Using a Semiotics-Inspired Tool for the Control of Intelligent Opponents in Computer Games

Victor K. Tatai  
DCA – FEEC – UNICAMP  
Cx Postal 6101 – 13081-970  
Campinas – SP - Brazil  
tatai@dca.fee.unicamp.br

Ricardo R. Gudwin  
DCA – FEEC – UNICAMP  
Cx Postal 6101 – 13081-970  
Campinas – SP - Brazil  
gudwin@dca.fee.unicamp.br

ABSTRACT
This paper describes the use of a semiotics-inspired technology - semionics, in the control system of an intelligent agent used as an opponent in a first-person shooter computer game - Counter-Strike.

Keywords
Computational semiotics, SNTool, intelligent systems, computer games, Counter-Strike.

1. INTRODUCTION
Semionics is a semiotic-inspired technology developed at the Dept. of Computer Engineering and Industrial Automation at the Faculty of Electrical and Computer Engineering - University of Campinas - Brazil. Its main purpose is to synthesize artificial minds for intelligent agents of generic kinds. This technology has its roots in many different methodologies used in intelligent systems development, like Complex Adaptive Systems (CAS - the former Classifier Systems from Holland [1]), Petri nets and neural networks, but its development was fostered by the desire to model sign-processes in a computational way, being so a contribution to the field of Computational Semiotics, or, in other words, the artificial synthesis of semiotic processes. The main modeling tool in semionics is the semionic network [2]. Alongside to the formal development of semionic networks, our group developed a software tool - SNTool [3,4,5], as a help on the development and simulation of semionic networks within practical applications. Particularly, in this paper, we describe the use of semionic networks (and also SNTool) in the design and control of intelligent opponents in a computer game. The game chosen is a first-person shooter (FPS) entitled Counter-Strike [6], which enjoys an immense popularity among the multiplayer gaming community. SNTool is thus responsible in the context of the game for controlling “intelligent” opponents or allies named bots, which play alongside humans (or other bots). This control is made by the use of behaviors regulated by emulated “emotions”.

2. SNTOOL
SNTool is a computational tool used to model and execute semionic networks. In semionic networks, both the model and the runtime operation of a system have simple graphical representations, fact which greatly aids the process of designing and debugging complex systems. A more detailed explanation of semionic networks may be found on [2].

The behavior of a semionic network is intended to simulate the processing of signs within an artificial mind. The main constituents of a semionic network are the signlets and the semionic agents (or s-agents). Signlets perform the behavior of a sign within the artificial mind of an agent being modeled. S-agents perform the role of micro-interpreters or, in other words, those active entities responsible for transforming one or more signlets into newer signlets, which are supposed to be the interpretants of the former within a semiotic process [2]. Both signlets and s-agents are represented by tokens and distributed into places, which are connected by arcs within a semionic network, being able to move around the network, according to some rules. Each place within the network is associated to a signlet class, which corresponds to a class as used in most object oriented (OO) languages. Places are linked to each other by directed arcs. The connection between an arc and a place is mediated by a gate. There are two distinct types of places: active and passive. Active places hold s-agents, which are responsible for: - selecting signlets; - destroying or preserving these signlets; - executing internal operations which may assimilate, merge or generate signlets. A sample network may be viewed on figure 1.

![Figure 1. A sample network depicting places, arcs and objects.](image-url)
actions to take place, a function is then composed of two distinct operators: the **match** and the **process** operators. The match operator determines the **utility value** of the function given the input signlets accessible through its input gates. The process operator then executes the necessary operations over these signlets, generating other signlets if pertinent. The type of action taken by a function over its input signlets is defined at each time step by the **access mode** which it uses to access a given input gate. The access mode indicates if the corresponding input signlet will have shared or exclusive access, or be consumed.

A semionic network is initialized by allocating signlets and s-agents to places, respecting the constraint regarding the classes for each place. Also, an s-agent must be inserted in a place which has all the input and output gates that it needs to properly work. As the network is started, at each time step selected s-agents are chosen to **fire** given their different utility values (see figure 2). The agents which are supposed to fire are determined through the use of a constraint-satisfaction algorithm [1], which solves conflicts arising from incompatible access modes giving higher priority to functions with a higher utility value.

Semionic networks are quite similar in behavior to Petri nets. They have however many advantages over them (or even over their derivatives, such as coloured or OO Petri nets). Semionic networks clearly distinguish the dynamic and static elements of a network, thus easing the design process, and are a more powerful representation form allowing for example complex adaptive systems to be more easily represented, also assuming the role of an integration tool between an arbitrary number of complex systems. This happens because agents can be as complex as necessary, consisting from a simple if statement to, for example, full neural networks, genetic algorithms, or even other semionic networks. The interaction among these complex elements always follows however the framework defined by the semionic network, thus allowing for a more organized and productive view of the design and execution process.

The computational tool which models and simulates semionic networks, SNTool, is currently in its third inception, having been implemented using the Java language. SNTool is a complete development platform, enabling a user to design the network, edit code, compile and debug. A screenshot for SNTool may be observed in figure 3.

Current SNTool implementation counts also with a C++ core engine, thus allowing it to be easily integrated in a computer game like Counter-Strike, with a more than satisfactory performance. This C++ module might also be used in the control of other real-time systems as well which present similar real-time requisites.

3. **COUNTER STRIKE**

Counter-Strike (CS) is a FPS game which allows a player to command her avatar (the player’s representation in the game) in a virtual three-dimensional world alongside other players. CS is usually thought of as purely multiplayer game, as it always pitches two teams of players against each other. These teams are labeled either as **terrorists** or **counter-terrorists**, which play roles and have objectives roughly "equivalent" to their real-world ones. The game’s objectives are multiple, and depend not only of the team the player is in, but also on the current world (denominated **map**) where the game is taking place. An objective however that is always valid is to eliminate all the players of the opposing team. Also the game defines a certain amount of time for the teams to accomplish a mission, after which one team wins (according to the objectives set for the current map). This game interval is called a **round**.

The objectives according to the map played may be:

- For the terrorists:
  - Plant a bomb at a pre-determined location.
  - Avoid the rescue of hostages.
  - Assassinate the VIP.
  - Escape to a pre-determined location.

- For the counter-terrorists:
  - Disarm bomb when armed.
  - Rescue hostages.
Central to this discussion is the games such as FPSs, Role Playing Games – RPG and Real character (note that here we consider primarily recent computer when developing a viable architecture to the control of a game

There are several factors that should be taken into account to the success of a team. One other interesting feature of CS is that all these equipments are purchased at the beginning of the round using money. This money is obtained by killing enemies or accomplishing objectives, what contributes even further to the team-oriented aspect of the game.

Another interesting aspect of CS is its rich user interaction, not only allowing for vision impairment with the use of gas and flash grenades, but also emitting different sounds for the player’s interaction with the environment. The game also has inter-player communication, either by typed text or voice, or by using a set of pre-defined messages. These pre-defined messages allow for a very simple yet effective way for human players (or even bots) to communicate with (other) bots.

The development of bots however presents a number of peculiarities regarding their interface to the CS game. First among these concerns the sensing of the world; it would be a daunting task to implement a vision recognition system to “view” the world as we do, as would be to implement voice, noise and natural language processing systems. The strategy commonly adopted is to access the internal games structures directly, thus bypassing some of these difficulties. Of course the interaction becomes more limited, but it is not usually viewed as a limiting factor to the performance of a bot. Performance here is usually assessed not as its killing efficiency – what might be easily accomplished by a bot’s 100% accurate aim – but as the bot’s capability to emulate as close as possible all the activities commonly performed by a human player, providing the human player with an enjoyable gaming session. In a certain sense then, the development of bots might be thought of as a different version of the Turing test (there is not, unfortunately, a Loebner prize available to one who implements a human-like bot).

4. CONSIDERATIONS

There are several factors that should be taken into account when developing a viable architecture to the control of a game character (note that here we consider primarily recent computer games such as FPSs, Role Playing Games – RPG and Real Time Strategy - RTS, not traditional games such as chess or go-moku). Central to this discussion is the playability of the game, which is usually directly influenced by the behaviour of the controlling intelligent strategy. An important aspect that defines playability is the concept of suspension of disbelief, that is, the ability the game has to bring the player inside its own “virtual world” which is “believable” according to its proposal and plausible regarding common sense. This concept is important not only in games, but also in movies and especially in animation, as Walt Disney so well executes it [7].

In a more detailed fashion, some of the aspects to be taken into consideration when developing a game AI engine are:

- **Performance**: although in recent years more and more clock cycles have been allotted to AI developers (especially with the appearance of dedicated graphics cards), performance still remains an issue, particularly in real-time games with many interacting AI engines.

- **Reactivity**: an AI engine should react in a timely manner to all stimuli coming from the world. Very few games allow for “philosopher” AI techniques, which take too long pondering upon their existence before taking an action.

- **Debug**: an AI engine should be easy to debug and to pinpoint errors, what in many occasions means “freezing” non-deterministic behaviours.

- **Internal states**: an AI generated behaviour is more readily accepted by the players and contributes to the credibility of the game (and thus to the suspension of disbelief) if it has the means to express internal states readily identifiable by the player (and act accordingly to them), fact which allows him (the player) to “anthropomorphize” the behaviour.

- **Serious bugs**: in order to maintain suspension of disbelief, it is better to have a consistent, mediocre AI strategy than to have a brilliant one which has a blatant flaw in certain situations.

- **Adaptivity**: an AI strategy which adapts itself to new situations and to the player’s gaming style gives a game extended replayability and enhances the gaming experience as a whole.

5. IMPLEMENTATION

Implementation was partitioned into two distinct phases. The first phase consisted of developing the C++ SNTool’s core engine, while the second phase consisted of implementing the control architecture using the framework implemented on the previous phase. All the development was made using Microsoft Visual C++, aided by a CASE (Computer Aided Software Engineering) tool (Rational Rose 2000).

The first phase (implementation of SNTool core engine) was implemented in accordance to the formal specifications presented on [3,4]. It is interesting to notice that although no graphical user interface was implemented, this newly implemented tool boasts several enhancements over its Java cousin, such as: - much better performance; - greater flexibility in code edition and debugging; - easier integration into external systems, easily accessing C++ or C systems. These advantages come mainly from the fact that SNTool C++ presents itself not as a complete development platform, but rather as a generic framework / tool designed to be embedded on other systems.

The second phase consisted in first defining a clear C++ interface to the generic control of bots, interface that we hope might be easily used on other FPS games with little or no
modification; second, on the implementation of the bots’ control architecture.

It is interesting to notice how the bot system interacts with the game engine proper. CS itself is not the game engine, being only a modification (mod) to the engine of the game Half-Life. A mod is essentially a software component packaged as a Windows DLL (Dynamic-Link Library) which is accessed by the real game engine, allowing the game to be customized. Functions then are called on this DLL and values are returned, the DLL being also able to make callbacks to the game engine. However, bots can only be implemented also through the use of DLLs, what present an interesting dilemma as we want to be able to use both CS’s DLL and the bots’ DLL. This was solved by [12] by implementing the bots’ DLL as a middle layer between the game engine and CS’s DLL, intercepting the relevant engine and DLL calls.

The implemented bot’s control architecture is based on the concept of behaviours introduced by the cognitive sciences community and already used in some other works [8,9,10]. A behaviour here is understood as any concept leading to an action, from very high level concepts to very low level ones. Because behaviours map directly to abstract concepts commonly used by human beings, the use of behaviours make the design of these systems extremely intuitive, easing the game designer tasks of designing and debugging the system.

Behaviours are structured hierarchically (similar to what is used in [11]), with the most “high-level” ones at the top and the most “low-level” ones at the bottom. Behaviours can thus be viewed as a tree consisting of parent and children behaviours. At the top of this tree are the sensors, which provide the necessary information to determine which top level behaviour should be activated. The activation of behaviours goes on descending the tree until a tree’s leaf fires some (or no) actuators. Only one child behaviour may be active at a given time step.

An interesting aspect of the architecture used here is the adoption of emulated “emotions”. Note that here we do not intent to simulate emotions, but to emulate them in order to allow a bot to appear more human-like to the player. Also the emotion architecture used is a rather simplistic one; for more elaborate ones see [15,16,17]. We adopt a discrete emotion system as proposed by [14], each emotional state being

![Figure 4. The complete network used to control the bot.](image)
represented by a real variable. These variables are characterized by having a decay rate, a rest state and an update rate (which is determined according to the bot’s environment and internal state, such as number of enemies around, health, etc). This model is inspired on the one proposed by Picard in [18], and has proven sufficient for our immediate needs.

Formally, the behaviour update equation is:

$$e_{j+1} = e_j - a_0 + b_0 \beta(\gamma)$$

restricted by the condition that:

$$\hat{e}_{j+1} \leq l_0 \text{ then } e_{j+1} \leftarrow l_0$$

where $\hat{e}_j$ is the behaviour value at time step j, $a_0$ is a constant decay rate, $b_0$ is a constant that regulates the impact $\gamma$ has on the next state, and $l_0$ is the minimal value allowed for the emotion (this allows to have for example naturally courageous bots). $\gamma$ is a value obtained arbitrarily from the bot’s internal state, such as the variation of the bots’ health value. Note that although the emotion decays linearly, we could easily adopt another decay value to have for example an exponential decay rate.

The next equation, a sigmoid function, reflects the effect the environment and the internal variables have on the emotion value:

$$\beta(\gamma) = \frac{1}{1 + e^{-a_0 \gamma + d_0}}$$

where $a_0$ is a constant regulating the inclination of the sigmoid and $d_0$ is another constant that represents the threshold value from which $\gamma$ starts to have an effect over the emotion.

It is also interesting to notice that emotions, together with other “drives” the bot may have, constitute a central piece if we futurely intend to implement a learning architecture. This important role played by emotions and drives (both of which may be thought of as special internal variables) has already been acknowledged by Damásio [19], and is used to drive learning in the work of Blumberg [10] and Grand and Cliff [20].

It is also interesting to notice here how the bot’s navigational system is implemented. The navigational system uses waypoints, a commonly adopted way of controlling bots in maps. These waypoints provide bots with points and paths which it can follow, either randomly, or towards specific targets. Waypoints can be assigned flags which indicate special locations, such as a bomb site, a lift, a hostage rescue area, a sniping spot, etc. They have the clear advantage of simplicity; for a given map properly waypointed navigation is easily done, and the bot has a wealth of information readily available. Also waypoints are usually less time consuming than other navigational schemes, because we can compute all the paths leading from one waypoint to another at startup time, using for example Floyd’s algorithm [13].

An execution cycle for the architecture proceeds as follows: first, the sensors sense the environment and internal variables, and the bot’s emotions are updated. Then the high-level behaviours assess their importance, represented by the utility value returned by the match operator. The engine then fires the relevant behaviour by calling its process operator. This match-process cycle is repeated until a behaviour fires an actuator, which is responsible for taking an action on the environment (or even over emotions, as a special “internal actuator”).

The semionic network implemented is composed of 35 places, 21 of which are active places. The full network may be viewed on figure 4. The description of these active places follows:

- **EnvSensor**: sensor responsible for getting information from the environment, providing the bot with all the information it needs such as friends, enemies, waypoints, obstacles, etc.
- **InternalSensor**: sensor responsible for sensing the internal state of the bot, such as its health, armour, ammo, weapons, etc.
- **EmotionsManager**: this place updates the emotional states of the bots according to what was previously explained. Right now only one emotion is used, *fear*, but new ones can be easily added as necessary.
- **Stuck**: a very low level behaviour, responsible for getting the bot out of situations when it is stuck in walls or other bots.
- **Flee**: behaviour which dictates that a bot should flee. Is altered by the *fear* emotion.
- **RunForYourLife**: a child behaviour to *Flee*, determines that a bot should turn away from its enemies and simply run as fast as it can.
- **Retreat**: also a child behaviour to *Flee*, determines a backwards retreat from the enemy.
- **Fight**: a top level behaviour that determines when a bot should fight. Also influenced by the *fear* emotion.
- **Charge**: behaviour that represents a straight on attack.
- **Wander**: behaviour that makes the bot wander aimlessly.
- **GotoWpt**: behaviour that takes a bot to a given waypoint.
- **PlantBomb**: behaviour that determines that a bot should plant a bomb.
- **PutBomb**: behaviour that puts a bomb at a bomb site.
- **GuardBomb**: behaviour to guard a bomb that has been planted.
- **Strafe**: actuator that makes a bot move sideways.
- **Jump**: actuator that makes a bot jump.
- **Turn**: actuator that makes a bot turns.
- **Move**: actuator that makes a bot move forwards or backwards.
- **Shoot**: actuator that makes a bot shoot.
- **SelectWeapon**: actuator that selects a weapon (or item).
- **Crouch**: actuator that crouches the bot.
It should be noted that the implemented bot only plays adequately as a terrorist on maps that have bombing a target as their objective (of course the bot may always play a free-for-all game, that is, simply kill its opponents).

6. **CONCLUDING REMARKS**

The architecture presented on this paper is simple albeit effective regarding both its computational performance and its overall game behaviour. The use of SNTool and semionic networks theory constitute doubtlessly an efficient formalism to the structuring of the system, however imposing sometimes some limitations, especially concerning the structured organization of signlets inside places.

Of course the architecture herein implemented has a number of shortcomings; specifically it implements no adaptation, and its behaviours are static and non-configurable. Ideally all behaviours should be equal, differing only on their parameterization, what could allow for a learning mechanism similar to the one found in [10], which allows not only learning by adjusting a behaviour importance but also by creating new ones when necessary. Other improvements yet to the bot architecture include better teamwork play, communications and the ability to play with any type of objective on any team.

7. **REFERENCES**


