UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

HUGO DA SILVA CORRÊA PINTO

Designing Autonomous Agents for Computer Games with Extended Behavior Networks: An Investigation of Agent Performance, Character Modeling and Action Selection in Unreal Tournament

Thesis presented in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Prof. Dr. Luis Otavio Alvares
Advisor

Porto Alegre, June 2005.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pinto, Hugo da Silva Correa

Designing Autonomous Agents for Computer Games with Extended Behavior Networks: An Investigation of Agent Performance, Character Modeling and Action Selection in Unreal Tournament. / Hugo da Silva Corrêa Pinto – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

78 f.:il.

Dissertation (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2005. Orientador: Luis Otavio Alvares.

1.Behavior Network 2.Computer Games 3.Character Modeling. I. Alvares, Luis Otavio II. Título.

ACKNOWLEDGEMENTS

Luis Otavio Alvares was a mentor in academic life, a friend in tough times and a guide into a different culture. His support in all these fronts had a great impact in this work.

The comments of Ana Bazzan and Paulo Engel during "Semana Academica" helped shape the thesis in its early stages.

The anonymous reviewers of SBGames 2004, ENIA 2005 and IVA 2005 made relevant suggestions, mostly incorporated in this work.

Professor Klaus Dorer's e-mail discussions and sharing of C++ code accelerated my understanding and Java implementation of extended behavior networks.

Without the patches of Joe Manojlovich and Jessica Bayliss, Gamebots use would have been much tougher.

My parent's support was absolutely crucial in the years of development of this work.

Cassio Pennachin's and Vetta Labs's flexibility in the last months of this thesis were also fundamental.

The Bonneti family has welcomed me warmly at my arrival at Porto Alegre, helping to make my adaptation easy.

The logistic support of Fernando Machado, Marco Aurelio Wehrmeister, Jorge Jesus, Aurelio Dias and Oscar Calcin made my academic life much easier.

I thank my II-UFRGS colleagues for the academic discussions, tips on department procedures and other unwritten relevant knowledge, specially João Valiati and Sandro Camargo.

The support staff, particularly Luis Otavio Soares, Paulo Jesus and Ângela Silva, has made a good job these years, making administrative mumbo-jumbo almost invisible to the alumni. They have my sincere respect and gratitude.

Profs. Soraia Musse, Luciana Nedel and Paulo Engel made suggestions and remarks during my defense that were mostly incorporated in this final version.

I thank my wife Liliane for the logistic, emotional an intellectual support. Her patience and day-to-day management during critical weeks was remarkable.

Last, but not least, I thank Newton Vieira for his support during my undergraduate years and his encouragement for pursuing graduate studies.

This work was made financial support of Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq.

I Have Come Into Bein**g**

by the Pro**c**ess of my Comi**n**g Into Being

the Process of Co**m**ing Into **B**eing is Established

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

EBN	Extended Behavior Network
ASM	Action Selection Mechanism

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This work investigates the application of extended behavior networks to the computer game domain. We use as our test bed the game Unreal Tournament.

Extended Behavior Networks (EBNs) are a class of action selection architectures capable of selecting a good set of actions for complex agents situated in continuous and dynamic environments. They have been successfully applied to the Robocup, but never before used in computer games.

PHISH-Nets, a behavior network model capable of selecting just single actions, was applied to character modeling with promising results. Although extended behavior networks are applicable to a larger domain, they had not been used to character modeling before.

We present how to design an agent with extended behavior networks, fuzzy sensors and finite-state machine based behaviors.

We investigate the quality of the action selection mechanism and its correctness in a series of experiments.

The performance is assessed comparing the scores of an agent using an extended behavior network against a plain reactive agent with identical sensory-motor apparatus and against a totally different agent built around finite-state machines.

We investigate how EBNs fare on agent personality modeling via the design and analysis of five stereotypes in Unreal Tournament. We discuss three ways to build character personas and situate our work within other approaches.

We conclude that extended behavior networks are a good action selection architecture for the computer game domain and an interesting mechanism to build agents with simple personalities.

Keywords: behavior networks, computer games, character modeling, autonomous agents, planning, action selection, personality, Unreal Tournament.

Construção de Agentes Autônomos para Jogos de Computador com Redes de Comportamentos Estendidas: Uma investigação de seleção de ações, performance de agentes e modelagem de personagens no jogo Unreal Tournament.

RESUMO

Este trabalho investiga a aplicação de rede de comportamentos estendidas ao domínio de jogos de computador.

Redes de comportamentos estendidas (RCE) são uma classe de arquiteturas para seleção de ações capazes de selecionar bons conjuntos de ações para agentes complexos situados em ambientes contínuos e dinâmicos. Foram aplicadas com sucesso na Robocup, mas nunca foram aplicadas a jogos.

PHISH-Nets, um modelo de redes de comportamentos capaz de selecionar apenas uma ação por vez, foi aplicado à modelagem de personagens, com bons resultados. Apesar de RCEs serem aplicáveis a um conjunto de domínios maior, nunca foram usadas para modelagem de personagens.

Apresenta-se como projetar um agente controlado por uma rede de comportamentos para o domínio do Unreal Tournament e como integrar a rede de comportamentos a sensores nebulosos e comportamentos baseados em máquinas de estado-finito aumentadas.

Investiga-se a qualidade da seleção de ações e a correção do mecanismo em uma série de experimentos.

A performance é medida através da comparação das pontuações de um agente baseado em redes de comportamentos com outros dois agentes. Um dos agentes foi implementado por outro grupo e usava sensores, efetores e comportamentos diferentes. O outro agente era idêntico ao agente baseado em RCEs, exceto pelo mecanismo de controle empregado.

A modelagem de personalidade é investigada através do projeto e análise de cinco estereótipos: Samurai, Veterano, Berserker, Novato e Covarde. Apresenta-se três maneiras de construir personalidades e situa-se este trabalho dentro de outras abordagems de projeto de personalidades.

Conclui-se que a rede de comportamentos estendida é um bom mecanismo de seleção de ações para o domínio de jogos de computador e um mecanismo interessante para a construção de agentes com personalidades simples.

Palavras-Chave: redes de comportamentos, jogos de computador, modelagem de personagens, agentes autônomos, seleção de ações, personalidade, Unreal Tournament.

1 INTRODUCTION

In the design of a game robot one of the key concerns is how it selects its actions so as to exhibit a goal-oriented behavior. When the robot has many possibly conflicting goals this task gets more complicated. If the robot is also in a fast changing environment and has to consider many factors at each instant we have a hard problem to tackle. Search-based approaches turn impractical due to the large search space and traditional planning is made much harder as the environment may have changed when the agent finishes planning.

Behavior Networks (MAES, 1989) were proposed as an action selection mechanism to select good actions in complex and dynamic environments. They favor actions that contribute to more than one goal and those that are part of an ongoing plan. They gracefully treat conflicts among the goals and are fast, robust and reactive.

Extended Behavior Networks (DORER,1999-a) are an extension for continuous domains capable of selecting actions concurrently and specifying situation-dependent goals. They were applied to the Robocup[1] with very good results and to the game Unreal Tournament (PINTO et al., 2005-a)[2], again with encouraging findings.

Good action selection is important, but how the agent selects its actions and how it affects its personality is also a concern in a computer game. Wouldn't it be interesting if while building an agent towards proper goal-oriented behavior we could take into consideration personality traits? Behavior Networks enable one to do just that. Rhodes (1996) has applied a behavior network model, PHISH-NET, to the design of character personalities and Pinto (2005-b) has applied extended behavior networks to the design of stereotypes.

Unreal Tournament is a modern 3D action game. In this game genre we have the agent situated in a 3D continuous virtual environment, interacting in many ways with several entities in real-time. The scenarios an agent may face are varied and complex. The agent has many weapons available, each with certain properties and several items to use. It moves over different landscapes and interacts with several other agents, both opponents and teammates. The action repertory is large (run, walk, turn, crawl, shoot, change weapons, jump, strafe, pickup item and use item among others) and an agent

---

[1] See (DORER, 1999) and (DORER, 2004). The Magma-Freiburg team, built using extended behavior networks, was the vice-champion of Robocup-1999.

[2] We must clarify that extended behavior networks were used in the game Unreal Tournament in our research, not by the developers of the game. The game Unreal Tournament makes no use of Extended Behavior Networks as far as we know.

may carry out more than one action simultaneously, such as shooting while jumping. Also, the agent has many possibly conflicting goals, such as fighting and keeping its safety.

We see that this domain provides a challenging scenario for an action selection mechanism. An agent has to deal with continuous measures, the combination of actions grows exponentially and planning has to deal with fast changes in its conditions.

In fact, modern computer games in general, and 3D action games in particular, offer an interesting research domain for artificial intelligence. Laird (2000) goes so far as to regard computer games as the contemporary "AI killer application".

Figure 1.1 shows game genres and the AI problems they pose.

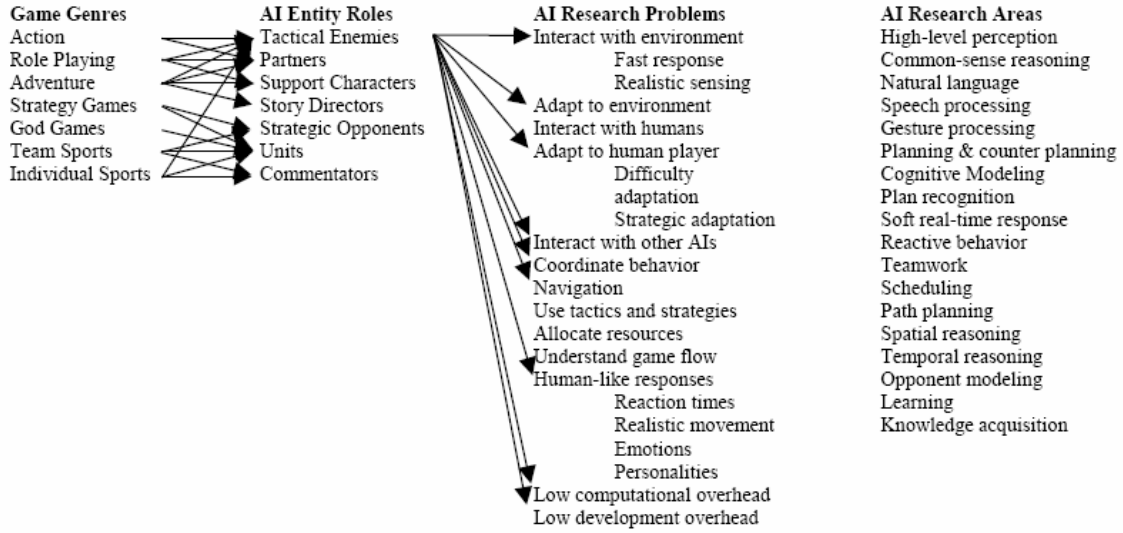

Figure 1.1: Game Genres, AI Roles and Research Problems. Reproduced from (LAIRD, 2000).

We see that these games provide domains to investigate difficult AI problems, such as reasoning with limited resources and spatial reasoning, for a very low price. The extensive testing of the games and their huge user base turn computer games into an interesting test bed for AI techniques and theories.

A word of caution is due: As Nareyek (2004) points out, many of the built-in AI interfaces for these games do not provide the necessary flexibility to do some kinds of research. Nonetheless there are many investigations possible with the current games and interfaces, such as the present work.

From an engineer's point of view these games have become an important application in itself. The computer game industry has beaten the movies industry in the USA, regarding revenue, for 2 consecutive years (ESA, 2005). The governments of Australia, Brazil and South Korea have special programs supporting the research and development of computer games (ABRAGAMES, 2004). Added to this hype is the opportunity - many problems posed by computer games resemble the problems faced in robotics and other artificial intelligence fields a decade ago. It is time to not only design new solutions, but to see if old solutions apply to these new but similar problems and adapt them to these new domains.

Nareyek (2004) points that most games made use of few AI paradigms and techniques well established in academia, with a predominance of finite-state machines, A* and scripting. One reason is that only recently game developers have enough processing power for sophisticated techniques, as graphics cards carry most of the graphic processing load, freeing up memory for AI. The other is that if graphics were the main competitive aspect of a game back in the 90's, nowadays AI has become one of the great divides. Good graphics are assumed as certain, while good AI still amazes a player.

In the last few years, interest in the application of sophisticated AI to games has increased, with a boom in game AI literature from 2002 on, as evidenced by (RABIN, 2002), (BUCKLAND, 2002), (RABIN, 2003) and (CHAMPANDARD, 2004). Games such as *Black and White*[3] not only employed adaptive AI but also based a great deal of its marketing on the intelligence of its game creatures.

The similarity of robotics and the action game domain has been explored in (CHAMPANDARD, 2004) and (YISKIS, 2003). The first presents a framework (FEAR, 2005) and an overview of techniques for building reactive agents with learning capabilities in the game Quake II. The later describes how to apply the subsumption architecture (BROOKS, 1986) on games, focusing on its integration with character animation.

Now we clearly see the contribution of the present work to the game AI engineering community – extended behavior networks are a well-tested, well-understood, simple and successful technique from robotics that can be applied with good results in the game domain[4].

In this thesis we investigate the design of agents with personality for the game Unreal Tournament. We evaluate the quality of the action selection and the performance of the agents in the game, and investigate ways to endow them with different personalities using three different approaches.

From a purely scientific stance, our investigation of extended behavior networks in the game domain adds to the understanding of various issues: How extended behavior networks fare in the computer game domain? Are EBNs generally a good action selection mechanism for complex and dynamic environments with continuous properties? Do the properties observed in the Robocup domain show up in other scenarios? How to evaluate the behavior and performance of a complex agent? How to integrate sensors, behaviors and decision making into a complete agent? How to build a behavior-network based agent?

These questions will be addressed in the following chapters, particularly chapter 5.

Chapter two gives a detailed exposition of the extended behavior network model and the game Unreal Tournament.

In chapter three we see in detail the architecture of the agent used in most experiments. We show the rationale behind our design decisions, how we built the

---

[3] EIDOS Interactive.

[4] As shown by (PINTO and ALVARES, 2005a) and (PINTO and ALVARES, 2005b) and the rest of this thesis.

behavior network, the agent's sensors and the behavior modules, how the sensory-motor apparatus is integrated to the network and how our agent is coupled to the game.

Chapter four presents the experiments carried out to assess action selection quality, agent performance and personality modeling.

In the fifth chapter we discuss the results of the experiments and contextualize them with a body of related work, when relevant. Also, we point extensions to our work and discuss the applicability and scope of this thesis.

Chapter six presents our conclusions and future work.

2 BACKGROUND

This chapter presents the fundamental background for the rest of this work.

The first section presents the extended behavior network architecture in detail, its structure and action selection algorithm.

The game Unreal Tournament and the add-on we used to build our agent are presented and discussed in the second section.

2.1 Extended Behavior Networks

Behavior Networks are a class of action selection architectures for selecting good enough actions[5] in dynamic and complex environments. They combine properties of traditional planners (chaining of actions based on preconditions and effects) and connectionist systems (activation spreading). They are defined by a static structure and an action selection algorithm.

The structure of a behavior network is composed of a set of behavior modules, a set of goals, a set of links that join modules to goals and other modules and a set of control parameters. A behavior module resembles a STRIPS[6] operator, having lists with both the expected effects of its action execution and preconditions for it becoming active. The links are made based on the effects of the modules and the conditions of modules and goals.

Action selection is based in the mutual excitation and inhibition among the network nodes, via activation spreading.

Behavior Networks have been constantly evolving since their first appearance (MAES, 1989), as shown by (TYRRELL, 1993), (RHODES, 1996), (GOETZ, 1997), (DORER, 1999-a) and (NEBEL and BABOVICH, 2003). They have been applied to animal simulation (TYRRELL, 1993), interactive storytelling (RHODES, 1996) and the Robocup (MÜLLER, 2001).

---

[5] Maes (1989) defined a good enough action selection policy as one that has the following characteristics: favors actions that contribute to the agent's goals (specially several goals at once), favors actions that contribute to ongoing plans, exploit opportunities, is fast, is robust and avoids conflicts among objectives.

[6] See (NILSSON, 1971).

Maes' (1989) behavior network was capable of selecting just one action at each cycle and its conditions and effects were boolean-valued. Tyrrell (1993) discovered many problems in the activation spreading of this model, which were addressed in subsequent architectures. The PHISH-Net of Rhodes (1996) and the Extended Behavior Network of Dorer (2004) are the current state-of-art of behavior network architecture (PINTO, 2004).

The PHISH-Net was made focusing on the control of characters in interactive storytelling environments. Its distinguishing features are its use of variables and its mechanisms to treat action failures, loops and impossible-to-satisfy modules. Like Maes' model, PHISH nets also have boolean-valued conditions and effects and select only one action at each cycle.

The Extended Behavior Network was designed with the goal of controlling agents in the Robocup (DORER, 2004). It uses real-valued propositions for effects and conditions, allows the specification of situation-dependent goals and selects actions concurrently. This last feature is what makes extended behavior networks suit to our domain of concern and what precludes other models of immediate applicability. Concurrent action selection is a must for most contemporary computer games.

2.1.1 Structure

An extended behavior network (EBN) is defined by a set of behavior modules (M), a set of goals (G), a set of sensors (S), a set of resources (R) and a set of control parameters (C). Figure 2.1 shows the specification of part of a behavior network used in our experiments and figure 2.2 the network built from this specification.

```
Module
precondition EnemyInSight
action ShootEnemy
effects EnemyHurt 0.6 LowAmmo 0.1
using Hands 2 Head 1  endModule
Resource
name Legs
amount 2 endResource
Resource
name Head
amount 1 endResource
Resource
name Hands
amount 2 endResource
```

```
Goal
condition EnemyHurt
strength 0.8
context endGoal
Goal
condition Not LowAmmo
strength 0.6
context  LowAmmo endGoal
Parameters
name ActivationInfluence value 1.0
name InhibitionInfluence value 0.9
name Inertia value 0.5
name GlobalThreshold value 0.6
name ThresholdDecay value 0.1
endParameters
```

Figure 2.1: Specification of a simple behavior network.

A goal *i* is defined by a proposition that must be met (*Gi*), a strength value (*Sti*) and a disjunction of propositions that provide the context for that goal, called the relevance condition (*Li*). The strength provides the static, context-independent importance of the goal and the relevance condition the dynamic, context-dependent one.

The use of two kinds of conditions in the goals enables us to express goals that become more or less important depending on the situation the agent is in.

Maintenance goals are those that preserve some state of the agent, and become ever more important as the current state diverges from the goal. The trivial condition for a maintenance goal is the negation of the goal condition. Goal *Not LowAmmo* in figures 2.1 and 2.2 is an example. The importance of not being low in ammunition increases as the agent's ammo drops.

Achievement goals represent the increasing motivation to reach a certain state, becoming more relevant the more the current state approaches the goal. The trivial condition for this kind of goal is the condition of the goal itself.

A context independent goal is modeled leaving it without relevance conditions. Goal *EnemyHurt* in figure 2.1 is an example of such a goal. Note that a goal without relevance conditions amounts to a goal that is always relevant, i.e., its relevance is always maximal.

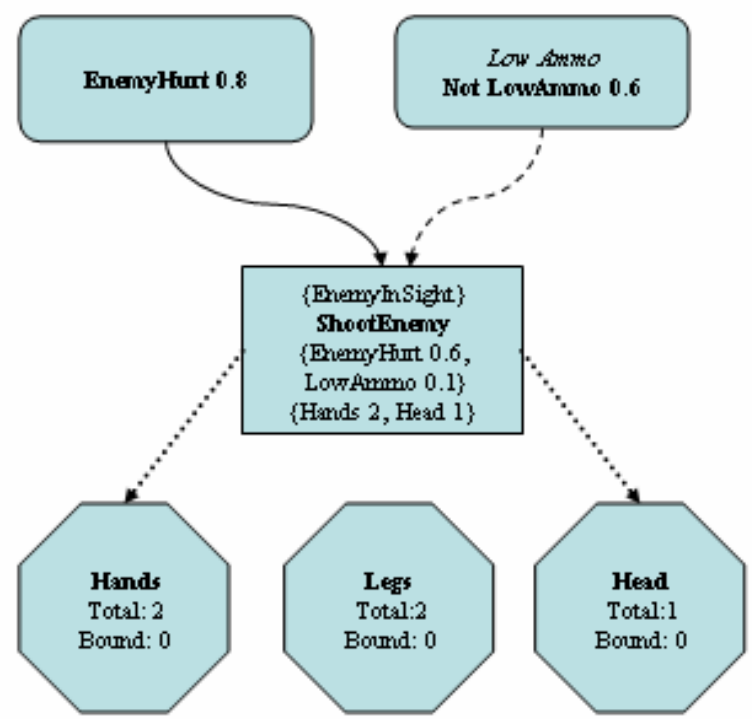

Figure 2.2: Simple Behavior Network Diagram. The goals are represented by round cornered rectangles, the behaviors by sharp cornered rectangles and the resource nodes by octagons. Straight lines represent predecessor links, dashed lines conflict links and pointed lines resource links. The directions of the arrows indicate the activation flow, for predecessor and conflict links. In resource links they indicate the module's dependence on the resource.

Each behavior module is specified by a conditions list, an action, an effects list and a resources list. The first list is a conjunction of real valued propositions that represent the needed conditions for the module to execute. The effects list is a conjunction of propositions (each possibly negated) whose values are the values that we expect them to have after the module's action execution. The resources list is made of pairs (resource, amount), each indicating the expected amount of a resource an agent uses to perform the action. Let us examine behavior *ShootEnemy* in figure 2.1. We see that the precondition

is that there is an enemy in sight {*EnemyInSight*}. The expected effects are that the enemy will become hurt with 60% chance (or, conversely, that *EnemyHurt* verity will be 0.6) and that the agent will be with low ammo with 10% chance {*EnemyHurt* 0.6, *LowAmmo* 0.1}. It needs both hands and its head to perform the behavior {Hands 2, Head 1}.

Goals and modules are linked with two kinds of links. Predecessor links go from a module or goal B to a module A, for each proposition in the condition list of B that is in the effects list of A such that the proposition has the same sign (true + and false -) in both ends of the link. The link from goal *EnemyHurt* to module *ShootEnemy* in figure 2.1 is an example. Conflict links go from a module or goal B to a module A, for each proposition in the condition list of B that is in the effects list of A such that the proposition has opposite signs at either end of the link. In figure 2.1, the link from *Not LowAmmo* to *ShootEnemy* is a conflict link. Conflict links take energy away from their targets and predecessor links input energy to their targets. This way a module or goal tries to inhibit modules whose execution would undo some of its conditions and attempts to bring into execution modules whose actions would satisfy any of its conditions.

Each resource is represented by a resource node and defined by a function *f(s)* that specifies the expected amount of the resource available in each situation *s*. In addition to *f(s),* each node has a variable *bound* that keeps track of the amount of bound resources and a resource activation threshold $\theta_{Res} \in (0..\theta]$, where θ is the global activation threshold. In figure 2.1 we see that the expected used amount of each resource is constant for all situations. This is not surprising as our agent has the same number of body parts available in any situation (the game does not account for limb loss or similar gruesome events).

The modules are linked to the resource nodes through resource links. For each resource type in the resources list of a module there is a link from the module to the corresponding resource node.

The control parameters are used to fine tune the network and have values in the range [0, 1]. The activation influence parameter γ controls the activation from predecessor links. Inhibition influence, δ, the negative activation from conflict links. The inertia β, the global threshold θ and the threshold decay $\Delta\theta$ have their straightforward meanings. Their function will become clearer in the next subsection. These parameters enable us to influence the degree of persistence of the agent (the higher the inertia the greater the persistence) and how reactive it is (the greater the global threshold the longer the sequence of actions considered when selecting a module for execution), among other properties. Default parameters that work well under various circumstances for the Robocup domain are shown in (DORER, 1999-c).

2.1.2 Action Selection Algorithm

The modules to be executed at each cycle are selected in the following way:

1) The activation *a* of each module is calculated.

2) The executability *e* of each module is calculated using some triangular norm operation[7] over its condition list.

3) The execution-value *h*(*a*,*e*) is calculated by multiplying *a* and *e*. Note that this value combines the utility of executing a behavior (activation) and the probability of executing it successfully (executability). This way even modules with conditions not much satisfied may execute if they have high activation.

4) For each resource used by a module, starting by the last non-available resource, the module checks if it has exceeded the resource threshold and if there is enough of that resource for its execution. If so, it binds the resource.

5) If a module has bound all of its needed resources it executes and resets the resources thresholds to the value of the global threshold.

6) Each module unbinds the resources it used.

The thresholds of the resources linearly decay over time, ensuring that eventually a behavior will be able to bind its needed resources and that the most active behavior gets priority.

The formulae of Figure 2.3 detail the activation spreading process.

$$a^t_{kg_i}{}' = \gamma \cdot f(\iota_{g_i}, r^t_{g_i}) \cdot ex_j \qquad (1)$$

$$a^t_{kg_i}{}'' = -\delta \cdot f(\iota_{g_i}, r^t_{g_i}) \cdot ex_j, \qquad (2)$$

$$a^t_{kg_i}{}''' = \gamma \cdot \sigma(a^{t-1}_{succ\,g_i}) \cdot ex_j \cdot (1 - \tau(p_{succ}, s)), \qquad (3)$$

$$a^t_{kg_i}{}'''' = -\delta \cdot \sigma(a^{t-1}_{conf\,g_i}) \cdot ex_j \cdot \tau(p_{conf}, s), \qquad (4)$$

$$a^t_{kg_i} = abs\max(a^t_{kg_i}{}', a^t_{kg_i}{}'', a^t_{kg_i}{}''', a^t_{kg_i}{}''''). \qquad (5)$$

$$a^t_k = \beta a^{t-1}_k + \sum_i a^t_{kg_i} \qquad (6)$$

Figure 2.3: Activation Spreading Formulae (DORER, 1999-b).

Formula (1) shows the activation that goes from a goal *i* to a module *k* through a predecessor link at instant *t*. Function *f*[8] is a triangular norm that combines the strength (l_{gi}) and the dynamic relevance of a goal (r^t_{gi}). The term ex_j is the value of the effect proposition that is the target of a link.

Formula (2) shows the activation that goes from a goal *i* to a module *k* through a conflict link at an instant *t*.

---

[7] A mapping T: $[0,1]\times[0,1] \rightarrow [0,1]$ is a triangular norm (t-norm) if and only if it is symmetric, associative, non-decreasing in each argument and T (*k*,1) = *k* for all *k* in [0,1]. In (DORER, 2004) and (PINTO, 2005a) the t-norm used was multiplication.

[8] In (DORER, 2004) and (PINTO, 2005a) the t-norm used was multiplication too.

Formula (3) shows the activation spreading from a module *succ* to a module *k* at an instant *t* through a predecessor link. p_{succ} is the proposition of the successor module and a_{succ} the activation of the successor module. $\tau(p_{succ}, s)$ is the value of p_{succ} in situation *s*. We see that the activation spreading increases as the proposition at the start of a predecessor link becomes less satisfied. Thus, we can see unsatisfied conditions as increasingly demanding sub-goals of the network. Function σ[9], shown below, is used to make the behavior modules strong attractors with a high probability. This reduces unnecessary behavior switches, as small changes in the percepts will be less likely to disrupt an ongoing behavior. Goetz (1997) shows that using formula (7) we do not have to normalize the total network activation, as was needed in Maes (1989).

$$\sigma(x) = (1 + e^{\kappa(\mu - x)})^{-1} \quad (7)$$

Formula (4) describes the activation spread from a module through a conflict link. a_{conf} and p_{conf} stand for the activation and proposition of the module that is the source of the conflict link, respectively.

Formula (6) shows that the activation of a module *k* at an instant *t* is its activation in the previous time step *t-1* weighted by the inertia constant β plus the sum of the activations retained of each goal *i*.

Formula (5) shows that a module retains just the activation of greatest absolute value from each goal. It amounts to keeping only the strongest path from a module to each goal, if we pay attention to the whole network.

2.1.3 Example

In this subsection we provide a step-by-step exposition of activation flow and action selection. We make a simplification to allow the reader to easily follow the values in the activation flux: instead of using the transfer function (7) when transmitting activation between modules we pass the activation unmodified, that is, we use the identity function.

Suppose we have the network of figure 2.4 with initial state and parameters as shown in figure 2.5.

---

[9] The use of this function in the activation spreading of behavior networks was first proposed by Goetz (1997). The details of the motivation for its use, based on dynamic systems theory and an analogy with Hopfield networks, is beyond the scope of our work. The interested reader is referred to (GOETZ, 1997), which offers a detailed and didactic explanation.

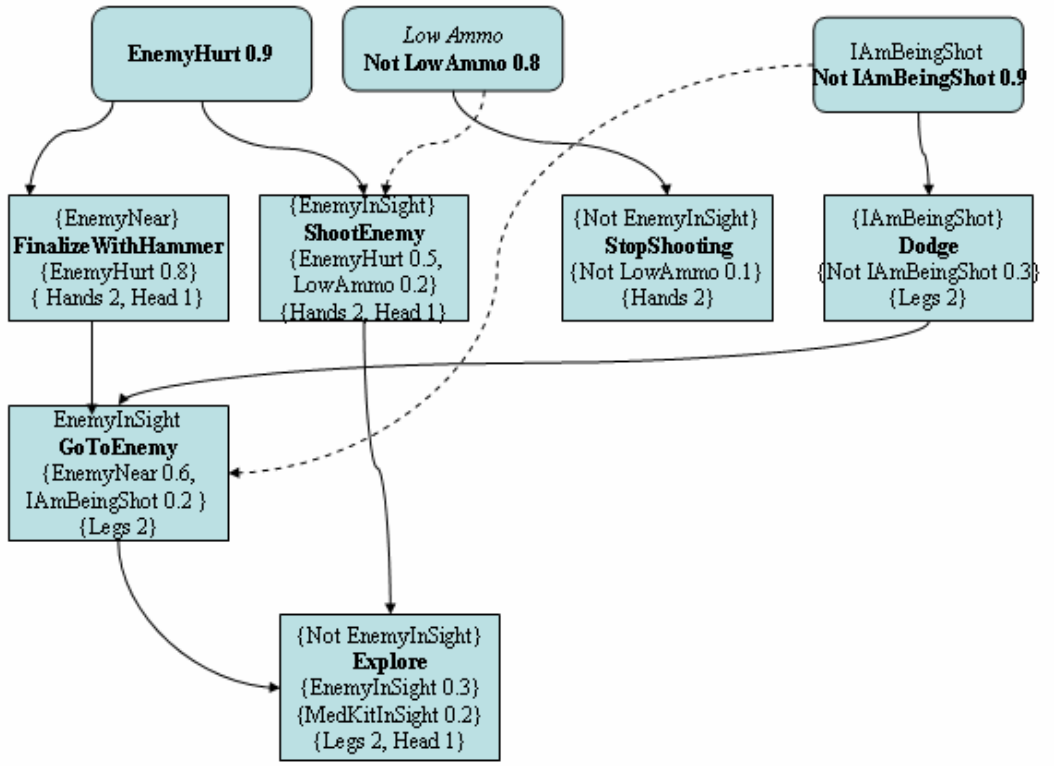

Figure 2.4: Example Behavior Network. Resource nodes are omitted for clarity.

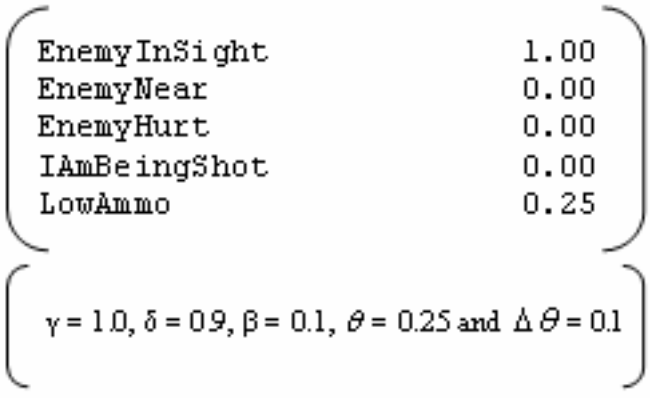

Figure 2.5: Initial state propositions and control parameters of example behavior network

These values lead to the goal importances shown in figure 2.6.

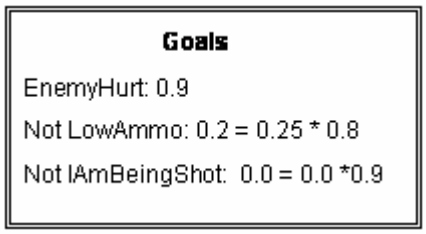

Figure 2.6: Example Goal Importances. The values in the right-hand side of the equations of *Not LowAmmo* and *Not IAmBeingShot* correspond the dynamic and static importances of each goal, respectivelly.

In figure 2.7 we see the first step of activation spreading.

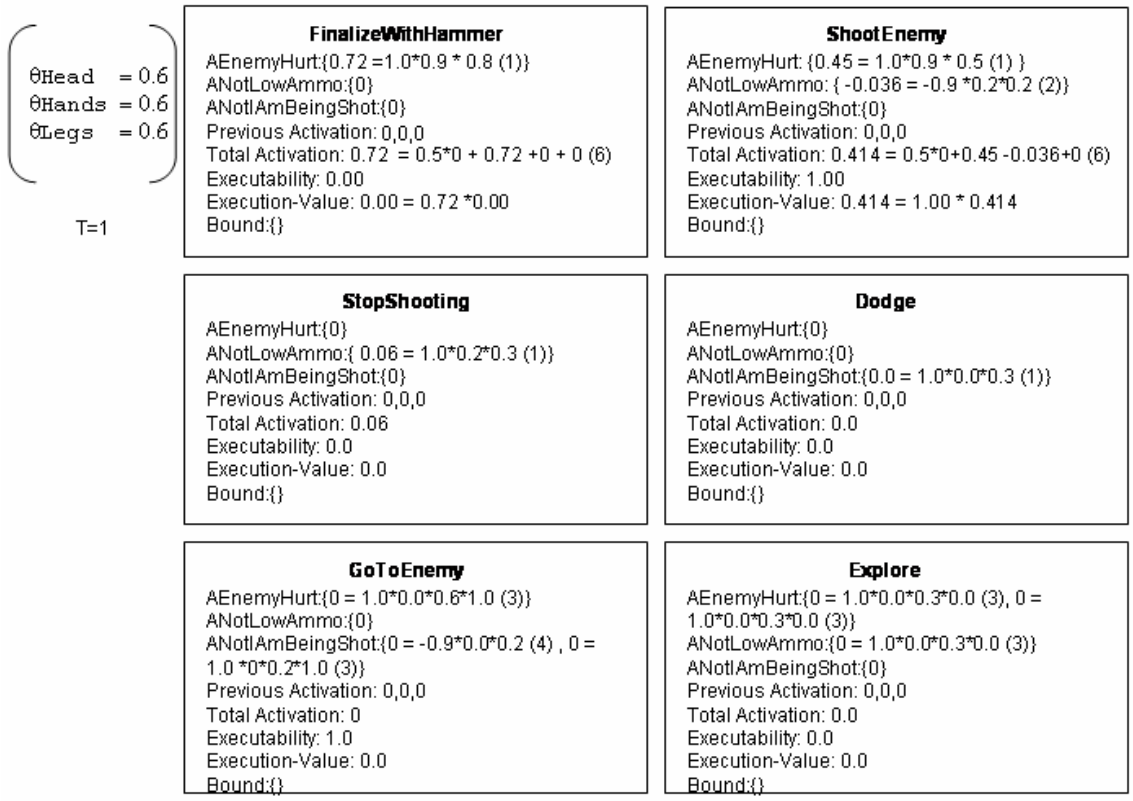

Figure 2.7: First step of activation spreading. The numbers in parenthesis indicate the formula used to calculate the activation value. Brackets indicate that more than one value is possible.

AEnemyHurt, ANotLowAmmo and ANotIAmBeingShot keep the activations relative to goals *EnemyHurt*, *Not LowAmmo* and *Not IAmBeingShot*, respectively. Note that *Explore* receives activation from both *GoToEnemy* and *ShootEnemy.* The first value in AEnemyHurt is from *ShootEnemy*, the second comes from *GoToEnemy*. The parenthesis at the right of a value illustrates the value used to achieve that value. Note that *1.0*0.9*0.8*, in AEnemyHurt of behavior *FinalizeWithHammer,* corresponds to (1), the formula governing activation spreading from goals to modules. In behavior *GoToEnemy*, at its AEnemyHurt field, we see that the product in its parenthesis illustrate the values used with formula (3) to calculate the activation received from module *FinalizeWithHammer*.

We can see examples of other formulae in use. In behavior *ShootEnemy* we see that the execution-value is obtained multiplying its executability and activation.

Examining the execution-values we see that no module is allowed to bind a resource at this step, as no execution-value has exceeded a resource threshold. Thus we proceed to the next spreading step, illustrated in figure 2.8

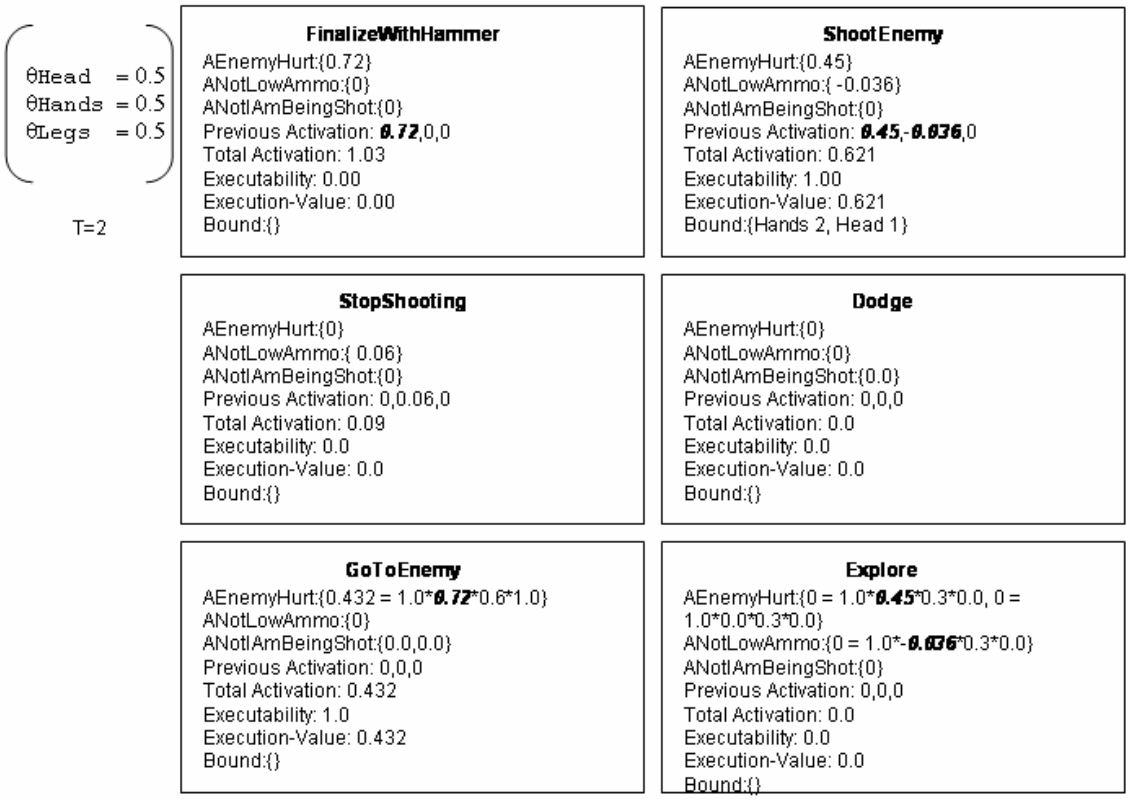

Figure 2.8: Second step of activation spreading. Bold is used to highlight the fact that the activation values used received by a module are those of the source module's previous step.

We see that now ShootEnemy has exceeded the thresholds of the Head and Hands resources, so it binds them. The activation received by a module is that of the source module's previous step. Examples are *GoToEnemy*, that receives activation from *FinalizeWithHammer* and *Dodge*(in fact it receives zero activation from *Dodge* at this step), and *Explore*, that receives activation from *ShootEnemy* and *GoToEnemy*. Note that activations relative to different goals are always kept separate, at all modules, at all steps.

At this step only *ShootEnemy* binds, resets the thresholds of the resources it uses and executes. Figure 2.9 illustrates the next step.

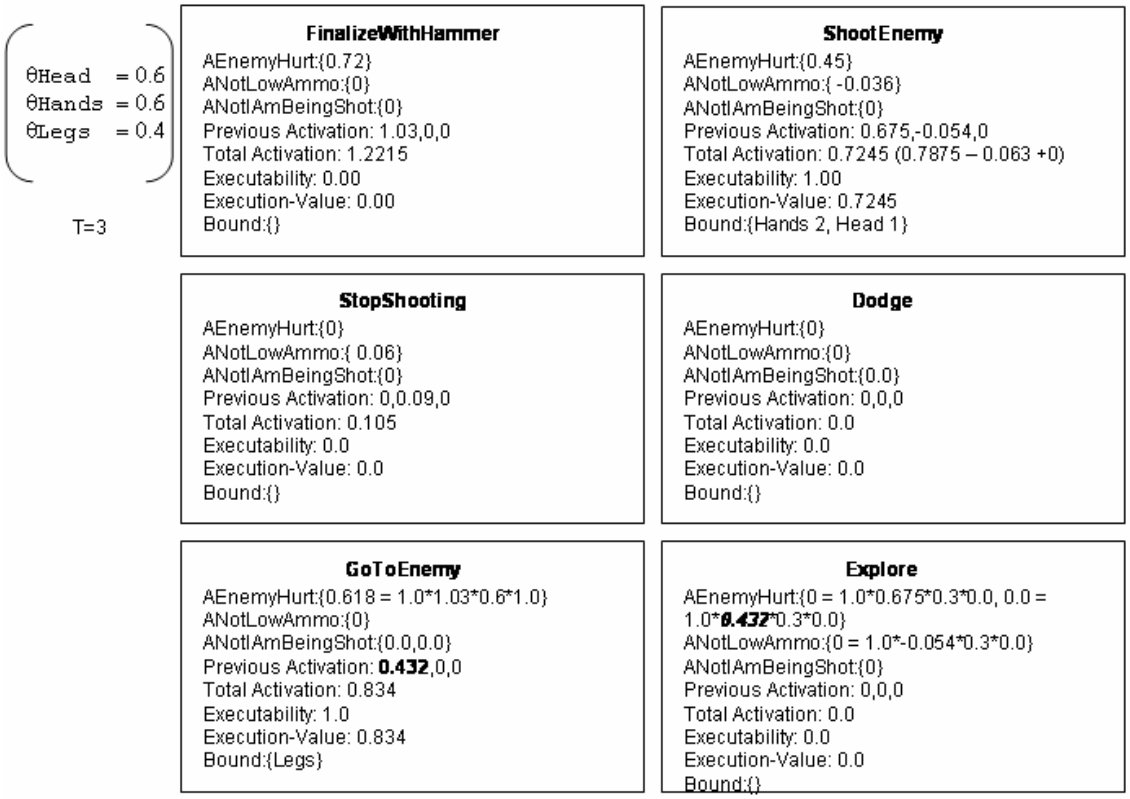

Figure 2.9: Third activation spreading step.

At this step behavior *GoToEnemy* surpassed the threshold of the *Legs* resource, so it binds them. *ShootEnemy* was able to bind and execute again, and once more it resets the thresholds of the resources it used. Now both *GoToEnemy* and *ShootEnemy* will be executing concurrently as they are not dependent on the same resources.

Explore is not receiving activation from the modules it is linked to as *EnemyInSight* has verity 1.0. Remember that we spread activation through a predecessor link as much as the source condition is not satisfied. As *EnemyInSight* is totally satisfied no energy is spread through the predecessor links.

Thus we came to the end of our exposition of extended behavior networks. In the third chapter we show how the network's propositions get their values, how the behaviors of the network are built, and how sensors, behaviors and the network are combined into an agent for Unreal Tournament. The next section presents a detailed overview of Unreal Tournament.

2.2 Unreal Tournament and Gamebots

Unreal Tournament is a top-selling 3D real-time action game. In the game mode we used, Death Match, the agent is an armed warrior who must kill all other warriors in an arena. The agent has many weapons available, each with certain properties (beat, pierce or explode) and several items to use. It moves over different landscapes and interacts with several other agents. The action repertory is large (run, walk, turn, crawl, shoot, change weapons, jump, strafe, pickup item and use item among others) and an agent may carry out more than one action simultaneously, such as dodging while shooting. The scenarios are 3 dimensional continuous spaces and the action happens in real-time, so the agent has to decide quickly what to do at each time step.

As the game properties determine in a large extent what kinds of agents we may design and how we design and implement them, an exposition of some game details is due.

Under normal game rules, a player starts with a health of 100 and dies when its health reaches zero or it falls into a pit. With special items (health vials) it may reach a health value of 200. A player is able to replenish its health by picking up medical kits ("medboxes" and "medkits") and health vials.

When a player dies it automatically is "resurrected" in another spot of the game level. Its score is diminished by one and the killer's score is augmented by the same amount. If the player kills himself, for instance, by walking into a pit, his score is decreased by one. The winner in a DeathMatch game is the player who first achieves a certain number of killings (a killing is called a *frag* in the game).

A player may pickup several items of armor (thigh pads, breastplate) that add up to its armor score (default max is 200). The higher this score, the more damage is absorbed from shots and explosions. Though the armors have names which make one associate them with body parts, the game does not take this into account – a thigh pad incredibly protects one from a head shot! As the player receives shots its armor score is lowered and its armor becomes less effective in absorbing damage.

Figure 2.10: An Unreal Tournament GOTY Screenshot (UNREALTOURNAMENT, 2005). At the upper right corner we see that the player has 150 units of armor and 100 units of health. In the bottom of the screen we see that its ammo for weapon number 0 (Sniper Rifle) is 46 units.

The player may walk or run. In addition to walking forward or backward a player may walk sideways (like a crab). Also, he may combine these moving modes with crawling – i.e. crawling sideways. The player is able to jump, either on spot or while moving.

The weapons inflict basically 4 types of damage: concussion, piercing, explosion and splash. The later two affect a player if it is within a certain radius of the place of impact of the shot. There is specific ammunition for each weapon type, and the maximum ammunition for each weapon is different.

The agent is able to look and shoot in all directions.

The game uses a client-server architecture. The players and bots send messages requesting sensory information or the execution of an action to a central server. The server processes the messages, synchronizes, handles events and informs the players the current game state.

We used a package that provides a clear set of messages for interacting with the game using sockets, simplifying the low-level aspects of agent design and programming. This package is called Gamebots (KAMINKA et al, 2002).

An agent receives both synchronous and asynchronous messages through Gamebots. Each synchronous message received is in fact a batch of messages of same timestamp, representing general information about the player and the game, such as game score, player health, player position, player rotation, whether the player is walking or running, etc.

Asynchronous messages report non-periodic events in the game, such as the picking up of a certain item, being shot or falling into a hole.

The complete set of messages of the package is included in Appendix A. It has minor differences from the documentation provided in (PLANETUNREAL, 2005) as we discovered minor errors in the description of the message formats and corrected them in the appendix.

Figure 2.11: Another Unreal Tournament GOTY Screenshot (UNREALTOURNAMENT, 2005). At the upper right corner we see that the player has 150 units of armor and 100 units of health. In the bottom of the screen we see that its ammo for weapon number 2 (Double Enforcer) is 199 units.

3 AGENT ARCHITECTURE AND DESIGN

Extended Behavior Networks were designed as a control mechanism for situated agents. From an engineering perspective this means that the abstractions used by the agent's "cognitive" system, such as propositions and representations of entities, are made relative to the tasks the agent performs, the environment he is embedded in and its goals.

The environment of the agent is defined not only by the game features and game rules but also by the primitive sensory-motor messages available to the agent for interacting with the game. Thus, when designing sensors, goals and behavior modules we take into account the Gamebots package messages and commands. The game mode used and the levels into which the agent interacts also provide additional constraints and needs upon the agent's design.

When we design an agent for a game we need to tackle at least three issues: How it will perceive its environment, how it will decide what to do at each time step and what basic behaviors will be available to it.

Each of these factors restricts and influences each other. An agent is able to make decisions based on its knowledge and perception of the environment. Conversely, the perception of the agent is influenced by its goals and beliefs, its interests in the moment and the actions it is carrying out, i.e., they are deeply tied to its cognitive apparatus and activity.

The behaviors of the agent must be appropriate to the decision mechanism employed so as to be properly selected and coordinated. We say that a behavior is appropriate to an action selection mechanism (ASM) when it can be easily modeled according to the expected ASM conventions and its granularity is properly dealt with at the level of the ASM.

How behaviors are executed is affected by perception and action selection. For instance, "attack weakest enemy" will be influenced by our perception of who among the enemies is the weakest. The "decidedness" to carry out an action is an example of influence from the action selection mechanism - strong volition to open a door usually leads to greater speed and force in its opening[10].

---

[10] Dorer (2004) describes variations of the behavior network model used in this work that parameterize the execution-values and use these values to influence the amount of

The real-valued propositions of the extended behavior network model may be realized by means of fuzzy sensors. Fuzzy sensors have been successfully applied to various domains in which we have continuous properties and arbitrary categories. For a game they are specially suited to establish the verities of propositions such as “enemy is near” and “enemy is strong”.

Behavior network modules are also called “competences”[11]. This gives us a hint of the granularity we should use when designing modules for a behavior network. We want to make behaviors such as “Go To Enemy”, “Shoot Enemy” and “Dodge Shot”, which resemble us of an ability or competence of the agent. Consider behavior “Shoot Enemy”. The subtasks involved, such as “get current position of enemy K”, “estimate position of K at shoot time” and “shoot at position X, Y, Z” should not be visible to the behavior network. Instead of considering each of these small steps in the decision making process, we make each behavior a finite-state machine in which the states correspond to the behavior’s low-level subtasks. Each behavior module is also responsible to monitor its own states, restarting and finishing when necessary.

We must note that our methodology for designing behaviors is not rigid and that what should be a complete module or just a state in the finite-state machine of a more complex module has a blurred boundary.

3.1 Behavior Network

In this section we present a behavior network that was the basis of the agents built in our experiments. We present the network as we have built it, step by step, showing the rationale behind each choice.

As mentioned earlier, our agent was designed to play in DeathMatch mode. This means that its basic concerns are killing as many enemies as possible and avoiding death. It must be able to detect, chase and shoot enemies, to avoid being shot, and to replenish its health.

We start by defining the agent’s goals. From the game rules, we see that it has to kill enemies and stay alive. To kill an enemy it has to hurt him until he dies and to stay alive the agent has to maintain its health and restore it when it is depleted. To fulfill the need to kill enemies we give the agent the goal of hurting enemies (*EnemyHurt*). For its self - preservation we add the goals of avoiding being shot (*Not IAmBeingShot*) and having a high health (*HaveHighHealth*).

Now we must design the behaviors that will achieve these goals. To hurt an enemy our agent must attack it. A sensible candidate would be behavior “AttackEnemy” with precondition *EnemyInSight* and effect *EnemyHurt*. The problem with such a general behavior is that if we attack with a shooting weapon, we have to specify the additional effect of ammo decrease. If we attack with a contact weapon (Impact Hammer or Chainsaw) we need to be near to the enemy and behavior execution will have no effect upon the ammunition. Thus, we create two attacking behaviors instead of just one: *FinalizeWithHammer* and *ShootEnemy*. Figure 3.1 shows these behaviors linked to goal

---

resources spent in behavior execution. In chapter 5 we discuss our reason for not using this variation.

[11] See (MAES,1989) for an example.

EnemyHurt. Note that they use exactly the same resources, which makes them mutually exclusive (unless we use this network with a four armed, two headed agent).

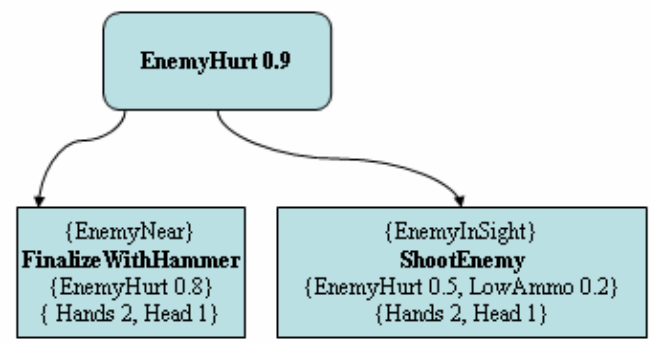

Figure 3.1: Goal EnemyHurt and its satisfying modules.

Now we need modules that make true *EnemyNear* and *EnemyInSight*. As the only sensing ability of the agent is vision, for it to tell if an enemy is near it must first have perceived an enemy in sight. We have three ways to make *EnemyInSight* true: stand still and wait for an enemy to appear in front of us, stay were we are and look for an enemy around us or actively explore the level to find an enemy. These three possibilities are reflected in behaviors *Stand*, *StandLookout* and *Explore*, respectively. To stand in place there are no preconditions, but we wish to explore or look out for an enemy only if there are no enemies in sight. If we are shot we must stop exploring and find what is shooting us, so we add the precondition *Not IAmBeingShot* to behavior *Explore*. A module that has *EnemyNear* as an effect is what is left to complete the paths to the goal. A trivial solution is module *GoToEnemy* that has as precondition *EnemyInSight* and as effects *EnemyNear* and *IAmBeingShot*. We add *IAmBeingShot* as an effect because going towards an enemy makes it likely to be shot. Figure 3.2 shows the network built so far, with only goal *EnemyHurt* included.

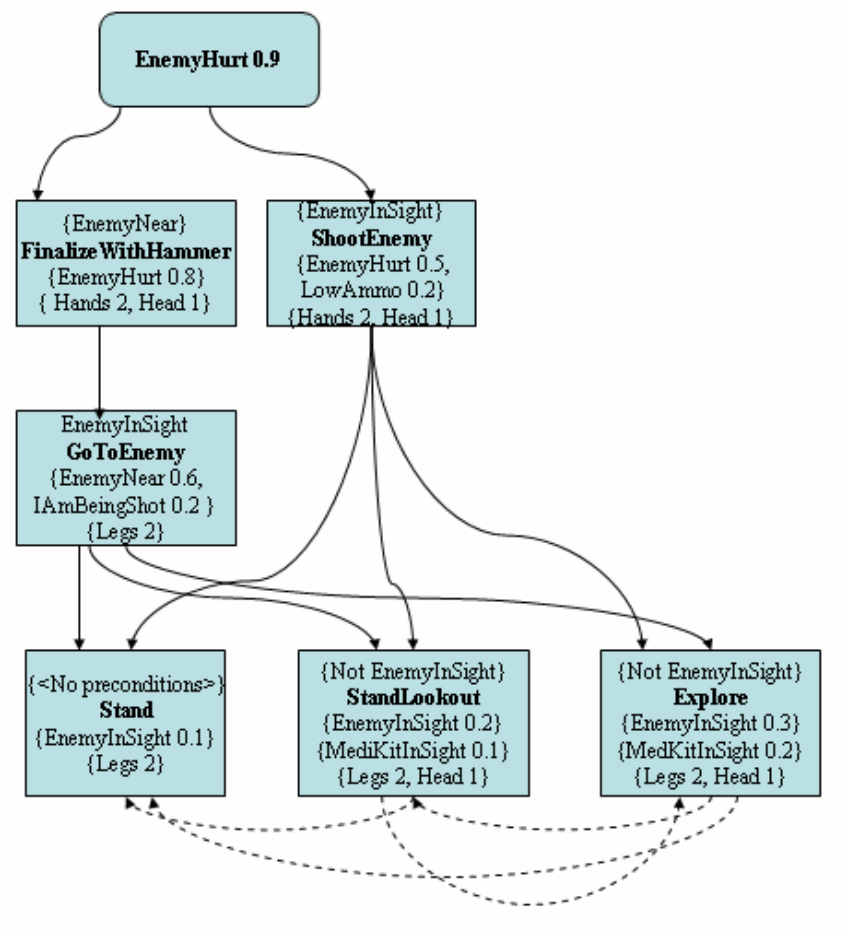

Figure 3.2: Behavior Network for goal EnemyHurt.

Now let us make modules to satisfy goal *Not IAmBeingShot*. The most basic way to avoid being shot, besides not fighting at all, is dodging the bullets. We create behavior *Dodge* with precondition *IAmBeingShot* and effects *Not IAmBeingShot*. The agent needs to use only its legs to dodge. The network now is as shown in figure 3.3.

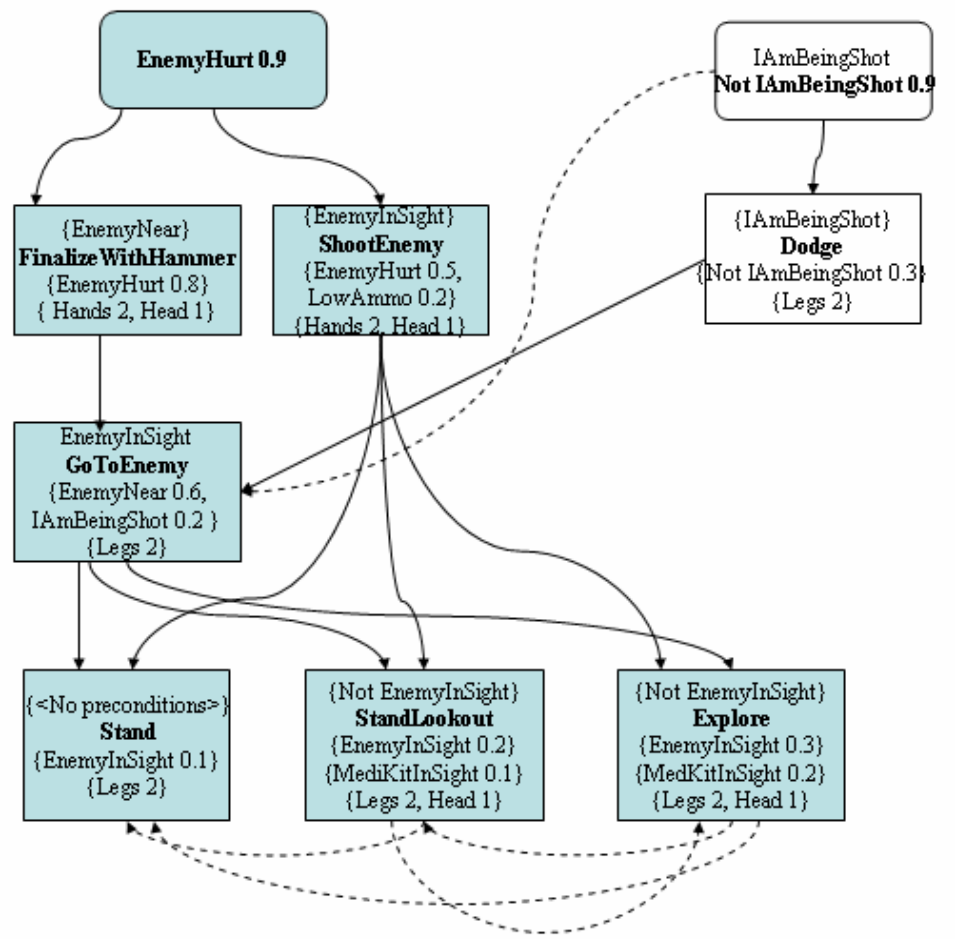

Figure 3.3: Behavior Network for goal *EnemyHurt* and *Not IAmBeingShot*.

Now lets see how to achieve goal *HaveHighHealth*. Health vials and medical kits, in Unreal Tournament, have fixed locations. With the game options we used in our tests, these kits, if taken, disappear for some time and then "magically" re-appear at the same location. Thus, health items have fixed locations within a level, enabling us to store their locations and just go to a known medical item whenever we need one. A module "Get Known Medical Kit" with precondition "Know Medical Kit Location" and effect *HaveHighHealth* would be our first solution. The problem with this module is that it is specified at a too high level of abstraction: getting a known medical kit a hundred steps afar is treated in the same module as getting a health vial in the immediate vicinity. Also, this module will have to contain checks to see if there is a way to get a medical kit (i.e. if it is reachable) as well as deciding which of the known kits to get. May this module be made simpler by exploiting environmental or sensory information?

Examining the Gamebots messages and protocol we see that the robot receives as primitive information about an item in sight its position, its location and whether it is reachable, i.e., if there is a straight path to the item. If we exploit this information we can then make three modules instead: *GetReachableMedKit*, *GoToMedKitInSight* and *GoToKnownMedKit*. The first has as precondition there being a reachable medical kit and as an effect *HaveHighHealth*. The second may make a medical kit in sight a reachable medical kit. The last can make *MedKitInSight* true by going to a known medical kit location. We see that the precondition of *GoToKnownMedKit* may be

satisfied by either *StandLookout* or *Explore*, as both tend to lead to a known medical kit location. Figure 3.4 shows these modules incorporated in the network.

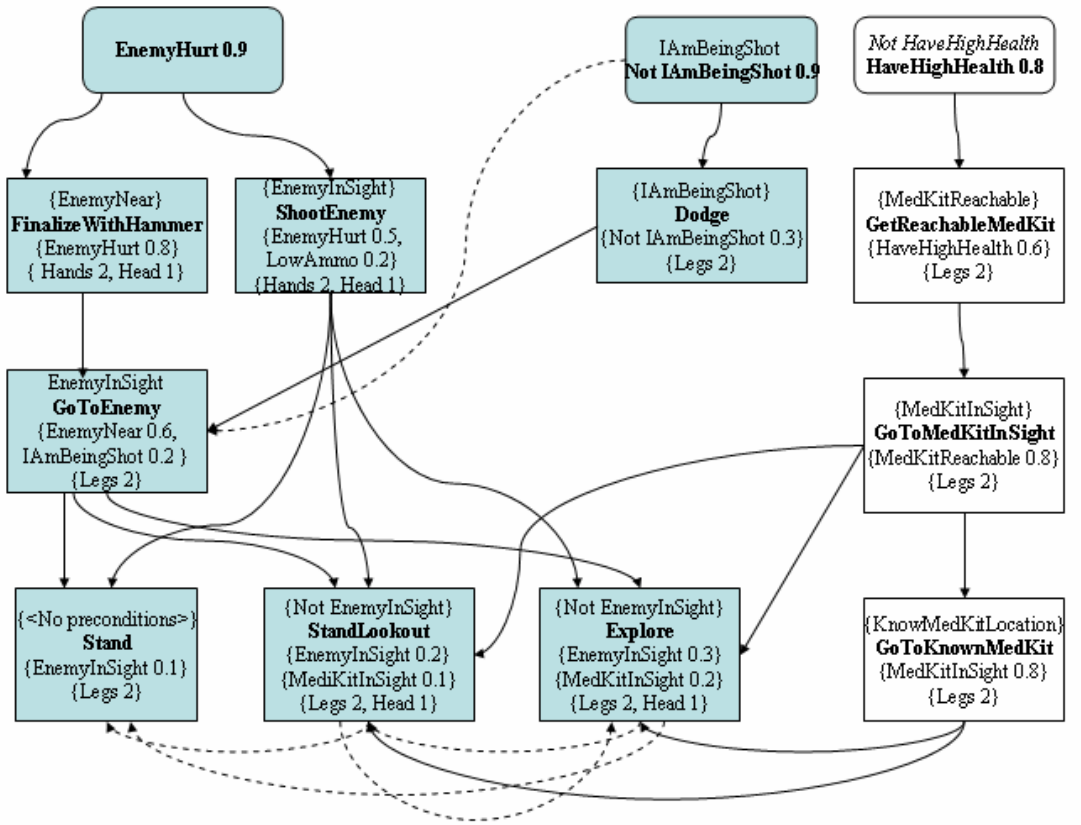

Figure 3.4: Behavior Network with health-related modules (white background) added.

Our network seems ready, but we haven't mentioned a particularity of Unreal Tournament that makes additional modules needed. Whenever we tell the game to shoot a certain agent or position (command SHOOT) it will keep shooting until we tell it to stop (command STOPSHOOT). We could well treat this inside a module, but as the need to stop shooting in fact is embedded in the aim of not wasting ammo, we create a goal for it, *Not LowAmmo*. Stopping to shot may be an independent, complete competence module, *StopShoot*, with preconditions *Not EnemyInSight* and effect *Not LowAmmo*. Figure 3.5 shows our complete network. We added the context condition *LowAmmo* to goal *Not LowAmmo* as we want the agent to "worry" about being with low ammunition the more it is actually with low ammo.

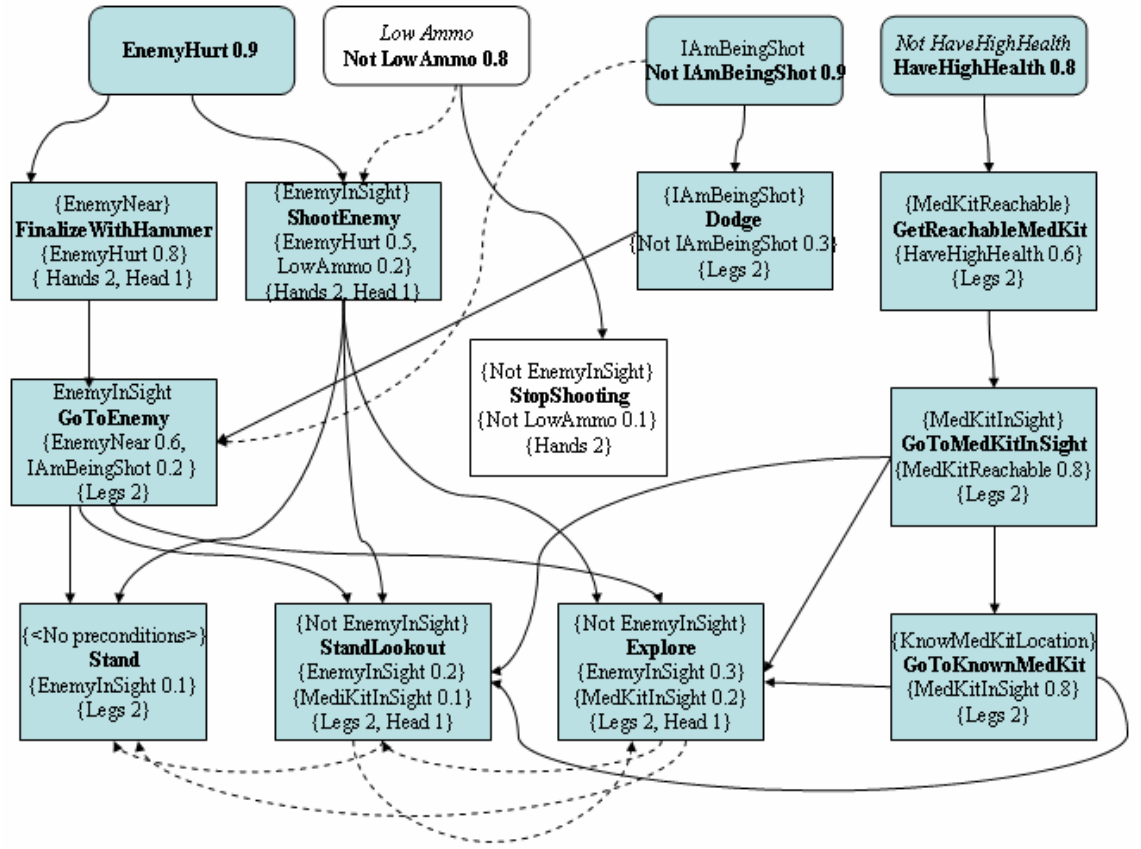

Figure 3.5: Complete Behavior Network

Having our network ready we are left with three questions: How will the network propositions be verified? How will each of the behavior actions be carried out? How will it all function together? In the next subsection we address the first question and provide a through overview of the agent's sensing mechanism.

3.2 Sensors

Before venturing into the exposition and discussion of our network's sensors lets us clarify of what kind of sensors we are talking about. In the beginning of this chapter we have pointed that agent perception is influenced by its internal state, its beliefs and its current activity. We also remember that primitive sensory information is defined by the messages of the Gamebots package. This information includes robot position, robot velocity, robot rotation, robot ammo, robot health, enemies in sight and items seen. The behavior network operates upon propositions, whose value attribution comprises a form of high-level perception. It is this high-level perception that is done by the sensors discussed in this section. Our sensors act upon the Gamebots messages and the knowledge of the agent to create the necessary information for action selection and behavior execution.

Each condition of the behavior network has a sensor associated to it. Each sensor has a membership function *P* and an internal state function *I*. Function *P* takes the current perceived state of the agent and returns a fuzzy proposition, corresponding to the condition, with value between [0..1]. Function *I* updates the internal state of the agent. Note that this functioning implies that the order in which the sensors are activated

matters – the internal state of the agent may be altered by each sensing. Figures 3.6 to 3.9 illustrate in a high-level how sensor *SensorEnemyInSight* operates.

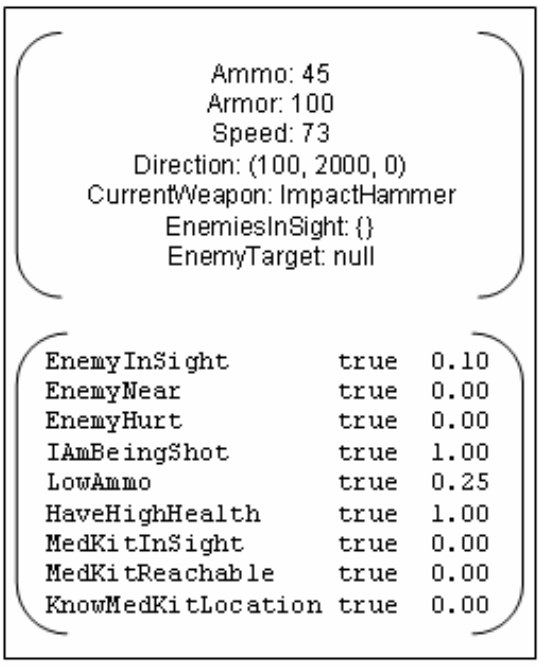

Figure 3.6: Agent internal state and propositions just before receiving Gamebots messages.

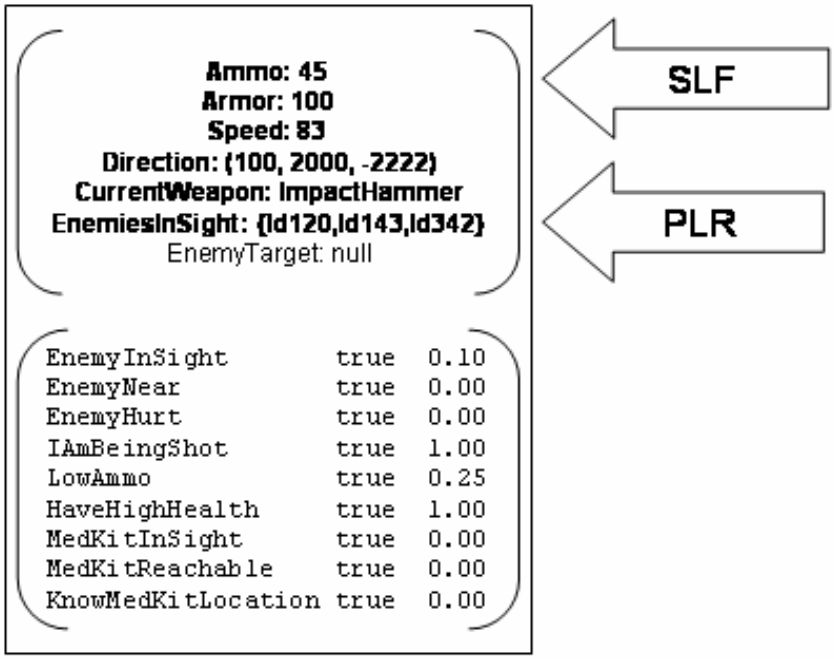

Figure 3.7: Internal state and propositions after receiving Gamebots messages SFL e PLR. Fields in bold signal what was changed by these messages.

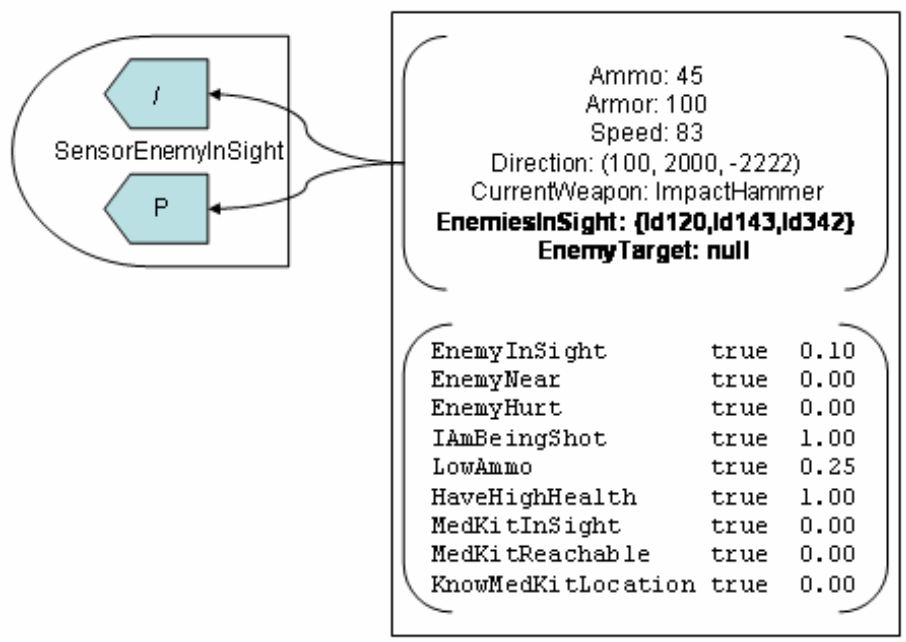

Figure 3.8: Sensor *SensorEnemyInSight* reads in the internal state of the agent. Both functions *l* and *P* receive the same input. The data in bold shows what is actually used by this sensor.

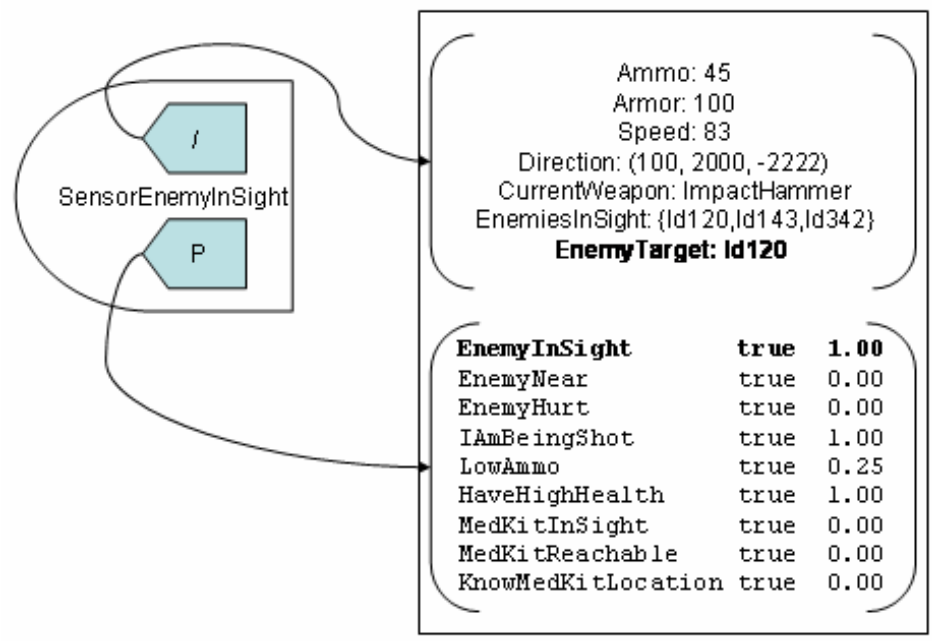

Figure 3.9: Sensor *SensorEnemyInSight* updates the internal state and the behavior network propositions. Function *l* sets the enemy target to Id20 and *P* sets the verity of *EnemyInSight*. In bold is what the sensor has changed.

The cautious reader may now be asking himself why we keep only true propositions. Two reasons: simplicity and storage saving. The operator we use for negating a proposition p is $N(p) = 1 - p$. (8), so we can always make a straightforward conversion between the value of a proposition and its negation.

To finish our exposition of sensors, let us proceed to examining how sensor *SensorEnemyNear* operates, as it provides a good example of the importance of sensor

ordering. As Figure 3.10 and Figure 3.11 show, *SensorEnemyNear* gets the target set by *SensorEnemyInSight*, the robot's own position and the target position and sets the verity of proposition *EnemyNear*. This sensor alters the propositions but leaves the internal state untouched. Note that it is crucial that *SensorEnemyInSight* operates first, because the assessment of nearness is done relative to the target set by this sensor. Figure 3.12 shows how *P* establishes the truth-value of *EnemyNear*.

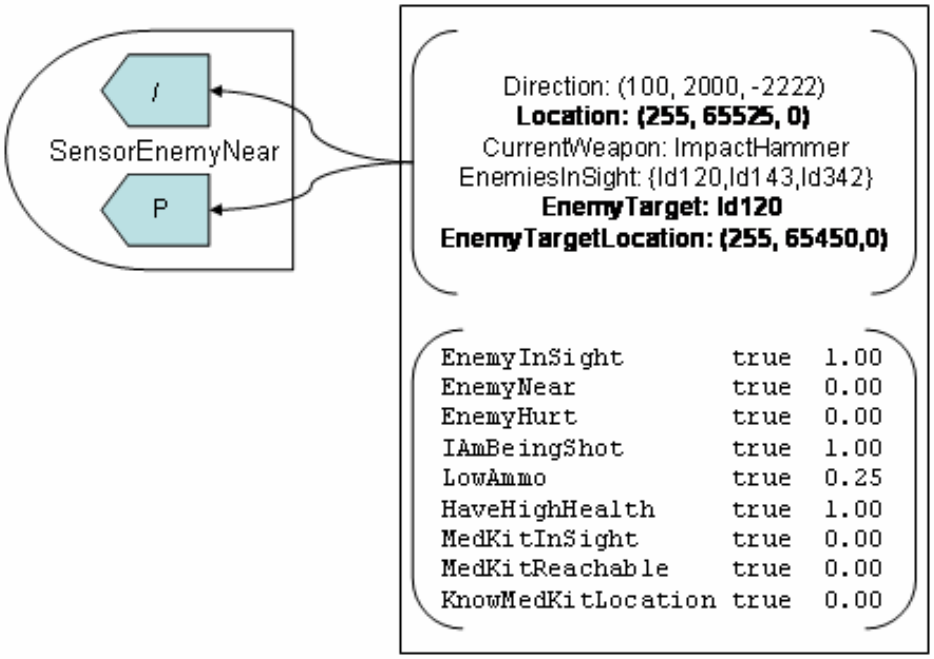

Figure 3.10: Sensor *SensorEnemyNear* reading internal state data.

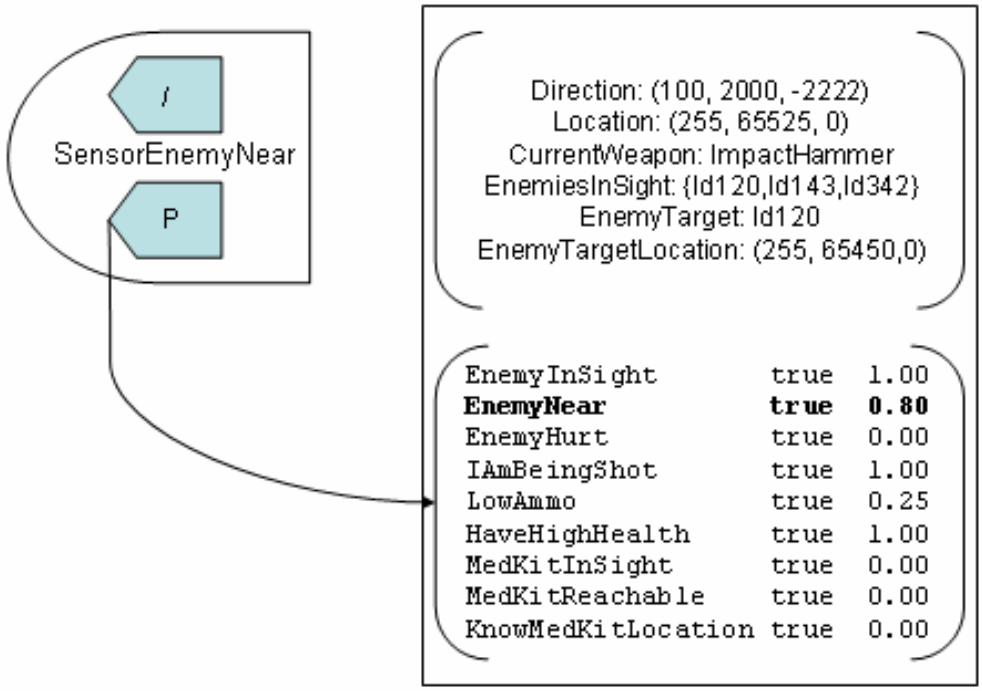

Figure 3.11: Sensor *SensorEnemyNear* updating proposition *EnemyNear* value.

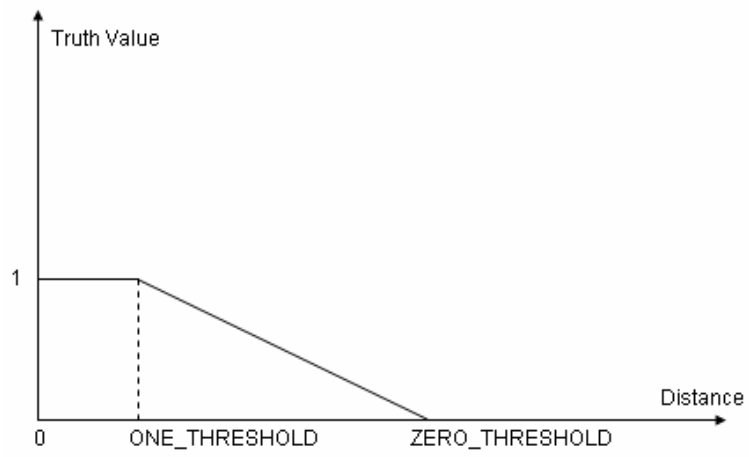

Figure 3.12: Truth-value formula of *SensorEnemyNear*. ONE_THRESHOLD is the distance less than which we consider the enemy surely near. ZERO_TRESHOLD is the distance such that for distances grater than it the enemy is surely not near.

Having seen the subtleties of agent sensing and how the network's propositions get their values, we are ready to examine the behavior modules.

3.3 Behaviors Modules

From chapter 2, we recapitulate that a behavior module is made up of a list of conditions, a list of effects and an action. We have seen how the conditions are set in the previous section, so now we are ready to see how behavior actions are constructed.

The actions of the behavior modules are built around augmented finite-state machines. When selected for execution each behavior action has a certain amount of time to execute. The module is responsible for monitoring its own action execution and start, resume and interrupt each state as necessary.

A behavior action may issue one or more primitive commands to the game when executing. All behaviors have access to the internal state of the agent.

Figure 3.13 shows the sequence diagram of behavior GoToEnemy.

We see that the behavior is fairly complex, even though the high-level action is simple. The agent has to discover a path to the enemy, deal with obstacles that may appear in its way and give up whenever it concludes that the target enemy is unreachable.

Let us follow some paths in the diagram to illustrate how the behavior relates to the perception and internal state of the agent and how the behavior action relates to the primitive actions of the Gamebots package.

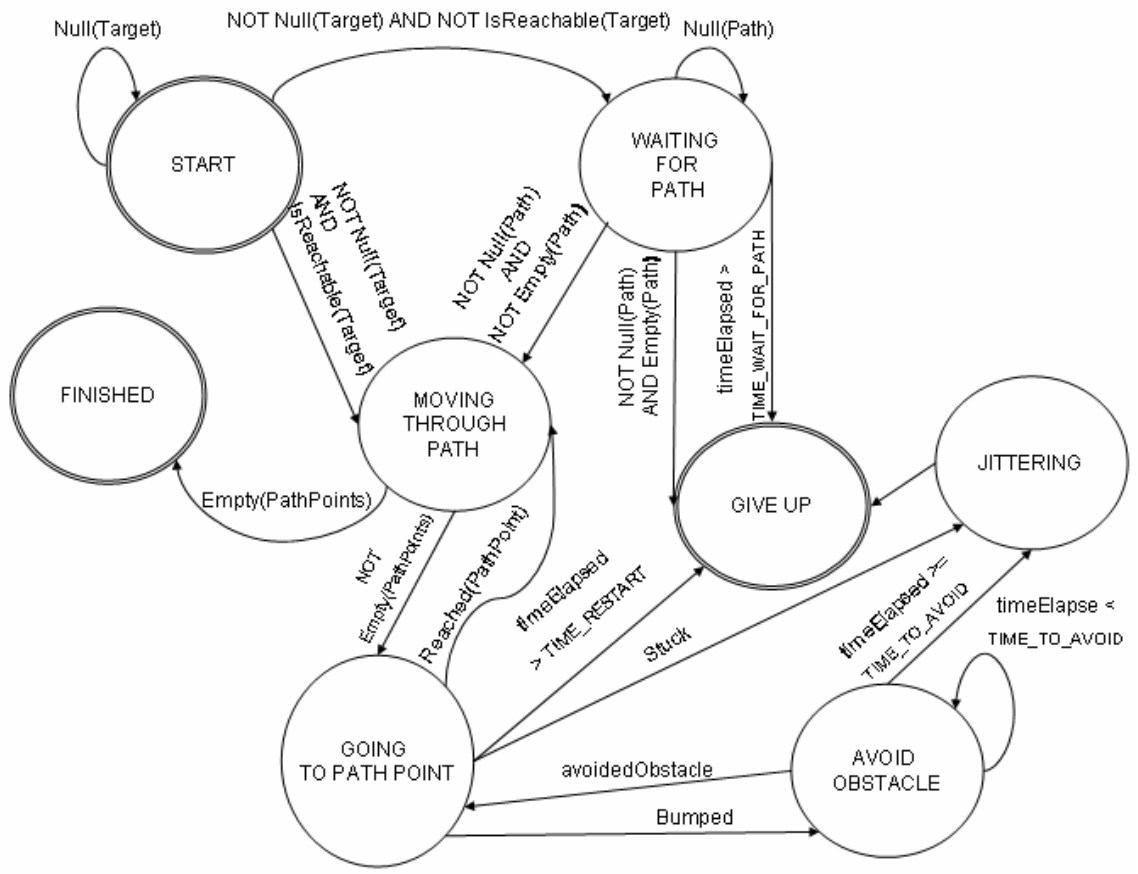

Figure 3.13: Sequence diagram of behavior GoToEnemy. Null(X), IsReachable(X), Reached(X) and Empty(x) are predicates. Stuck and Bumped are boolean flags. Target, TimeElapsed, PathPoints and PathPoint are variables. TIME_TO_AVOID, TIME_WAIT_FOR_PATH and TIME_RESTART are constants used for the behavior self-monitoring. States START, FINISHED and GIVE UP are terminal states. FINISHED is successful reaching of the target, GIVE UP is a desistence.

Let us start at state START. The variable target is the one set by sensor *SensorEnemyInSight,* as described in the previous section. If there is no target set (Null(target)), the agent does nothing. If the target is not immediately reachable the module requests a path to it and enters WAIT FOR PATH. Upon receiving the path to the target the agent starts moving (MOVING THROUGH PATH).

In state WAIT FOR PATH the module sends a GetPath command to the game and waits for the corresponding path message (PTH). If the time of wait (timeElapsed) goes beyond the maximum allowed (TIME_WAIT_FOR_PATH) the agent gives up going to that target (state GIVE UP).

But what are these paths all about? Unreal Tournament levels are covered by graphs that span most of the level area. Each node of the graph is a possible destination and is called a navigation point (NavPoint). A navigation point may be the place of an item, a weapon or just a point where the agent may move. We are able to give specific 3D coordinates for the agent to move, but more often we provide just a NavPoint id. The game provides the facility to get a path to any point, using the GetPath command. The response to this command provides a list of NavPoints to the target if there is a path to it or *Null* is there is no path.

Let us continue following the diagram from MOVING THROUGH PATH. When the agent has no more points in its path it goes to state FINISH and ends the behavior successfully. If there is a point to go it enters state GOING TO PATH POINT. If it encounters an obstacle while going to the desired NavPoint, it goes to state AVOID OBSTACLE. If after a certain time the agent has not managed to avoid the obstacle it goes to state JITTERING. In this state the agent tries some random action just to get away from where it is. It is a primitive heuristic to deal with occasions when the agent is stuck. A better implementation would make an internal map of the level and perform some reasoning on the possible stuck cases (crates that leave a space too narrow for the agent to go through, holes, jumpable obstacles, etc). After jittering the agent always gives up (state GIVE UP). We decided to take this course of action because we consider that whenever the agent has failed to avoid an obstacle there has probably been too much time since we started moving towards the target, and it is better to restart from state START.

The cautious reader by now may be wondering: How does the behavior actually executes? Does it go from state from state until reaching a terminal node? Can it be interrupted in the middle of a state?

Remember from the beginning of this section that we said that each behavior kept track of its own state and was responsible for making the state transitions, interruptions, etc. All modules have a single method available to the agent, called *perform*. This method receives as parameter the internal state of the agent. So, we see that the agent never interrupts a behavior, only the module itself may do it.

The case of interruption common to all behaviors is when it is in a non-terminal state and too much time has elapse since it was last executed. In this case it is re-set and starts from state START. Each call of the execute method make at most one state transition. This was made to allow a great deal of flexibility on how behaviors would be executed, scheduled, etc.

Now that we have seem how the network, the sensors and behaviors were constructed it is time to put it all together. The next section describes how our agent is integrated into the game.

3.4 Integration

In this section, in addition to showing how the agent is integrated into the game, we provide some final remarks on how the behavior network, the behavior modules and the sensors are combined. Figure 3.14 illustrates the communication between the agent, Gamebots and Unreal Tournament server.

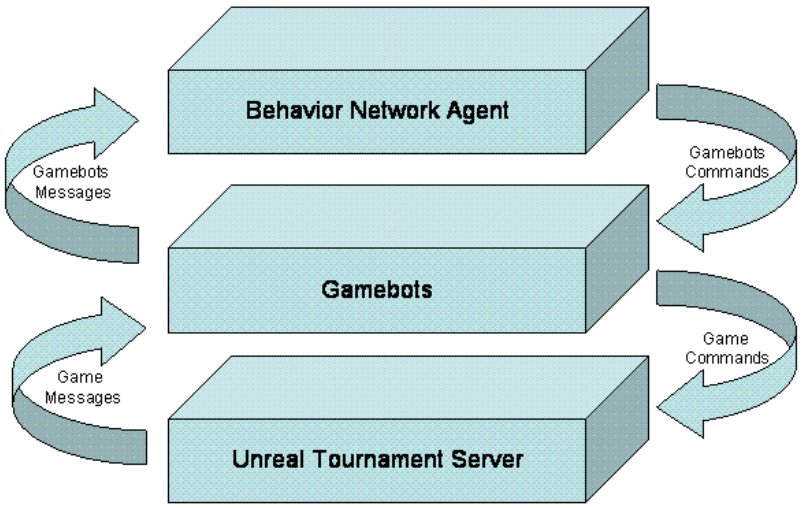

Figure 3.14: Integration of the Behavior Network Agent with the Gamebots package and Unreal Tournament server.

Unreal Tournament server sends game messages to Gamebots. In turn Gamebots sends its own messages to our agent. The agent's sensors process these messages and update the values of the conditions of the behavior network and the internal state of the agent. Action selection takes place and the appropriate actions are carried out by sending commands to Gamebots that in turn send low-level Unreal Tournament commands to the server.

Well, but when specifically action selection happens? From chapter two we remember that messages in Gamebots can be synchronous or asynchronous. Synchronous messages come in batches at short intervals, usually each 50 milliseconds (we may configure the game to make synchronous updates less or more often). Whenever we receive an asynchronous message we treat it as if it had arrived at the time of the most recent synchronous batch, i.e., we update the network propositions only at the ending of a synchronous batch processing. This has no significant impact on performance as even the biggest possible delay in the agent's response due to this modification is not noticeable to a human.

Action selection takes place right after sensor updating. The selected behaviors are activated via calls to their *execute* methods. We see that the agent performs a full action selection cycle many times each second, leading to a fast response to changes and events in the environment.

4 EXPERIMENTS

In this section we describe the experiments to assess action selection quality, to measure agent performance and to bring forth character personality in the domain of Unreal Tournament.

In the first section, by means of analysis of the actions selected at each turn and of direct observation of the agent behavior in the game, we verify if the extended behavior network exhibit the properties of a good enough action selection mechanism: chaining of actions, good balance between reactivity and persistence, proper resolution of conflicts and preference for actions to contribute to more than one goal. Also, we verify one additional property essential to our domain: proper combination of concurrent actions.

The second section presents two experiments designed to asses the performance of an agent using extended behavior networks in this domain. In the first experiment we used a totally different agent, built by another group, to play against our extended behavior network agent. In the second experiment we used a robot identical to ours except for the action selection mechanism employed. The first experiment prevented bias on our part on the opponent's design. The second enabled us to verify if the action selection mechanism employed did make a great difference in the overall performance of our agent.

In the third section we show how to design stereotypes using extended behavior networks and how to tune the network so as to achieve different personality traits. This section also presents an investigation of the effects of different parameter settings in the overall behavior of the agent.

4.1 Action Selection Quality

Our first series of experiments were designed to asses the quality of action selection, using the behavior network of figure 4.1. We give a high-level description of the perceived state of the agent, the actions it executed and the values of the control parameters of the network during the experiment. The default configuration for the network was: γ (ActivationInfluence) = 1.0, δ (InhibitionInfluence) = 0.9, β (Inertia) = 0.5, θ (Global threshold) = 0.6. These parameters worked well in most cases. In a few experiments we tried extreme values for some parameters, specially the inertia β and the global threshold θ.

Each of the discussions below was based in log analysis and direct observation of agent behavior (Unreal Tournament allows one to log as "spectator" and just watch the game without interfering).

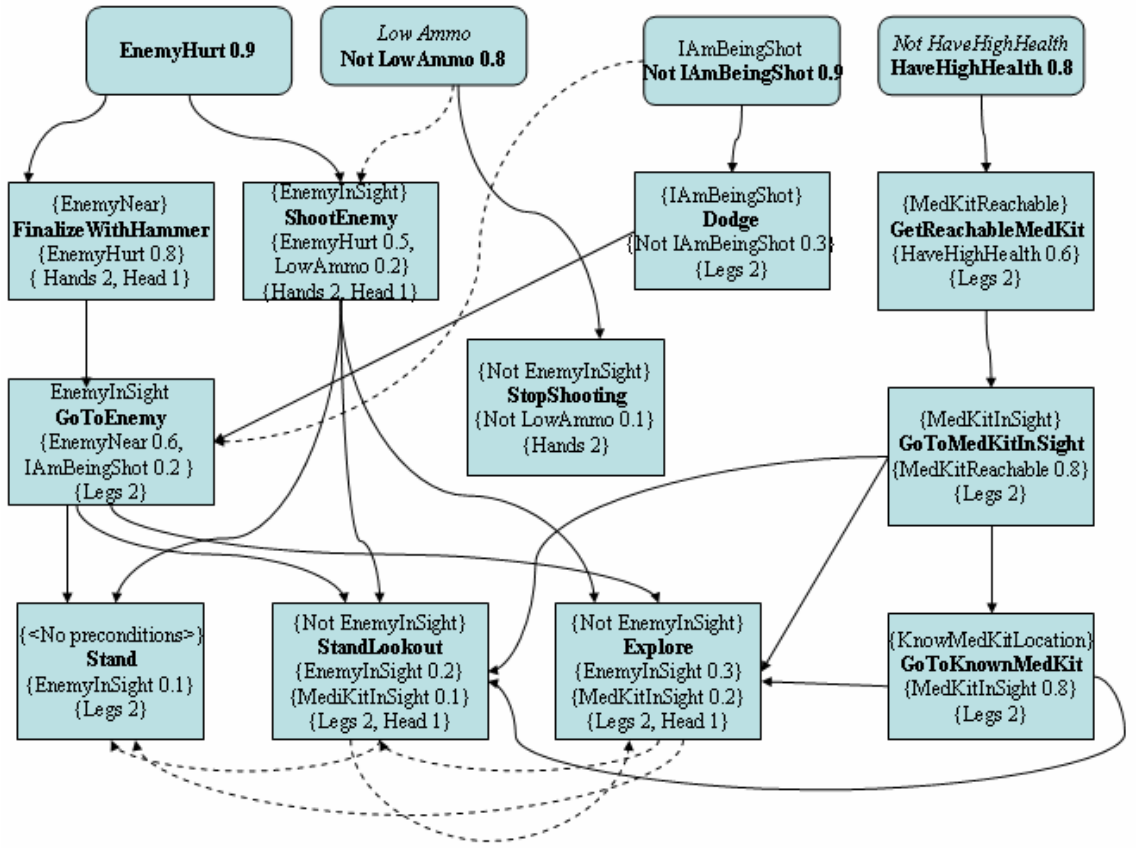

Figure 4.1: Behavior network used in the investigation of action selection quality.

4.1.1 Overall Behavior

The agent exhibited an intelligent behavior. It started exploring the level and kept wandering until it found an enemy (*Explore*). Upon finding and enemy it started shooting (*ShootEnemy*). If not being shot back it usually approached the enemy (*GoToEnemy*). Upon reaching the enemy it switched weapons and used the more powerful weapon Impact Hammer (*FinalizeWithHammer*). After the enemy died, it stopped shooting (*StopShoot*) and started wandering again. When shot repeatedly it kept shooting and after a while stopped going to the enemy and started dodging subsequent shots. If the enemy stopped shooting it would go towards it again. When the agent was hurt in combat, if it knew the location of a medical kit (*GoToKnownMedkit* had a high truth value), it would go to it and restore its health after a while.

4.1.2 Chaining of Actions

Three common action sequences were observed. The sequence {*StopShooting* and *Explore, GoToEnemy* and *ShootEnemy*, *FinalizeWithHammer*} was the usual attack sequence of the agent and {*GoToKnownMedkit, GoToMedkitInSight*, *GetReachableMedkit*} the one that was carried out when it had no enemies in sight and was not with high health. The long sequence {*Explore, GoToEnemy* and *ShootEnemy*,

FinalizeWithHammer, StopShooting and *GoToKnownMedKit, GoToMedkitInSight, GetReachableMedKit*}, that is basically the two previous ones one after the other, was the most common overall behavior of the agent. We see that the agent makes reasonably long consistent chains of actions even though the agent makes no formal planning.

4.1.3 Reactivity and Persistence

We can see the sequence of actions {*Explore, GoToEnemy, FinalizeWithHammer*} as a plan to fulfill the goal *EnemyHurt.* If while going to the enemy the agent was shot, it stopped behavior *GoToEnemy,* executed *BehaviorDodge* and, having evaded the shot, resumed *GoToEnemy*. We see that the agent reacted to an event in the environment and then got back to our perceived "plan". It exhibited a good compromise between reactivity and persistence. For large values of β, the inertia, the agent took some time to dodge after perceiving a shot and only dodged when a sequence of shots happened.

4.1.4 Resolution of conflicts

Let us take a look at figure 4.1 again. We see that goal *EnemyHurt* tries to make behavior *ShootEnemy* execute and goal *Not LowAmmo* tries to prevent *ShootEnemy* from executing. The goal *Not LowAmmo* has little influence until the agent starts to be with very low ammunition. When this happens, the conflicting influence of *Not LowAmmo* makes the agent switch to the hammer weapon (*FinalizeWithHammer*), because it does not need ammunition. It is an unusual though sensible approach to ammunition saving that emerged by the interaction of the goals (note that the "designed" way to save ammo is using behavior *StopShooting*).

Another case of conflict resolution happens at behavior *GoToEnemy*. To better fulfill *EnemyHurt* the agent has to get near, but to satisfy *Not IAmBeingShot* it better not. In 4.1.1 and 4.1.3 we saw that the network deals with this conflict well, dodging when appropriate and resuming the going to the enemy after avoiding the shot.

4.1.5 Preference for actions that contribute to several goals

The network always preferred *StandLookout* and *Explore* to *Stand* when they had equal executability, even when we used larger expected values for the *EnemyInSight* effect of *Stand*. The reason is simple: both *StandLookout* and *Explore* contribute to *EnemyHurt* and *HaveHighHealth*, while *Stand* just contributes to *EnemyHurt* (In fact it contributes to *ShootEnemy* that in turn contributes to *EnemyHurt*). The modules that contribute to more goals are able to accumulate more activation, being selected more often.

4.1.6 Proper combination of concurrent actions

We see that the agent makes good use of its resources and combines the actions properly. It shoots while dodging a bullet or running towards the enemy, it stops shooting while exploring or getting a medical kit and even continues to shoot while getting a medical kit. All combinations of actions are reasonable and the good action combinations we could conceive were observed, as the previous analysis attests.

4.2 Agent Performance

To asses the performance of an agent built around extended behavior networks we designed two experiments. In the first we tested our agent (EBN_Bot) against a totally different agent built by another group. This provided us with a baseline comparison and prevented the possibility of ourselves cheating on the robot opponent. The second experiment used a robot that had identical sensors and behaviors but a different action selection algorithm. This second experiment was made to detect with more precision the contribution of the extended behavior network to the agent's performance. If we got a better result than a robot that used a different sensory-motor apparatus, it could be due to the apparatus, not the action selection mechanism.

4.2.1 First Experiment: The Behavior Network Agent Compared to a Totally Different Agent Built Around Finite-State Machines.

In our first experiment we used Carnagie Mellon's CMU_JBot, a Java agent based on finite-state machines that comes with the Javabots package (JAVABOTS, 2005). Our robot used the network presented in the previous section.

We made a series of 10 games of 1 minute. For each game we recorded the number of times the agent hit the opponent, the number of times the agent was hit, the number of times the agent was killed and the number of times the agent killed he opponent. The total score was got by giving 0.1 to each time the agent hit the opponent and 1 to each time the agent killed the opponent. Table 1 summarizes our results. The rightmost column presents the difference between the score of the agent using the extended behavior network (EBN_Bot) and the score of CMU_JBot.

Table 4.1: Results of Death Match between CMU_JBOt and EBN_Bot

Experiment #	EBNBot Hit	EBNBot Kill	CMU JBot Hit	CMU JBot Kill	Difference
1	0.7	0	0.2	0	0.5
2	0.1	0	0.0	0	0.1
3	0.3	1	0	0	1.3
4	0.7	0	0.1	0	0.7
5	0.9	0	0	0	0.9
6	0.4	1	0.1	0	1.3
7	0.6	0	0	0	0.6
8	0.7	1	0.0	0	1.7
9	0.9	0	0.1	0	0.8
10	0.2	1	0.2	0	1.0
Mean	0.55	0.4	0.06	0	0.8

We see that our agent scored much higher than the agent that used finite state machines. The low number of killings, even when many hits happened, is due to the

absence of a chasing mechanism in both agents. The agents wandered through the environment, shot each other and then separated, several times.

This experiment adds evidence to the suitability of behavior networks as an action selection mechanism for Unreal Tournament agents, but a doubt remains: Is the better performance due to our sensors and behaviors or due to the action selection mechanism used? The next experiment sheds light on this issue.

4.2.2 Second Experiment: The Behavior Network Agent Compared to a Plain Reactive Agent that Uses the Same Sensory-Motor Apparatus

In the second experiment we compared the EBN agent to an agent that has exactly the same sensors and behaviors, but uses a different action selection strategy: At each time step we disregard activation spreading for action selection and take into account only the executability of each module. Using only the executability results in a totally reactive agent with fuzzy sensors in which the modules that have the most satisfied pre-conditions execute. Note that the concurrent action selection is properly carried in this agent too.

Now that we do not have activation we are faced frequently with situations in which two modules have the same execution-value. Let us consider for instance *FinalizeWithHammer* and *ShootEnemy*. When we have *EnemyNea*r = 1.0 we necessarily will have *EnemyInSight*=1.0 (remember from chapter 3 that the sensor for *EnemyNear* uses information created by the sensor for *EnemyInSight*). So, both will have identical execution-values, creating the need to hard code some priority rules or insert additional conditions to decide which one to launch when appropriate. We have opted for the first approach in most cases, to differ as little as possible from the original behavior network.

Behavior *Dodge* has priority over *GoToEnemy*, behavior *GoToReachableMedkit* has priority over both *GoToMedkitInSight* and *GoToKnownMedkit*, and behavior *GoToMedkitInSight* has priority over behavior *GoToKnownMedkit*. We incorporate one subtle but important rule: *FinalizeWithHammer* gets priority over *ShootEnemy*, because we want the robot to hammer if the enemy is near. We changed module *Explore* to have the condition *Not IAmBeingShot*, both in the behavior network agent and in the plain reactive agent. This was made to separate the case when it is better to wander (*Explore*) and the case when it is better to stay on lookout (*StandLookout*).

To overcome the low number of killings we implemented a chasing behavior (identical) in both agents. Table 2 summarizes our results for 10 games of 30 seconds each.

Table 4.2: Death Match of EBN_Bot and the Reactive Agent.

#	EBNBot Hit	EBNBot Kill	ReactiveBot Hit	ReactiveBot Kill	Difference (EBN-Reactive)
1	0.7	0	0.9	1	-1.2
2	0.1	1	0.1	0	1.0
3	0.3	1	0.2	0	1.1
4	0.2	0	0.3	1	-1.2
5	0.9	1	0.3	0	1.6
6	0.4	1	0.1	0	1.4
7	0.0	0	0.1	0	-0.1
8	0.7	1	0.9	0	0.8
9	0.6	0	0.6	1	-1.3
10	0.2	1	0.3	0	0.9
Mean	0.44	0.6	0.34	0.3	0.3

Our robot had significantly bigger overall scores, as in the previous experiment. One interesting point is that our robot had 100% more killings but just a little over 30% more hits. This is due to the quality of its action chains. It stopped to heal itself when very hurt and dodged bullets when taking many consecutive shots.

Another point that catches attention is that the mean difference in total score was much smaller in this experiment. It is evidence that the quality of sensors and behaviors was in great part responsible for the superior performance of our agent against CMU_JBot.

These two experiments together show that extended behavior networks are a good action selection mechanism for agents in action games and that indeed the high action selection quality is responsible for a great part of the agent performance.

4.3 Character Design

We have designed five stereotypical personalities suited to the combat scenario of Unreal Tournament: The Veteran, The Novice, The Coward, The Samurai and The Berserker. For each, we used one or more of the following approaches: changing the global parameters, changing the goal strengths and changing the network topology itself. Besides illustrating personality design, this section explores in greater detail the configurations of the parameters.

The Veteran is calm and rational, trying to maximize all its goals in the long run. He has great self-control and persistence and wants to kill as many enemies as possible, but never at the expense of his life.

The Novice aspires to be like the veteran, has similar values, but still lacks the endurance and discipline to act properly. He is impulsive and frequently does not take the best action for a circumstance

The Coward's main goal is getting out of the combat alive and unhurt. He will avoid direct confrontation and will attack only when no other good option exists, always prioritizing maintaining and restoring its bodily integrity.

The Samurai is cold, persistent and aggressive. To die in battle is his highest honor. Killing his opponent is his stronger goal and he will try to achieve it even at the expense of his life. When in a fight with an enemy it won't stop to attack another agent, nor will be stopped by pain or danger.

The Berserker is aggressive, undisciplined and non-persistent. Once in the arena he will attack fiercely its opponents, in a mad frenzy. He is insensitive to pain, and most times will not stop attacking to heal itself or even to dodge bullets.

In the following subsections we describe the design of each character. We start with the Veteran and proceed by showing the modifications made upon its design to achieve the different personas.

4.3.1 The Veteran

The personality requirements for the veteran are very similar to the requirements for an agent that wants to maximize its score over a series of games. This was the case of the agent presented at the experiments of section 4.2, so we use it as a basis for the Veteran. Figure 4.2 shows the Behavior Network and the control parameters used for this character.

The overall behavior of the agent could be described as follows: It started exploring the level and kept wandering until it found an enemy (*Explore*). Upon finding an enemy it started shooting (*ShootEnemy*) and approaching the enemy (*GoToEnemy*). Upon finding the enemy it switched weapons and used the more powerful weapon Impact Hammer (*FinalizeWithHammer*). After the enemy died, it stopped shooting (*StopShoot*) and started wandering again. When shot repeatedly it kept shooting and after a while stopped going to the enemy and started dodging subsequent shots. If the enemy stopped shooting it would go towards it again. When the agent was hurt in combat, if it knew the location of a medkit (*GoToKnownMedkit* had a high truth value), it would go to it and restore its health after a while. If when approaching the enemy the agent became with very low health, if there was a reachable medical kit (*MedKitReachable*), the agent would stop going to the enemy and go to the medkit while keeping shooting, unless it was close to the enemy, in which case it attempted a killing with the hammer (*FinalizeWithHammer*).

We see that this behavior matches the personality of an archetypical combat veteran: The agent is persistent when killing, heals itself when it is safe to do so and has the endurance to keep fighting even when being shot back, without panicking.

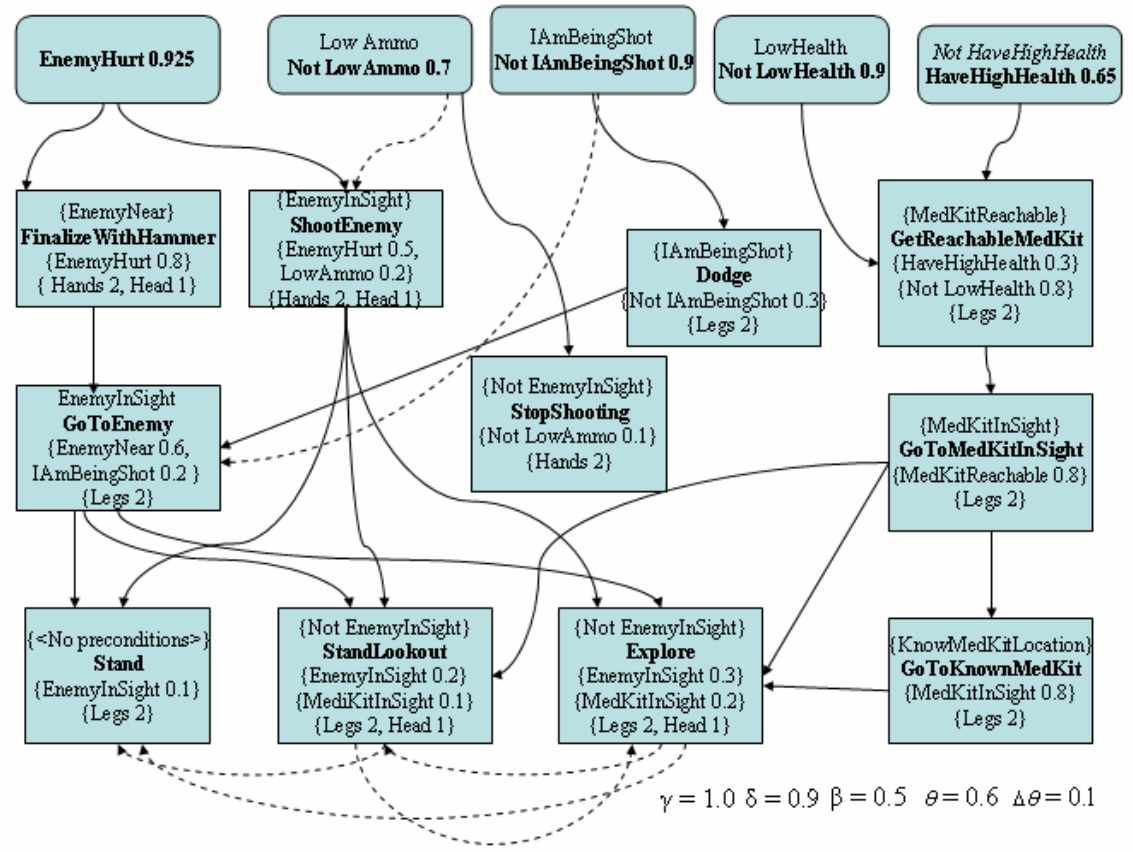

Figure 4.2: Veteran Behavior Network and Global Parameters. The predecessor link from *GoToKnownMedKit* to *StandLookout* is suppressed for clarity.

4.3.2 The Novice

As exposed in the beginning of this section, the novice lacks endurance, is less disciplined and more impulsive than the Veteran, but has similar goals. To achieve this lower discipline and greater impulsiveness we investigated lowering two global parameters, the inertia β and the global threshold θ.

We lowered the inertia to 0.1. This way the agent would be far more reactive. The agent, when shot, immediately attempted to dodge. Also, if while pursuing an enemy the agent became with very low health, upon spotting a reachable medkit the agent would immediately go to it. This resulted in poorer behavior – often the enemy was as hurt as the agent and an attack with the hammer would deliver victory. The same could be said of the pursuit – dodging is good, but only to prevent being extremely damaged. As dodging is not totally guaranteed to succeed it is usually a bad tactic to be away from the enemy and not approach it ever, specially in case of more than one enemy in the scene.

To augment the number of mistakes of the novice, we lowered the global threshold. This way we decreased the quality of action selection, as many modules surpassed the threshold simultaneously, and among modules that are above the threshold no one has priority over others. Now, often the agent shot the enemy even when it was very near and could hammer. For the extremely low value of 0.1, the agent also often just stood still (*Stand*) instead of exploring the level (*Explore*).

The best parameters we found to bring forth the character of the Novice were γ = 1.0, δ = 0.9, β = 0.1, θ = 0.25 and $\Delta\theta$ = 0.1.

4.3.3 The Coward

To bring forth the Coward working on the global parameters would be of little use, as he is as persistent as the Veteran and we have no reason to believe him to make decisions less thoughtfully. Instead, we worked on the goal strengths. We lowered *EnemyHurt* and raised *HaveHighHealth*. We left the global constants untouched.

The behavior of the coward could be thus described: It started exploring the level until he found an enemy. With an enemy in sight he started shooting .When shot back, if the enemy missed him, he would start dodging after a little while, for all subsequent shots. If actually hit he would go get the medkit immediately if there was one reachable. If there was none he would keep dodging and fighting until it had a low health. When it happened he would flee combat and go restore its health, even if he had to go all the way to a far known medkit.

We see that even though the agent is far more concerned with its health and could not be described as brave anymore a key point of its specification is missing: its active avoidance of engagement. To achieve this we implemented a new module for the network: *GoAwayFromEnemy*. With this module added, when the Coward spotted an enemy, he would go away from him while shooting (*GoAwayFromEnemy* and *ShootEnemy* were executed concurrently). Figure 4.3 shows the full network of the Coward character. Note that adding a new behavior was a simple modular operation, dispensing adjustments.

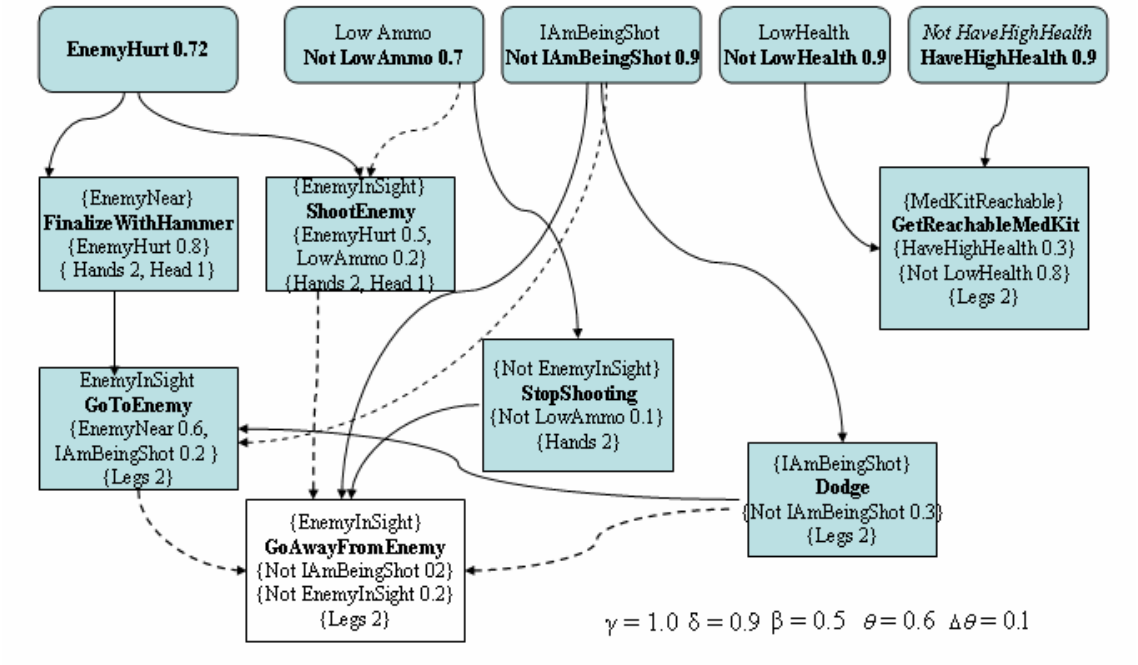

Figure 4.3: Coward Behavior Network

4.3.4 The Samurai

To transform the Veteran into a Samurai we worked on the goals strengths. We set *EnemyHurt* to 1.0, *NotIAmBeignShot* to 0.6, *NotLowHealth* to 0.5 and *HaveHighHealth*

to 0.4. With these strengths the agent will always approach the enemy instead of dodging bullets and will not stop to get medkits if in a fight. We verified that whenever he found an enemy it went towards it shooting and then attacked with the hammer (*FinalizeWithHammer*) if the enemy had not died yet. If there was no enemy in sight the Samurai would go after medkits to restore its health. For a gamer the Samurai displayed the exact behavior we desired: He was never disturbed by pain (shoots) or danger (low health) in his pursuit of an enemy and employed good tactics (shooting from afar and hammering when near).

4.3.5 The Berserker

To bring into being the mad berserker we started from the Samurai. We lowered even more its sensibility to pain by decreasing the importance of goal *LowHealth* and to bring forth his frenzy we diminished both its inertia and its global threshold. Lowering β made the agent very reactive and lowering θ made the agent take insane actions, such as shooting instead of hammering at close quarters.

The overall behavior of the berserker was as intended – he would not stop to dodge or heal while in combat and he fought madly, hammering and shooting everything that went into his path.

5 DISCUSSION

In this chapter we discuss the results of the experiments of the previous chapter, contextualize our work within a body of related work and point extensions to our work. We point extensions to enable the agent to play in other game modes, ways to incorporate learning capabilities and strategies for deeper personality modeling.

5.1 Agent Performance and Action Selection Quality

When compared to another robot built around finite-state machines that used different sensors and behaviors but the same low-level commands, our robot performed very well. This opponent being of another group prevented bias of our part on this first test. However, the question of whether this superior performance was due only to a better sensory-motor apparatus remained.

We decided to make another experiment, this time with an agent with identical sensory-motor abilities that used a different action selection algorithm. This way we could isolate the contribution of the extended behavior network to the agent's performance.

Our robot had significantly better scores in this second experiment.

This better score in itself is positive evidence of the importance of the extended behavior network for the agent's performance, as the sensors and behaviors in both agents were identical.

We may also regard as positive evidence the fact that our robot had a slightly higher hit rate (30%) but a much higher killing rate (100%). This was due to proper action selections – its sequences of actions made it die less and kill more. Searching the action log we found out that it stopped to heal itself only when necessary and was not easily disturbed while fighting. It engaged the enemy more often to deliver the deadly hammer blow.

When we compared the mean of the differences of the scores of our robot in the first and second experiments we saw that the difference was much smaller in the second one. We regard it as evidence that indeed a good part of the better score in the first experiment was due to better sensors and behaviors.

These two experiments, taken together, show that action selection quality plays an important role in agent performance and are evidence of the viability of extended

behavior networks as an action selection mechanism for complex agents with concurrent behaviors in complex, dynamic and continuous environments.

The experiments of section 4.1 enabled us to verify the properties of a "good enough" action selection mechanism in the 3D action game domain, namely, persistence, exploitation of opportunities, preference for actions that satisfy multiple goals, proper resolution of conflicts, performing of actions in sequences to achieve goals and sensible selection of concurrent actions. These experiments are additional evidence of the suitability of extended behavior networks for complex and dynamic environments in general, and for the 3D game domain in particular.

5.2 Personality Design and Global Parameter Setting

For an agent controlled by a behavior network, we illustrated three approaches to build its personality: changing the global parameters, changing the goal strengths and changing the network topology itself.

Changing the global parameters allowed us to control two key personality characteristics: Thoughtfulness (through the activation threshold θ) and persistence (through the inertia β).

A high activation threshold θ lead to better action selection as only actions that had a high activation could execute, that is, it required on average more activation spreading cycles to decide what to do next and also required higher executability of the modules. For an external observer it amounts to a thoughtful behavior, as an agent does mostly what seems effective and proper to its goals. We saw a thoughtful behavior typical of an experienced soldier in the Veteran and the thoughtless behavior of a madman in the Berserker. We should be cautious with putting θ at a too high value – for some agents it may make them too slow.

A high β leads to a persistent behavior: An agent only changes its behavior if there is a large or long change in its sensory information. This was the case of the Veteran taking some time to start dodging bullets. A single shot was not enough to make he interrupt his course of action. Symmetrically, the Novice changed actions due to slight changes in its sensors.

The predecessor link activation constant γ, the conflict link activation constant δ and the threshold decrease $\Delta\theta$ were not used to design agent personality. Rhodes(1996) points that $\delta > \gamma$ may lead to an overall behavior were the character is seen as stubborn - once a goal was satisfied or a behavior was executing it would be too hard to make the agent change its course of action so as to do any action that would undo the precondition of the goal or module that was the source of the conflict link.

Changing the goal strengths was our first try when changing the global constants could not lead to the desired behavior. This is somewhat harder because the strength of a goal must be set in relation to the other goals of the agent – what maters is their relative magnitude when compared to each other. Altering the goal strengths is the default way to alter deep personality characteristics, the very motivations and values of a character. This was the solution needed to implement both the Samurai and the Coward.

Finally for some cases we may have to add a whole new module. Adding a module to a behavior network is a straightforward operation – the network itself takes care of its

integration, with the automatic creation of predecessor and conflict links. It may be a time consuming option due to subtleties that may arise in the actual implementation of the module, if one does not exists yet. If we have a library of behavior modules, then this option is also easy.

Summing all up, we could make the reverse question: how do we build agents with different personalities from scratch? First we define the agent's goals and their relative importance. Next we assemble a set of modules capable of achieving these goals. Next we tune the global parameters to achieve the subtleties of the personality. Having one working agent, making others with radically different personalities is simple.

5.3 Comparison with Other Approaches to Personality Design

The design of agents with personality has a long tradition. Sophisticated models, with a focus on agent personality and interaction, have been developed over the last decade and the present. Usually they address the question "What is the best way to design an agent with personality and emotional traits capable of carrying out sophisticated interactions with humans and other agents?" They have shown promising results in the domains where applied, such as embodied conversational characters (CASSELL, 2000) and interactive drama (RAYES-ROTH, 1996).

Our work answers a different question: "Given that I have to design several complex agents capable of having good performance (or scores) in a real-time continuous and complex game environment in a short time span, how may I make them with different personalities?" This precludes solutions that require long processing or very complex design and favors solutions that produce a fast acceptable result. Sophisticated interaction with humans are not a concern as the interactions are quite simple and do not involve mood detection, gesture recognition or the exchange of roles.

The only previous application of behavior networks to character design that we are aware of is (RHODES, 1996). In this work, a behavior network model called PHISH-Nets was used to design the Big Bad Wolf and the Three Little Pigs of the famous kid's tale in a simple 3D environment. There was no pressure for the actions to be carried in real time. Most experiments investigated how the agent handled action failures and its capability to improvise. Despite its interesting results for character modeling we could not use this model to answer our question, unless it was drastically modified, as for complex real-time games we need to select several actions concurrently and deal with continuous quantities.

Isla and Blumberg's work on synthetic characters, particularly the architecture described in (ISLA et al, 2001), seems potentially fit to address the problem we pointed. It integrates learning capabilities and allows deeper emotional modeling, being more sophisticated and somewhat more complicate.

Other architectures have been proposed to the modern game environment domain. Of immediate interest is an implementation (HAWES, 2002) of the Cog-Aff architecture (SCHEUTZ and SLOMAN, 2002) that used an anytime planner (ZILBERSTEIN, 1996), A-UMCP. It was deployed in the Unreal Tournament domain, though for the game mode Capture the Flag. Although the Cog-Aff architecture explicitly takes into account agent personality into its design, it was not mentioned in (HAWES, 2002).

Finally we may cite the applications of the Soar (NEWELL, 1990) architecture to computer games (MAGERKO et al, 2004). Soar stands in a different stratum: It is a sophisticated cognitive architecture aimed at human-level intelligence, with a considerable learning curve. Usually the foci of these works revolved around cognitive plausibility and depth of the agent models (LAIRD, 2000-b). Soar seems a good choice when one's focus is fidelity and depth, but overkill for creating agents with simple personas.

5.4 Other architectures for computer game agents

In this section we present a brief overview of other architectures for game agents. Comparisons of architectures and the review of the whole pool of techniques employed in the development of game agents are outside the scope of our work.

For review of techniques we recommend (VAN WAVEREN, 2001) and (WOOD, 2004). The first illustrates in detail the building of a robot for a scenario similar to our own, using a combination of traditional game AI techniques. The second investigates the effects of several different technologies in the playability and perceived intelligence of games, using a Space Invaders clone as test-bed.

So, let us get back to the overview of game agent architectures.

The Excalibur architecture (NAREYEK, 2001) is proposed as a generic architecture for autonomous agents in complex game environments. Its distinctive feature is its ability to incorporate resources in its planning process in a sophisticated manner. This makes it interesting for most modern computer games, as hit points, ammunition, armor and other game features are as important to consider in planning as plan length.

Excalibur defines its planning task as a constraint-satisfaction problem and uses local search to guide the solution of the problem. The architecture provides a straightforward way to interleave sensing, acting and planning. Though suit to the action game domain, we know of no application of Excalibur to such a computer game genre.

FEAR stands for Flexible Embodied Animat aRchitecture (CHAMPANDARD, 2005). It is a framework for building learning reactive agents for games, with direct support for the game Quake II. This game is similar to Unreal Tournament – you are a warrior who must kill your enemies with several weapons. Agents in this framework are based in the Animat model (WILSON, 1991).

Champandard (2003) presents two agents built using this framework: An agent controlled by a subsumption architecture and another one that learned how to behave via reinforcement learning. Unfortunately, no account of the performance of the agents was found. As in our agent, most behaviors were built around finite-state machines. However the behaviors implemented were much more sophisticated and elaborate than the ones we used in our experiments.

We see that FEAR is better seen as a framework capable of supporting different architectures. One could use an extended behavior network to control the basic behaviors that come with the framework. The trickiest part of such an endeavor would be adapting the sensors (or creating additional sensors) to output proposition values to feed the behavior network.

Let us revisit Soar (NEWELL, 1990). Soar is a cognitive architecture based on production systems. All its procedural knowledge is encoded as goals and its learning

mechanism is chunking. It was applied to Quake II, where an agent built a model of its opponent and anticipated his opponent's actions based on this model (LAIRD, 2002-c). The authors report that the agent had a good overall performance, beating novice players and being challenging to the average human. However no comparison with other agents was presented.

Hundreds of rules were needed to bring about the Soar agent for Quake II. The following seems the general procedure to build a Soar game agent: Define the interface with the game, define new goals and operators for the Soar system and design hundreds of production rules. The interesting feature of Soar is that a major part of these production rules are domain-independent, making reuse of the knowledge base possible. Soar, though far heavier than most behavior-based systems, including extended behavior networks, is able to deal with 6-10 agents in Quake II (VAN LENT, 1999). Thus, Soar is an interesting option to build sophisticated agents for computer games.

Van Waveren (2001) describes the implementation of a Quake III Arena robot. Quake III Arena is yet another action game in the vein of Unreal Tournament. The robot used fuzzy sensors for some tasks and a network that resembled a hierarchical finite-state machine as its central decision making mechanism. This work shows how to apply and integrate a plethora of techniques to solve common problem faced by an action game agent.

5.5 Extensions

This section discusses some interesting extensions to the agents and behavior networks discussed in this work. The first subsection presents extensions to the network of chapter 3 to allow an agent to play in the game modes *Capture-the-Flag* and *Domination*. The second subsection discusses interesting uses of learning in extended behavior networks. In the third subsection we discuss extensions to allow deeper personality modeling.

5.5.1 Other Game Modes

There are two other game modes in Unreal Tournament: *Capture-the-Flag* and *Domination*.

In the capture-the-flag game, a team of agents need to take the flag of an opponent team and bring it to its base. While doing so they must prevent the enemies from getting their flag and carrying it to their home base.

For this game mode we need to coordinate our agents and make them able to communicate for a good strategy. As agent coordination and multi-agent strategy is beyond the scope of our work we point only the modifications necessary to the behavior network itself and its related sensors. However with just these modifications we would be not surprised if our agents performed decently.

We would need to add goals *TeamHasEnemyFlag*, *TeamHasOwnFlag*, *EnemyFlagInHomeBase* and *OwnFlagInHomeBase*. *TeamHasEnemyFlag* would have as context its negation. The same would apply to *EnemyFlagInHomeBase*.

The new modules would be *GoToReachableEnemyFlag*, *GoToEnemyFlagInSight* and *CarryFlagToHomeBase*.

Sensors for detecting an enemy flag (*SensorEnemyFlagInSight, SensorEnemyFlagReachable*), for knowing if our team is with the enemy flag (*SensorTeamEnemyFlag*) and if it holds its own flag (*SensorTeamOwnFlag*) would also be needed, as well as a sensor to know if the flag is in the base(*SensorFlagInBase*).

In domination games a team of agents must reach and remain at certain map locations for a certain time, that is, the whole group must secure a set of locations for a given duration.

The added goal would be *BeInDominationPoint*, the goal of reaching and staying at a domination point.

The new module would be *GoToFreeDominationPoint*, the module of going to a non-occupied domination point. Surely, we could break it in three modules to better explore game information, just as we did with the medical kit modules in chapter 3. The precondition of the module would be *FreeDominationPointInSight*, a domination point not occupied by a robot of the agent's team inside the field of view.

To detect if a domination point was occupied, we disregard enemies in it – this leads to our agent killing the enemy in the point and just seeking another point if a friendly bot is in it. We would call this sensor *SensorFreeDominationPoint*.

To make domination work well the agents would have to communicate and coordinate, even though we see that if the agents explored the entire level eventually every agent would reach a domination point, making the team succeed.

5.5.2 Learning and Adaptation

Learning can be very useful for a behavior network agent. We could use learning to improve the expected effects of our actions and to tune the global parameters of the network.

The expected values of the propositions in the effects list of the modules, both in our work and in Dorer's (DORER, 1999-c), were guessed and intuitively tuned. As the agent always has access to the value of these propositions before and after action execution, we could use learning techniques to enable to agent to make better predictions. We could use a reinforcement learning approach to correct our estimates or use a neural network to make a prediction based on a given window of past percepts. This would enable an agent to adapt to different opponents and environments. If the agent's aiming was bad or the opponent was very good on dodging bullets the expectation of *EnemyHurt* after *ShootEnemy* would decrease with time, probably favoring the selection of *FinalizeWithHammer*. A network augmented with such learning capabilities would probably have better action selection.

The global parameters, in (MAES, 1989), (TYRRELL, 1993), (RHODES, 1996), (GOETZ, 1997), (DORER, 1999-c) and (PINTO, 2005-a) were set by experimentation. Dorer(1999-b) set the parameters using a brute force approach. He found the best value for a parameter while maintaining the others fixed. As the parameters are dependent upon one another, he repeated this procedure for each one in a round-robin fashion until the improvement in the overall score was very low.

This setting of parameters seems amenable to a treatment with genetic algorithms, specially if we have only performance in mind. We could use the total score of the agent, as computed in the experiments of the previous chapter, as the fitness function.

Darran Singleton (2002) used a genetic algorithm to set the global parameters of a variation if Maes (1989) behavior network. However, the modifications to the original behavior network model were so radical that the resulting network ended up being a quite different model. In Singleton's network, instead of having only the global parameters described in chapter 2, each behavior had its own threshold and each link had its own activation influence constant. The genetic algorithm found values for all these parameters. In this work, Singleton shows that the algorithm converged. As generations went on, the performance of the agent got better, in an environment similar to Tyrrell's(1993).

5.5.3 Deeper personality

One interesting extension for an extended behavior network would be the adaptive tuning of its global parameters so as to achieve different personality traits. Our focus here would be not performance, but personality depth and fun.

An easy approach is examining a given amount of the percept history of the agent and increasing or decreasing each of the global parameters. For instance, if the agent was safe we could increase its global threshold, so as to make it more deliberative. If the agent was attacked constantly in the last 3 minutes we could decrease its inertia, making it more reactive. This way, agent personality would vary smoothly at least among the dimensions of reactivity and thoughtfulness.

Another approach would be selecting between pre-defined sets of parameters, where each set corresponded to a given personality or mood. This would be the case of an agent changing from a Veteran to a Berserker. We could allow the values of the goals to be changed too, so deep personality changes could happen – our agent could change between all the five stereotypes discussed in the previous chapter. In fact, if needed one could design a sophisticated emotional module and use it to tune the goal strengths and global parameters. The "mood" and personality of the agent would be able to change its very core values, at least for some time.

However, if such deep personality modeling is needed, one could as well seek other models. Extended Behavior Networks are a competitive model for action selection and performance in dynamic and complex environments. Their strength, regarding agent personality, lies in their simplicity and flexibility. For agents with deep personalities, other models are probably better, such as (CASSELL, 2000), (LAIRD, 1999-b) and (RAYES-ROTH, 1996).

6 CONCLUSION

We have seen that the extended behavior network is a good action selection mechanism for a complex game agent with complex goals and actions situated in a dynamic and continuous environment.

We verified the properties of a good enough action selection mechanism in the 3D action game domain, namely, persistence, exploitation of opportunities, preference for actions that satisfy multiple goals, proper resolution of conflicts, performing of actions in sequences to achieve goals and sensible selection of concurrent actions.

We have seen positive evidence that extended behavior networks are a good action selection mechanism for continuous and dynamic environment and that good action selection is crucial for agent performance. Our agent had better scores than an agent that was identical to it, except for the decision mechanism. It also performed better than a totally different agent. The smaller difference in scores in the first experiment remembers us that for this game genre the sensory-motor apparatus is also very important, so we have to be cautious when attributing credit to each component of an agent.

From an agent design perspective the strong points of extended behavior networks are its modularity and easy integration of new goals and behaviors. One may add a new module and the network takes care of its interaction with other modules. This enables the designer to develop one module at a time, and maintain a library of basic behaviors.

To build an agent one starts by setting goals that reflect the character values and motivations. Next, one proceeds by assembling a set of modules capable of achieving those goals (modules that have the goal conditions as effects). Finally, by adjusting the goal strengths and global parameters one tunes the overall behavior of the agent.

We have used an approach of empirical investigation and informed guessing of the parameters to achieve good agent performance. However we believe that a genetic algorithm could be applied at this phase.

When our focus is agent personality we may tune the network parameters, particularly the inertia and the activation threshold, to achieve different personalities. The limitation is that personalities built in this way are static and simple, allowing us to achieve only stereotypes.

Our experiments in stereotype design show that for simple personas extended behavior networks are a competitive solution, as they are simple and enable us to make very different agents by the mere adjusting of constant values.

In section 5.5 we have seen ways to achieve deeper personality modeling, namely the dynamic adjusting of global parameters and stereotype selection based on the recent percept history.

We see that extended behavior networks are complementary to other common game AI techniques, such as finite-state machines, reinforcement learning, neural networks and genetic algorithms. The EBN is responsible to selecting what actions execute at each instant, but the internals of each behavior are better grasped with other approaches. Learning and optimization techniques may be applied to improve the basic behaviors and the behavior network itself.

Our next steps are incorporating learning so as to enable the agent to adjust the expected values of the action effects and investigating the use of genetic algorithms to set the global parameters. Once we have these improvements done we intend to extend the network to enable our agent to play in Domination and Capture-the-Flag modes.

REFERENCES

ABRAGAMES. **Plano Diretor da Promoção da Indústria de Desenvolvimento de Jogos Eletrônicos no Brasil**. 2004.

AGRE, P.; CHAPMAN, D. Pengi: An implementation of a theory of activity. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 6., 1987. **Proceedings…** [S.l]: Morgan Kaufmann, 1987. p.268-272.

BROOKS, R. A Robust Layered Control System for a Mobile Robot. **IEEE Journal of Robotics and Automation**, New York, V. RA-2, nº1, 1986.

CASSELL, J. et al. Human Conversation as a System Framework: Designing Embodied Conversational Agents. In: CASSELL, J. et al. (Ed.), **Embodied Conversational Agents**, Cambridge, MA: MIT Press, 2000. p. 29-63.

CHAMPANDARD, A. **AI Game Development.** [S.l]: New Riders Publishing, 2003.

DORER, K. Extended Behavior Networks for the Magma Freiburg Team. In: **RoboCup-99 Team Descriptions for the Simulation League**. [S.l]: Linkoping University Press, 1999-a. p. 79-83.

DORER, K. Behavior Networks for Continuous Domains Using Situation-Dependent Motivations. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 16., 1999. **Proceedings…** [S.l]: Morgan Kaufmann, 1999-b.

DORER, K. **Motivation, HandlungsKontrolle und Zielmanagement in autonomen Agenten**. 1999-c. PHD Thesis. Albert-Ludwigs, Universität Freiburg.

DORER, K. Concurrent Behavior Selection in Extended Behavior Networks. In: ROBOT WORLD CUP SOCCER GAMES AND CONFERENCE, 4., 2000. **Proceedings…** Melbourne: 2000-a.

DORER, K. The magmaFreiburg Soccer Team. In: RoboCup, 3., 1999. **RoboCup-99:** Robot Soccer World Cup III, Berlin: Springer, 2000.

DORER, K. Extended Behavior Networks for Behavior Selection in Dynamic and Continuous Domains. In: ECAI**,** 2004. **Proceedings…** Valencia: 2004.

ESA – Entertainment Software Association. "Computer and Video Game Software Sales Reach Record $7.3 Billion in 2004". Available at <http://www.theesa.com/archives/2005/02/computer_and_vi.php>. Visited on June 16, 2005.

FRANKLIN, S. **Artificial Minds**. Cambridge, Massachusetts: MIT Press, 1995.

GAMEBOTS. Available at <http://www.planetunreal.com/gamebots/docapi.html> Visited on June 30, 2005.

GOETZ, P. **Attractors in Recurrent Behavior Networks**. 1997. Phd Thesis. Buffalo, University of New York.

HAWES, Nick. An Anytime Planning Agent For Computer Game Worlds. In: INTERNATIONAL CONFERENCE ON COMPUTERS AND GAMES, 3., 2002. **Proceedings…** Edmonton: CG'02, 2002. p. 1-14.

ISLA, D., BURKE, R., BLUMBERG, B. et.al. A Layered Brain Architecture for Synthetic Characters. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2001. **Proceedings…** Seattle: IJCAI, 2001. p.1051-1058.

JAVABOTS. Available at <http://utbot.sourceforge.net/> Visited on Mar 28, 2005.

KAMINKA, G. et al. "GameBots: a flexible test bed for multiagent team research". **Communications of the ACM**. V. 45 , nº 1, 2002. p.43-45.

KLINE, C.; BLUMBERG, B. The Art and Science of Synthetic Character Design. In: SYMPOSIUM ON AI AND CREATIVITY IN ENTERTAINMENT AND VISUAL ART, 1999. **Proceedings…** Edinburgh: 1999.

LAIRD, J.; LENT, M. von. Human-level AI's Killer Application: Interactive Computer Games. In: AAAI FALL SYMPOSIUM, 2000. **Proceedings…** North Falmouth: AAAI Press, 2000-a.

LAIRD, J. et al. Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. In: AAAI FALL SYMPOSIUM, 2000. **Proceedings…** North Falmouth: AAAI Press, 2000-b.

LAIRD, J. It Knows What You're Going To Do: Adding Anticipation to a Quakebot. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, 5., 2000. **Proceedings…** [S.l], 2000-c.

MAES, P. How to do The Right Thing. **Connection Science Journal**, [S.l], V. 1, nº 3, 1989.

MAES, P. A Bottom-up Mechanism for Behavior Selection in an Artificial Creature. In: INTERNATIONAL CONFERENCE ON SIMULATION OF ADAPTIVE BEHAVIOR, 1., 1991. **Proceedings…** [S.l.]: Mit Press/Bradford Books, 1991.

MAES, P. Modeling Adaptive Autonomous Agents. In: **Artificial Life I.** 1994.

MAGERKO, B., LAIRD, J. et al, AI Characters and Directors for Interactive Computer Games. In: ARTIFICIAL INTELLIGENCE CONFERENCE, 2004. **Proceedings…** San Jose: AAAI Press, 2004.

MINSKY, M. **The Society of the Mind**. New York: A. Touchstone Book, 1986.

MÜLLER, K. **Roboterfußball:** Multiagentensystem CS Freiburg. 2001. Diplomarbeit. Univ. Freiburg, Germany.

NAREYEK, A. Beyond the Plan-Length Criterion. In: NAREYEK, A. (Ed.). **Local Search for Planning and Scheduling**. Berlin: Springer, 2001. p. 55-78. (Lecture notes in Artificial Intelligence, 2148).

NAREYEK, A. Artificial Intelligence in Computer Games: State of the Art and Future Directions. **ACM Queue**, [S.l], V. 1, N° 10, 2004.

NEBEL, B. and Y. BABOVICH. Goal-Converging Behavior Networks and Self-Solving Planning Domains, or: How to Become a Successful Soccer Player. In: INTERNATIONAL JOINT CONFERENCE ARTIFICIAL INTELLIGENT, 3., 2003. **Proceedings…** Acapulco: 2003.

NEWELL, A. **Unified Theories of Cognition**. Cambridge, MA: Harvard University Press, 1990.

NILSSON, N. J. "STRIPS: A New Approach to the application of Theorem Proving to Problem Solving". **Artificial Intelligence**, [S. l], V. 5, n° 2, 1971.

PINTO, H.; ALVARES, L.O. An Extended Behavior Network for a Game Agent. In: ENCONTRO NACIONAL DE INTELIGÊNCIA ARTIFICIAL, 5., 2005. **Anais...** Brazil: 2005-a.

PINTO, H.; ALVARES, L.O. Extended Behavior Networks and Agent Personality: Investigating the Design of character Stereotypes in the Game UnrealTournament.In: INTERNATIONAL WORKING CONFERENCE ON INTELLIGENT VIRTUAL AGENTS, 15., 2005. **Proceedings…** Greece: 2005-b.

PINTO, H. S. C. **Redes de Comportamentos**. 2004. 47f. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

RAYES-ROTH, B. Improvisational puppets, actors, and avatars. In: COMPUTER GAMES CONFERENCE, 1996. **Proceedings…** [S.l.]: 1996.

RHODES, B. **PHISH-Nets: Planning Heuristically in Situated Hybrid Networks** . 1996. MSc Thesis. USA, MIT.

RHODES, B. **Pronomes in Behavior Nets**. [S.l]: MIT Media Lab,1995. (Technical Report # 95-01).

SCHEUTZ, M.; SLOMAN, A. A Framework for Comparing Agent Architectures. In: UKCI, 2002. **Proceedings…** Birmingham, 2002.

SINGLETON, D. **An Evolvable Approach to the Maes Action Selection Mechanism**. 2000. M Sc Thesis. UK, University of Sussex.

TYRRELL, T. An Evaluation of Maes Bottom-up mechanism for behavior selection. **Journal of Adaptive Behavior**, [S.l], V. 2, n° 4, 1994.

TYRRELL, T. **Computational Mechanisms for Action Selection**. 1993. PhD Thesis. UK, University of Edinburgh.

UNREALTOURNAMENT. Availabel at <http://www.unrealtournament.com> Visited on June 30, 2005.

VAN LENT, M.; LAIRD, J. Developing an Artificial Intelligence Engine. In: GAME

DEVELOPERS CONFERENCE, 1999. **Proceedings…** [S.l]: 1999. p. 577-588.

YISKIS, E. **A Subsumption Architecture for Character-Based Games.** In: RABIN, S. (Ed.). **AI Game Programming Wisdom 2**. Hingham: Charles River Media, 2003.

WILSON, S. The Animat Path to AI. In: INTERNATIONAL CONFERENCE ON SIMULATION OF ADAPTIVE BEHAVIOR, 1991. **From animals to animats**: proceedings. Cambridge: MIT, 1991. p. 15-21.

WOOD, O. E. **Autonomous Characters in Virtual Environments:** The technologies involved in artificial life and their affects of perceived intelligence and playability of computer games. 2004. M Sc. Thesis. Durham, University of Durham.

ZILBERSTEIN, S. Using Anytime Algorithms in Intelligent Systems. **AI Magazine**, [S.l], V. 17, n° 3, 1996.

APPENDIX A GAMEBOTS MESSAGES

The text and messages below are reproduced and adapted from (GAMEBOTS, 2005)

Messages from the server are always of the form "MSGTYPE {arg1 arg1val} {arg2 arg2val}..." (Real message of course won't have the quotes on either end. You can write a parser based on the following assumptions (it won't choke on anything you are sent, but may require additional parsing of some messages):

- All characters up to, but not including the first space are the message type. (All message types are currently 3 characters long, but best not to live by that assumption).
- Everything else in the message will be in the form of attr/val pairs enclosed by "{}"
- The attribute name in a attr/val pair consists of every character up to, but not including, the first space.
- The value includes all characters after the space terminating the attr up to the "}" and may include spaces.

Thus a correct parsing of "MSG {Id Player-1} {String Attack the base!} {Location 12,23,34}" would be:

Message type = "MSG"
Attr1Type = "Id"
Attr1Value = "Player-1"
Attr2Type = "String"
Attr2Value = "Attack the base!"
Attr3Type = "Location"
Attr3Value = "12,23,34"

Commands that your client sends to the server should follow the same basic format. A message type, followed by a space, followed by attr/val pairs enclosed in "{}". Each attr in a attr/val pair should be space terminated.

Data types notes:

Most measurements of rotation and location sent to your bot will be in absolute terms using UT's measurements. For this reason, it is helpful to know a little about some Unreal Tournament data types and measurements before you read the Network API.

Location is described in UT units. They have no direct scale correspondence to the real

world, but as an idea, a character in the game has a collision cylinder (a cylinder tightly bounding the graphical model that defines how close something has to be before it collides with the character) 17 units in radius and 39 units tall. Location is always passed in "x,y,z" order and format (values separated by commas.

Rotation is also defined by three ordered values, "Pitch,Yaw,Roll". Yaw is side to side, pitch up and down, and Roll the equivalent of doing a cartwheel. You probably won't need Roll, so don't sweat it. A full rotation using UT's measurements is 65535. To convert the values you are sent to radians, divide by 65535 and multiply by 2 * pi.

Sensory Messages:

The sensory messages sent to your client from the game consist of two types, synchronous and asynchronous. In the lists below, each message type is listed, along with current argument types.

Synchronous messages will come to your client in a batch at a configurable interval. They include things like a visual update of what the bot sees and a status report of the bot itself. At the start of a batch, the server transmits a "BEG" message marked with a timestamp. All messages received until an "END" message with the same timestamp are part of the synchronous batch. They are all sent at the same instant of gametime and thus refer to a single discrete state of the game.

Asynchronous messages on the other hand come as events happen in the game. (Although they will never appear between a "BEG" and its associated "END"). They represent things that may happen at any point in the game at random, less frequent intervals such as taking damage, a message broadcast by another player, or running into a wall. You can always be sure that event triggering an asynchronous message occurred in game time between the synchronous batches before and after it, but there is no guarantee that an asynchronous message refers to the same discrete state of the game that any other message does.

Synchronous Messages:

- BEG - begin of a synchronous batch
    - Time - timestamp from the game
- SLF - information about your bot's state.
    - Id - a unique id, assigned by the game
    - Rotation - which direction the player is facing in absolute terms
    - Location - an absolute location
    - Velocity - absolute velocity in UT units
    - Name - players human readable name
    - Team - what team the player is on. 255 is no team. 0-3 are red, blue, green, gold in that order
    - Health - how much health the bot has left. Starts at 100, ranges from 0 to 200.

    - Weapon - weapon the player is holding. Weapon strings to look for include: "ImpactHammer", "Enforcer", "Translocator", "GESBioRifle", "ShockRifle", "PulseGun", "Minigun2", "UT_FlakCannon", "UT_Eightball", "WarheadLauncher"
    - CurrentAmmo - How much ammo the bot has left for current weapon
    - Armor - how much armor the bot is wearing. Starts at 0, can range up to 200.

- GAM - information about the game
    - PlayerScores - player score will have a list of values - one for each player in the game. Each value will be a list with two values. The first is the id of the player and the second that player's score. (e.g. "GAM {PlayerScore {player1 2} {player2 5}...")
    - TeamScores - like PlayerScore, but for teams. Team is identified by the team index (same number used to describe team for PLR and SLF messages. Not sent in normal deathmatch.
    - DomPoints - like the previous two, this is a multivalued message. This will have one item for each domination point in a Domination game. First value will be Id of the DOM point, the second will be the index of the team that owns the domination point.
    - HaveFlag - sent in CTF games if the bot is carrying an enemy's flag. Value is the team number of whose flag you have.
    - EnemyHasFlag - sent in CTF games if the bot's team's flag has been stolen. Value is meaningless.

- PLR - Another character (bot or human) in the game. Only reports those players that are visible. (within field of view and not occluded).
    - Id - a unique id for this player, assigned by the game
    - Rotation - which direction the player is facing in absolute terms
    - Location - an absolute location for the player
    - Velocity - absolute velocity in UT units
    - Team - what team the player is on.
    - Reachable - true if the bot can run to this other player directly, false otherwise. Possible reasons for false: pit or obstacle between the two characters
    - Weapon - what class of weapon the character is holding.

- NAV - a path node in the game. Pathnodes are invisible (at least to humans) objects placed around a level to define paths for the built in bots to follow. They

provide a totally connecteed graph that spans almost all of the level. Note the Mutator called "Path Markers" that, when added to a game makes the path nodes visible to human players as a debugging aid.

  - Id - a unique id for this pathnode, assigned by the game
  - Location - an absolute location
  - Reachable - true if the bot can run here directly, false otherwise

- MOV - a "mover". These can be doors, elevators, or any other chunk of architecture that can move. They generally need to be either run into, or activated by shooting or pressing a button. We are working on ways to provide bots with more of the information they need to deal with movers appropriately.
  - Id - a unique id for this mover, assigned by the game
  - Location - an absolute location
  - Reachable - true if the bot can run here mover, false otherwise
  - DamageTrig - true if the mover needs to be shot to activated.
  - Class - Class of the mover.
- DOM - identical attributes to NAV above except for Controller (see below). A domination point in a domination game.
  - Controller - which team controls this point
- FLG - a flag. (Only for CTF games).
  - Id - a unique id for this flag, assigned by the game
  - Location - an absolute location of the flag
  - Holder - the identity of player/bot holding the flag (only sent if flag is being carried).
  - Team - the team whose flag this is
  - Reachable - true if the bot can run here directly, false otherwise
  - State - whether the flag is "Held" "Droped" or "Home"
- INV - an object on the ground that can be picked up
  - Id - a unique id for this inventory item, assigned by the game.
  - Location - an absolute location
  - Reachable - true if the bot can run here directly, false otherwise
  - Class - a string representing type of object

- END - end of a synchronous batch
    - Time - timestamp from the game

Asynchronous Messages:

- NFO - helpful info about the game provided right after connection made to server. Your client should wait for this message BEFORE trying to send "init" back to the server.
    - Gametype - What you are playing (BotDeathMatchPlus, BotTeamGame, BotDomination)
    - Level - name of map in play
    - TimeLimit - maximum time game will last (if tied at end, goes into sudden death overtime)
    - FragLimit - number of kills needed to win game (BotDeathMatchPlus only)
    - GoalTeamScore - number of points a team needs to win game (BotTeamGame, BotDomination)
    - MaxTeams - max number of teams. valid team range will be 0 to (MaxTeams - 1) (BotTeamGame, BotDomination)
    - MaxTeamSize - Max number of players per side (BotTeamGame, BotDomination)
- AIN - added inventory. Bot got new inventory item
    - Id - a unique id for this inventory item, assigned by the game. Unique, but based on a string describing the item type.
    - Class - a string representing type of object
- VMS - recieved message from global chat channel
    - String - a human readable message sent by another player in the game on the global channel
- VMT - recieved message from global chat channel
    - String - a human readable message sent by a team mate in the game on the private team channel
- VMG - recieved tokenized message from another player. If you want to use these, enter the game as a player and use the voice menu (by default press the key "V" while playing). Send the messages you want to use to use and have a client log the incoming messages to figure out Types and Ids.
    - Sender - unique id of player who sent message

    - Type - type of message (e.g. Command, Taunt, etc...)
    - Id - message id. specifies which message is being sent
- ZCF - foot changed zones. Feet of bot changed from one artificial area in the game to another (can tell you when entered water or lava or some such)
    - Id - unique id of zone entered
- ZCH - head changed zones. Its ok if feet are under water, but having your head under can mean trouble...
    - Id - unique id of zone entered
- ZCB - bot changed zones. Entire bot now in new zone
    - Id - unique id of zone entered
- CWP - bot changed weapons. Possibly as a result of a command sent by you, maybe just because it ran out of ammo in its old gun. (bots autoswitch when empty, just like human players)
    - Id - unique id of new weapon, based on the weapon's name
    - Class - a string representing type of weapon
- WAL - collided with a wall. Note it is common to get a bunch of these when you try to run through a wall (or are pushed into one by gunfire or something).
    - Id - unique id of wall hit
    - Normal - normal of the angle bot colided at.
    - Location - absolute location of bot at time of impact
- FAL - bot just hit a ledge. If walking, will not fall. If running, you are already falling by the time you get this.
    - Fell - True if you fell. False if you stopped at edge.
    - Location - absolute location of bot
- BMP - bumped another actor
    - Id - unique id of actor (actors include other players and other physical objects that can block your path.)
    - Location - location of thing you rammed
- HRP - hear pickup. You head someone pick up an object from the ground
    - Player - unique ID of player how picked up the object
- HRN - hear noise. Maybe another player walking or shooting, maybe a bullet

hitting the floor, or just a nearby lift going up or down.

  - Source - unique ID of actor making the noise

- SEE - see player. A message generated by the engine periodically (on the order of 1 or 2 times a second) when another player is visible by you. Possibly usefull if you have the delay between synchronous updates very long. In that case, this can prevent someone from walking by unseen. May be depricated.
  - Id - a unique id for this player, assigned by the game
  - Rotation - which direction the player is facing in absolute terms
  - Location - an absolute location for the player
  - Velocity - absolute velocity in UT units
  - Team - what team the player is on.
  - Reachable - true if the bot can run to this other player directly, false otherwise. Possible reasons for false: pit or obstacle between the two characters
  - Weapon - what weapon the character is holding.
- PRJ - incoming projectile likely to hit you. May give you a chance to dodge.
  - Time - estimated time till impact
  - Direction - rotation value that the projectile is coming from. Best chance to dodge is to probably head off at a rotation normal to this one (add ~ 16000 to the yaw value)
- KIL - some other player died
  - Id - unique ID of player
  - Killer - unique ID of player that killed them if any (may have walked off a ledge)
  - DamageType - a string describing what kind of damage killed them
- DIE - this bot died
  - Killer - unique ID of player that killed them if any (may have walked off a ledge)
  - DamageType - a string describing what kind of damage killed them
- DAM - took damage
  - Damage - amount of damage taken
  - DamageType - a string describing what kind of damage

- HIT - hurt another player. Hit them with a shot.
    - Id - unique ID of player hit
    - Damage - amount of damage done
    - DamageType - a string describing what kind of damage
- PTH - a series of pathnodes in response to a getpath call from client
    - Id - an id matching the one sent by client. Allows bot to match answer with right querry.
    - Multiple pathnodes: A variable number of attr items will be returned, one for each pathnode that needs to be taken. They will be listed in the order in which they should be travled to. Each one is of form "{0 id 3,4,5}", with the number of the node (starting with 0) followed by a space, then a unique id for the node (will never have a space) then a location of that node.
- RCH - a boolean result of a checkreach call.
    - Id - an id matching the one sent by client. Allows bot to match answer with right querry.
    - Reachable - true if the bot can run here directly, false otherwise
    - From - exact location of bot at time of check
- FIN - no attributes. Sent when game is over.

Commands:

Your bot takes action in the world by transmitting commands to the server. They are formated like the server messages - a command name, followed by zero or more arguments with values, each surrounded by "{}" and seperated by spaces. For example the message to initialize your bot with a name of MYBOT on team 1 would look like this (sans quotes):
"init {Team 1} {Name MYBOT}"
Parsing at the server is case insenstive. It should not matter what case you send commands, argument names, and their values in. Arguments may also be suplied in any order. The above example could have passed the name before the team and the command would have been the same. There are however some commands that have multiple options for how to specify a desired value. A good example is the runto command, which can take the id of an object or player, or a location in the world. You can send either or both, but the server will only use the first one it parses (order for each command type is listed below).
Note that most commands have persistant effects. Movement and rotation, once started, will continue until you reach your destination. Start shooting and you will keep shooting. There is **NO** advantage to sending commands repeatedly. It is quite likely that some kind of filter will be put in to discourage spamming the server.

- INIT - message you send to spawn a bot in the game world. You must send this message before you have a character to play in the game. **DO NOT SEND UNTIL YOU RECIEVE NFO MESSAGE FROM SERVER.**
    - Name - Desired name. If in use already or argument not provided, one will be provided for you.
    - Team - Prefered team. If illegal or no team provided, one will be provided for you. Normally a team game has team 0 and team 1. In BotDeathMatchPlus, team is meaningless, but this will still set you skin color to match what you select.
- SETWALK - set whether you are walking or running (default is run). Note that walking only applies to RUNTO command. STRAFETO always moves at run speed.
    - Walk - Send "True" to go into walk mode - you move at about 1/3 normal speed, make less noise, and won;t fall off ledges. Send anything else to run.
- STOP - stop all movement/rotation
- JUMP - causes bot to jump. Not very useful yet, working on this one.
- RUNTO - turn towards and move directly to your destination. May specify destination via either Target or Location argument, will be parsed in that order. (i.e. if Target provided, Location will be ignored). If you select an impossible place to head to, you will start off directly towards it until you hit a wall, fall off a cliff, or otherwise discover the offending obstacle.
    - Target - the unique id of a player/object/nav point/whatever. The object must be visible to you when the command is recieved or your bot will do nothing. Note that something that was just visible may not be when the command is recieved, therefore it is recomended you supply a Location instead of a Target.
    - Location - Location you want to go to. May be provided as space or comma delimeted. ("40 50 45" or "40,50,45"). May also be provided as three seperate arguement value pairs, one each for X Y and Z ("{X 40} {Y 50} {Z 45}").
- STRAFE - like RUNTO, but you move towards a destination while facing another point/object.
    - Location - Location you want to go to. May be provided as space or comma delimeted. ("40 50 45" or "40,50,45"). May also be provided as three seperate arguement value pairs, one each for X Y and Z ("{X 40} {Y 50} {Z 45}").
    - Target - the unique id of a player/object/nav point/whatever that you want to face while moving. Must be visible to you currently.

    - Focus - a location value of where to face while moving. Follows same rules as location for what to send. Used only if no Target.

- TURNTO - specify a point, rotation value or object to turn towards.
    - Target - the unique id of a player/object/nav point/whatever that you want to face. Must be visible.
    - Rotation - Rotation you want to spin to. May be provided as space or comma delimeted. ("0 50000 0" or "0,50000,0") and should be in absolute terms and in UT units (2pi = 65535 units). May also be provided as three seperate arguement value pairs, one each for Pitch Yaw and Roll ("{Pitch 0} {Yaw 50000} {Roll 0}"). Used only if no target provided.
    - Location - Location you want to face. Normal rules for location. Only used if no Target or Rotation.
- ROTATE - turn a specified amount.
    - Amount - amount in UT units to rotate. May be negative to rotate counter clockwise.
    - Axis - if provided as Vertical, rotation will be done to Pitch. Any other value, or not provided, and rotation will be to Yaw.
- SHOOT - start firing your weapon
    - Location - Location you want to shoot at. Normal rules for a location specification.
    - Target - the unique id of your target. If you specify a target, and it is visible to you, the server will provide aim correction and target leading for you. If not you just shoot at the location specified. Note you still must provide location.
    - Alt - If you send True to this you will alt fire instead of normal fire. For normal fire you don't need to send this argument at all.
- CHANGEWEAPON - start firing your weapon
    - Id - Unique Id of weapon to switch to. If you just send "Best" as Id, the server will pick your best weapon that still has ammo for you. Obtain Unique Id's from AIN events.
- STOPSHOOT - stop firing your weapon
- CHECKREACH - check to see if you can move directly to a destination without hitting an obstruction, falling in a pit, etc...
    - Target - the unique id of a player/object/nav point/whatever. Must be visible.

    - Location - Location you want to go to. Normal location rules. Only used if no Target is sent.
    - Id - message id made up by you and echoed in response so you can match up response with query
    - From - exact location of bot at time of check

- GETPATH - get a path to a specified location. An ordered list of path nodes will be returned to you.
    - Location - Location you want to go to. Normal location rules.
    - Id - message id made up by you and echoed in response so you can match up response with query
- MESSAGE - send a message to the world or just your team. This will likely have some restrictions placed on it soon.
    - String - string to send.
    - Global - If True it is sent to everyone. Otherwise (or if not specified), just your team.
- PING - if for some reason 10 updates a second or whatever your default is isn't frequent enough connection detection for your tastes, use PING. Server will return "PONG".

APPENDIX B CONTRIBUIÇÕES

No projeto de um agente de um jogo uma das preocupações centrais é como fazer a seleção de ações de modo que o agente exiba um comportamento orientado aos objetivos.

Se o agente tiver muitos objetivos, alguns deles conflitantes, nossa tarefa se torna mais complicada. Se, além de ter muitos objetivos, o agente tiver que considerar muitos fatores ao mesmo tempo e estiver em um ambiente dinâmico, nós teremos um problema difícil de ser tratado.

Abordagens baseadas em busca se tornam impraticáveis devido ao tamanho do espaço de busca e planejamento tradicional se torna muito mais difícil,dado que o ambiente pode ter mudado quando nosso plano estiver concluído.

Redes de comportamentos (MAES, 1989) foram propostas como um mecanismo de seleção de ações para selecionar ações boas o suficiente em ambientes dinâmicos e complexos. Elas favorecem ações que contribuem para mais de um objetivo e ações que façam parte de uma seqüência em execução. As redes de comportamentos são rápidas, robustas e reativas.

As redes de comportamentos estendidas (DORER, 1999a) são uma extensão para ambientes contínuos capaz de selecionar ações de maneira concorrente e especificar objetivos dependentes de contexto. Foram aplicadas com sucesso na Robocup[12]. Nosso trabalho foi a primeira aplicação deste modelo em jogos de computador (PINTO, 2005-a).

PHISH-Nets (RHODES, 1996), um modelo de redes de comportamentos capaz de selecionar apenas uma ação por vez, foi aplicado à modelagem de personagens, com bons resultados. Apesar das redes de comportamentos estendidas serem aplicáveis a um conjunto de domínios maior, nunca foram usadas para modelagem de personagens antes de nosso trabalho (PINTO, 2005-b).

Unreal Tournament é um jogo de ação tridimensional. Neste jogo, no modo Death Match, o agente é um guerreiro que deve eliminar os oponente em uma arena. O agente interage com várias entidades em tempo real, aliados e inimigos, em diferentes cenários. Existem várias armas e itens disponíveis. O repertório de ações é grande (pular, andar,

---

[12] See (DORER, 1999) and (DORER, 2004). The Magma-Freiburg team, built using extended behavior networks, was the vice-champion of Robocup-1999.

correr, virar, agachar, atirar, pegar item,etc) e o agente pode executar mais de uma ação ao mesmo tempo, como atirar enquanto pula.O agente tem objetivos conflitantes - lutar e manter sua integridade física, por exemplo.

Jogos de computador se tornaram uma aplicação importante, ultrapassando o faturamento da indústria de cinema nos EUA por 2 anos consecutivos (ESA, 2005). Os governos da Austrália, Coréia do Sul e Brasil têm programas especiais de apoio à pesquisa e desenvolvimento de jogos (ABRAGAMES, 2004). O avanço da capacidade das placas gráficas dos PCs e consoles liberou poder de processamento para os jogos, que passaram a ter na inteligência artificial um de seus principais fatores competitivos (NAREYEK, 2004).

Isso levou a um aumento do interesse na aplicação de técnicas avançadas de inteligência artificial em jogos de computador. A pesquisa em IA pra jogos teve um aumento súbito, como mostram (VAN WAVEREN, 2001), (RABIN, 2002), (BUCKLAND, 2002), (RABIN, 2003), (CHAMPANDARD, 2004) e (WOOD, 2004). Estes trabalhos buscaram não só desenvolver novas técnicas, mas também aplicar e adaptar técnicas conhecidas ao domínio dos jogos.

Uma de nossas contribuições se insere neste último caso: aplicamos uma técnica de sucesso no domínio de futebol de robôs ao domínio dos jogos.

Esta contribuição pode ser dividida em três partes:

1) Esboçamos uma metodologia de projeto de agentes baseada em redes de comportamentos e sensores nebulosos (Cap. 3 deste trabalho).

2) Demonstramos a aplicabilidade e a adequação das redes de comportamentos estendidas aos jogos de computador. Verificamos a qualidade da seleção de açõe através da observação do agente ao longo de vários jogos e da análise dos registros das ações selecioandas. Sua performance foi analisada medindo seu placar e o placar de outros dois agentes. Um agente era totalmente diferente e baseado em máquinas de estado finito. O outro tinha comportamentos e sensores idênticos, mas era plenamente reativo. Ver (PINTO e ALVARES, 2005-a) e os capítulos 4 e 5 da dissertação para maiores detalhes.

3) Delineamos uma metodologia do projeto de estereótipos e a ilustramos com cinco casos (PINTO e ALVARES, 2005-b). Comparamos com outras abordagens e delimitamos sua aplicabilidade. Concluímos que é no projeto de personalidades simples para agentes complexos que as redes de comportamento se destacam.

As contribuições teóricas são mais modestas, mas importantes. Os resultados dos experimentos para averiguação da aplicabilidade contribuem para validar as redes de comportamento como um mecanismo de seleção de ações adequado para agentes situados em ambientes complexos, contínuos e dinâmicos em geral.

Como contribuições secundárias cabe citar a comparação com outras abordagens para projeto de personagens, o esboço de redes para permitir que o agente jogue em outros modos de jogo e as sugestões para incorporação de aprendizado à rede de comportamentos estendida.