CEC-04)*, pages 1420–1426, June 2004.

[Freed *et al.*, 2000] M. Freed, T. Bear, H. Goldman, G. Hyatt, P. Reber, A. Sylvan, and J. Tauber. Towards more human-like computer opponents. In *Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, pages 22–26, 2000.

[Funge, 2004] John D. Funge. *Artificial Intelligence for Computer Games*. A. K. Peters Ltd, 2004.

[Houlette, 2004] R. Houlette. *Player Modeling for Adaptive Games. AI Game Programming Wisdom II*, pages 557–566. Charles River Media, Inc, 2004.

[Hy *et al.*, 2004] R. Le Hy, A. Arrigoni, P. Bessière, and O. Lebeltel. Teaching bayesian behaviors to video game characters. *Robotics and Autonomous Systems*, 47:177–185, 2004.

[Iida *et al.*, 2003] Hiroyuki Iida, N. Takeshita, and J. Yoshimura. A metric for entertainment of boardgames: its implication for evolution of chess variants. In R. Nakatsu and J. Hoshino, editors, *IWEC2002 Proceedings*, pages 65–72. Kluwer, 2003.

[Johnson, 2004] S. Johnson. *Adaptive AI*, pages 639–647. Charles River Media, Hingham, MA, 2004.

[Kaiser and Wehrle, 1996] S. Kaiser and T. Wehrle. Situated emotional problem solving in interactive computer games. In N. H. Frijda, editor, *Proceedings of the VIXth Conference of the International Society for Research on Emotions*, pages 276–280. ISRE Publications, 1996.

[Kaiser *et al.*, 1998] S. Kaiser, T. Wehrle, and S. Schmidt. Emotional episodes, facial expressions, and reported feelings in human computer interactions. In A. H. Fisher, editor, *Proceedings of the Xth Conference of the International Society for Research on Emotions*, pages 82–86. ISRE Publications, 1998.

[Korb *et al.*, 1999] K. B. Korb, A. E. Nicholson, and N. Jitnah. Bayesian poker. uncertainty in artificial intelligence, 1999. Stockholm, Sweden.

[Laird and van Lent, 2000] John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI)*, pages 1171–1178, 2000.

[Manslow, 2002] John Manslow. *Learning and Adaptation*, pages 557–566. Charles River Media, Hingham, MA, 2002.

[Park, 2003] Jeong Kung Park. Emerging complex behaviors in dynamic multi-agent computer games. M.Sc. thesis, University of Edinburgh, 2003.

[Ponsen and Spronck, 2004] Marc Ponsen and Pieter Spronck. Improving adaptive game AI with evolutionaey learning. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 389–396, Microsoft Campus, Reading, UK, November 2004.

[Rep, 2002] Credit Suiss First Boston Report. 11:7–8, May 15 2002.

[Stanley *et al.*, 2005] Kenneth Stanley, Bobby Bryant, and Risto Miikkulainen. Real-time evolution in the NERO video game. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 182–189, Essex University, Colchester, UK, 4–6 April 2005.

[Taatgen *et al.*, 2003] N. A. Taatgen, M. van Oploo, J. Braaksma, and J. Niemantsverdriet. How to construct a believable opponent using cognitive modeling in the game of set. In *Proceedings of the fifth international conference on cognitive modeling*, pages 201–206, 2003.

[Vomlel, 2004] J. Vomlel. Bayesian networks in mastermind. In *Proceedings of the 7th Czech-Japan Seminar*, 2004.

[Woodcock, 2001] Steven Woodcock. Game AI: The State of the Industry 2000-2001: It's not Just Art, It's Engineering. August 2001.

[Yannakakis and Hallam, 2004a] Georgios N. Yannakakis and John Hallam. Evolving Opponents for Interesting Interactive Computer Games. In S. Schaal, A. Ijspeert, A. Billard, Sethu Vijayakumar, J. Hallam, and J.-A. Meyer, editors, *From Animals to Animats 8: Proceedings of the $8^{th}$ International Conference on Simulation of Adaptive Behavior (SAB-04)*, pages 499–508, Santa Monica, LA, CA, July 2004. The MIT Press.

[Yannakakis and Hallam, 2004b] Georgios N. Yannakakis and John Hallam. Interactive Opponents Generate Interesting Games. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 240–247, Microsoft Campus, Reading, UK, November 2004.

[Yannakakis and Hallam, 2005a] Georgios N. Yannakakis and John Hallam. A generic approach for generating interesting interactive pac-man opponents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 94–101, Essex University, Colchester, UK, 4–6 April 2005.

[Yannakakis and Hallam, 2005b] Georgios N. Yannakakis and John Hallam. How to generate interesting computer games. submitted, January 2005.

[Yannakakis and Maragoudakis, 2005] Georgios N. Yannakakis and Manolis Maragoudakis. Player modeling impact on player's entertainment in computer games. In *Proceedings of the $10^{th}$ International Conference on User Modeling*, Edinburgh, 24–30 July 2005.

[Yannakakis *et al.*, 2004] Georgios N. Yannakakis, John Levine, and John Hallam. An Evolutionary Approach for Interactive Computer Games. In *Proceedings of the Congress on Evolutionary Computation (CEC-04)*, pages 986–993, June 2004.

# Author Index