ICE: An agent-based kernel for games

Felipe R. Meneguzzi[1]
Paulo H. S. Schneider[1]
Thais C. W. Santos[1]
Michael C. Móra[1]

[1]Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul
{felipe,pschneider}@jeeklabs.com, {twebber,michael}@inf.pucrs.br

Abstract
This paper describes the Intelligence Control Engine *(ICE), a lightweight AI engine aimed at the implementation of autonomous agents for computer games. The engine thus described comprises a simple agent language that can be compiled into an object-oriented programming language, as well as a set of libraries to bind the agent into a game. Usage of such engine is described as well as examples of operation.*
Keywords: *BDI Agents, Game AI*

1. Introduction

One of the first commercial games to use notions of artificial intelligence was Pacman [11]. In this game the player controls an avatar (*i.e.* a player's representative within the game world), that must escape four "ghosts" in a labyrinth. One of its most appealing features was that these ghosts did not have the same individual behavior, rather, each one had a distinct plan of how to chase down and attack the player using different path search strategies.

From Pacman to the present, artificial intelligence has evolved and has gained increasing strength and importance within game projects, so far as having dedicated AI programmers within development teams right at the beginning of the game development process. Developers have been allocating an increasingly larger amount of processor time for AI processes, which demonstrates a shift in focus within the gaming industry. These favorable changes with regards to AI have been occurring mainly due to an increasing demand for what is commonly known as game "playability". Therefore, many AI techniques had been employed in the game context. One of the most popular are the pathfinding algorithms, since it is vital that the game's mobile elements can navigate through its terrain in a reasonable fashion. The most used algorithm of this sort is the A*, which already has various implementations and optimizations within games. Another important notion which has become increasingly popular is A-Life, which seeks to model a life form behavior. A famous game that used such technique was The Sims, which simulates the life of a human being, or a family, through a simulated social interaction between virtual characters, thus, developing custom personalities and behaviors. Agent techniques are commonly used bearing a few restrictions, nevertheless few games are known to have been developed using agent techniques without relying on "ad-hoc" modifications.

This work describes a game behavior kernel based on agents called ICE (Intelligence Control Engine). This paper is organized in the following sections: Section 2 presents related works, Section 3 shows the proposed architecture, Section 4 presents the ICE language, Section 5 describes a few case studies and Section 6 shows the concluding remarks.

2. Related Work

The development of generic AI libraries and engines has been recently described as being the next big "thing" in game development [8] as AI is an important aspect of any commercial game. Efforts on the conception and implementation of such engines have been made both at the academic level [10,1,2,3] and aiming at the gaming industry [9]. While research on the academic side clearly aimed at theoretical soundness while neglecting agent auton-

omy, the work targeted for computer game AI has focused on momentarily responses and algorithm performance and adaptability.

Among the academic agent architectures the most widely known are the descendants of the PRS agent systems, the latest implementation of which is called AgentSpeak [10, 1]. These architectures focus on procedural reasoning to allow implemented systems to react in a timely, though not very flexible manner. An interesting implementation approach is the SOAR architecture [5], which using its own theoretical model, allows the modeling of agent systems using a variety of cognitive notions like emotional states.

Regarding agents in game programming, the Excalibur project [9] is an ongoing effort into the heuristic implementation of autonomy into game AI, as well as the standardization of AI algorithms in a way similar to what currently exists regarding graphics. An example of commercial AI toolkit initially aimed at game development is SimBionic [4], which provides a graphical interface for the development of Finite State Machine-based behavior, as well as an executing environment to be integrated into a game architecture.

3. Architecture

The ICE environment is essentially a multi-agents system. According to generic definitions, agents perceive its surrounding world through sensors, and act in it through actuators. Therefore, the main and theoretically unique interface required between agents are actions generated by actuators and perceptions received by sensors.

Under a simplified analysis, this interface can be seen as a message passing system, the implementation of which is similar to most simulation systems. Thus, the ICE architecture is composed of a component in charge of this sort of message exchange. This component is called Ether, referencing the mythical substance that revolves the universe, it is used as an environment by all game agents, according to Figure 1, its importance is detailed in Section 3.2.

The action mapping into perceptions is accomplished by the Ether component, based on the ICE configuration for the creation of

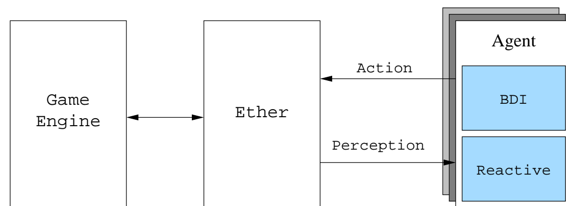

Figure 1: ICE Architecture.

the game environment. This comprises the ICE kernel, since it is through this component that the communication between agents will occur. Embedded in the Ether reside agents that represent game entities, and are composed by two layers: a low level component, that represents an reactive agent model, and a high level component, that represents the BDI agent model.

This game entities, existing thus far only in its logical representation, must be represented, somehow, to the player. Although the agent's cognition and interaction are the main goal of the ICE environment, an interface with the game engine must exist. This interaction is needed in order to represent visually to the user the game entities states, a fundamental pre-requisite for the player interaction with the architecture. In order to allow such communication between the game logic (*i.e.* ICE environment) and the visualization layer, the Ether must provide an interface that allows the game engine to extract the need information to generate the visualization.

3.1. ICE Agents

The agent architecture proposed as the ICE model will consist in a hybrid model, containing reactive agent-based components, which we will call RA, and BDI agents (Belief, Desire and Intention) based components, therefore called BDIA. Each entity representation described with ICE in a game world shall have an RA component in the lowest level, and optionally, a BDIA component.

The reason for this composition of game entities will become clear after an analysis of each component's features. The RA component, giving its purely reactive feature, will make a quick implementation possible, since

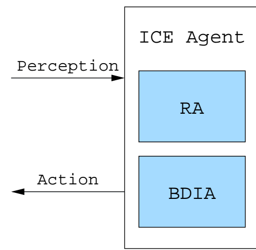

Figure 2: Overview of an ICE agent.

its behavior will be defined in a similar manner than conditional clauses (if, then). Also, it will be used to describe effects whose behavior can easily be described at prior by the developer. Generally, this is a typical behavior for inanimate entities, or even the ones who do not display intelligent behavior. As an example, a chair may break if it perceives it have 20 points of damage, or also a zombie, who walks from right to the left until it finds a living being, and then attacks it.

The BDIA component, based in [13] BDI architecture, has the goal of describing more complex behavior generally displayed by intelligent beings. This component may have a planning capability, thus allowing a higher degree of abstraction by the developer. Obviously, the BDIA component will also display a higher degree of processing time however, since this is an optional component, it may be left to entities of higher importance, such as a soldier. This will allow the predictability of the computational cost; on the same time allow balancing the cognitive capabilities of the game components. Internally, the agents might be equipped with a structure that resolves conflicts in the architecture's hybridism. The communication flow of such structure is displayed in Figure 3.

As soon as the agent receives a perception from the ambient, it will be delivered to the RA component, which will check in its rules if there is a defined reaction for the current perception. When an action is executed by an agent, this action will be delivered to a function that will revise the beliefs.

The BDIA component considers the updated beliefs to decide if the current objective is still viable or it is already fulfilled. If reconsideration is needed, the intention generator will, based on the user-defined objectives, analyze the beliefs and select the viable desires. Having the selected desires, the intentions generator will sort out these desires with some kind of user-defined priority function and will begin the planning to solve them. In the ICE agents, the intentions match the planning steps, in other words, concrete actions. Since the intentions are the planning steps, they can be represented as queue of actions to be executed. Therefore, when the reactive component (RA) generate an action, this will be placed in front of the intention queue, giving a higher priority to this component, compared to the BDIA. This priority system is appropriate to the RA expected semantic, in other words, the ability to quickly react to changes in the ambient. As an example a soldier, which we will call Hans, just planned and decided the needed actions to defend his base when an enemy suddenly hits him. Hans must react by lowering his health attribute, and this change must take place immediately rather than wait for the accomplishment of the whole defense plan.

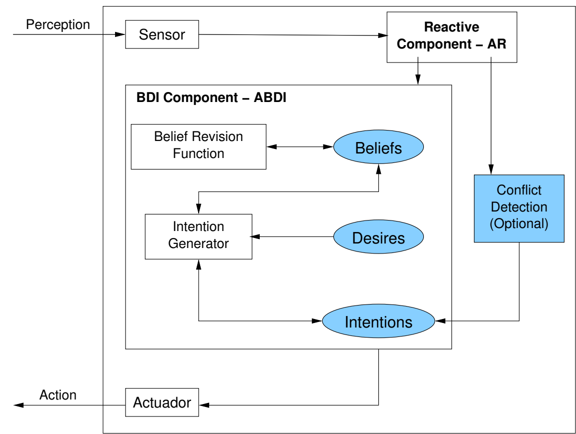

Figure 3: Internal Agent Architecture.

ICE also allows the agent to process its planning in parallel to the execution of previously planned actions. This is possible because the intention generator, which makes the planning, can insert its results in the execution while the agent extracts and executes the actions in front of the queue. The RA component also allows that reactions may be defined for critical situations, where a reconsideration pause and a re-planning would be

unacceptable in terms of waiting time. An aspect that should be considered is the possibility of a conflict between steps of a selected plan and a reaction. To solve this problem, a few hybrid architectures have a conflict detector. The ICE architecture allows the usage of such component, however, in the current implementation, this conflict detector component is not available. Therefore, the removal of possible conflicts is up to the user.

One of the agent's most fundamental features is its independency from the environment, since there is no way to predict the agent behavior through its interface. In a general sense, the ICE agents may or may not have objectives, since the planning and cognition process must be exclusive to entities who have this features. It is important to remember that the process of choosing objectives, selecting commitments and planning its execution is a costly process and should not be executed by most of the agents embedded in the world.

3.2. Ether: the agent environment

The ICE architecture clearly splits the work of the game from the work of the agents. This division was conceived to provide a higher abstraction level to the game developer (who will develop the game engine), since he does not, or should not, have the immediate knowledge in how to manipulate the agents. Therefore, the Ether is such an important layer in the architecture.

Ether is the kernel of the multi-agent system and corresponds to the message passing system between the agents in the ICE architecture. Since the only conceptual agent interfaces are the actions from the actuators and the perceptions received by the sensors, Ether is in charge of receiving the agent's actions, processing them and sending the perceptions back to their agents. According to [12], Ether can be classified as inaccessible, non-deterministic, episodic, dynamic and discreet. Besides "routing" the agent's perceptions, Ether works as an ambient representation. It will map the desired worlds constants for the agents.

Ether also implements the exchange of two message types between the ICE agents: actions and perceptions. Action messages, which are sent by an agent to the Ether, have the goal of informing the multi-agent system which function an agent plans to execute. When the Ether receives this message, it should find out what agents can be affected by this message and, send then a perception message, showing what happened in the environment. The Ether functionality is directly linked to the game engine. As the game program executes its main loop where the critical processing take place, such as showing images, polling input devices and effecting the game logic, a time slice will be designated to Ether for message processing. This will make possible to control the logical processing time. In the same way as the game engine, Ether can change the processing time of the ICE agents, giving them time slices. Therefore, the agent action processing can be balanced in such way as the elements outside the player's line-of-sight, considered "out of bounds", can process less than elements near the player.

4. Agent-Oriented Language

The development of successful games is an experimental process, even based on trial and error at times [7]. In order to ease the process of experimentation it is important that various game elements be parameterized outside program code so that behavior is separated from the code. This separation is achieved in the ICE architecture through a behavior definition language based in hybrid agents that merge definition elements from BDI and reactive agents. Therefore, the language should provide adequate constructs to describe characteristics from both architectures. This language is called IADL (ICE Agent Description Language) and will be described throughout this item using a grammar in which the production rules relative to each characteristic will be presented in the corresponding item. Some grammar rules are not described in this paper, whereas the entire grammar is available in [6].

4.1. Language constructs

4.1.1. The Agent

The syntax defined for an agent description is very similar to that used for a class descrip-

tion in object orientation, bearing in mind the change in paradigm.

In order to supply the agent's BDI characteristics, a set of beliefs relevant to the agent should be defined, beyond that, a set of objectives corresponding to the agent desires should be defined. The agent intentions are not defined in the language, given the fact that these are defined through the desires and beliefs, at run time. Since planning is not currently the focus of this project, the set of plans shall be specified through the language. Regarding the reactive characteristic, it is contemplated through a set of perception-triggered reactions. Common to both agent models used in this project is the definition of a set of actions for possible execution. These characteristics are materialized in the following rule:

```
<agent> -> IDENTIFIER {
               <beliefs>
               <actions>
               <objectives>
               <plans>
               <reactions>
           }
```

4.1.2. Types

Agent specifications using first-order logic are usually complex and excessively slow [12], this last fact especially important in the context of computer games. The group therefore has chosen the definition using simple types, similar to those used in contemporary programming languages such as C++ and Java. Beyond simple types, two complex type possibilities have been added to the language: the construction of composite types, similar to C `structs`, and the construction of type lists.

4.1.3. Actions

A common element to both agent models used in this hybrid architecture is the notion of action, which defines the environment modification ability of an entity. Actions are defined through the "action:" section in the agent definition.

Parameters are specified for each action and modify its effects. Beyond that the effects of each action over the beliefs are specified. Regarding action effects two semantic alternatives were considered:

In the first alternative the agent executes its action and waits for a return from the environment regarding the effects of the same. In this alternative the effects of an action are not described within the agent, letting only the perceptions received from the environment inform its results to the agent. This alternative implies that the action-to-perception mapping function located in the Ether is more complex. This alternative makes the implementation harder, although its results are more interesting.

In the second alternative, which makes the agent behave like a "schizophrenic agent", the agent assumes that the effects of its actions are always true and immediately after the their execution, it alters its beliefs base to reflect such effects. In this alternative the agent will discover whether he failed or not if a perception informs him of this fact by updating its beliefs.

The alternative chosen for ICE agents was that of the schizophrenic agent, because it is the simplest one to implement and allows the Ether component to be simpler and more efficient. Besides that, it allows the agent to work pro-actively, taking into account its actions result expectancy. In the event of failure detection he can choose a new objective.

The constructs provided in the grammar are the following:

```
<actions> -> action:
              <action_declaration_list>

<action_declaration> ->
  IDENTIFIER(<parameter_declaration_list
      >){
        <action_effects>
  }
```

4.1.4. Beliefs

The first BDIA component related structure to be described by the language is the definition of the beliefs. They compose the world model as seen by the agent and are defined in the form of identifiers that use the types defined in Section 4.1.2. The beliefs represent, therefore, what the agent in question knows, or believes about the state of the world around him. Beliefs do not necessarily represent the truth about the facts, as they could be outdated, or even completely different from the reality. This can happen when the world changes and the agent does not perceive the change. Or even in situations where the Ether

was deliberately constructed to introduce fake information.

```
<beliefs> -> belief:
             <belief_declaration_list>

<belief_declaration> ->
    <type> <belief_declarator_list>

<belief_declarator> ->
    IDENTIFIER = <expression> |
    IDENTIFIER
```

4.1.5. Objetives

The proactive behavior of BDIA is defined by its objectives. In IADL, the objectives are defined through a set of values that are to be met by the beliefs. These values are declared by the "pre" construct through an expression where the belief states desired are connected by logical operators. When this expression is evaluated as true the objective in question will have been achieved. The objectives are defined through the following grammar constructs:

```
<objectives> -> objectives:
             <objective_declaration_list>

<objective_declaration> ->
    <objetive_name>(<parameters>)
        pre(<conditions>)
        pos(<conditions>)
```

4.1.6. Plans

Considering the fact that the construction of a complete planner is not the main objective of this project, the planning of each agent will be achieved through a pre-set plan library. Each agent has a set of plans, each of which has an objective it intendeds to achieve, a set of conditions regarding the beliefs that determine the execution viability of the plan, and a priority value that determines the preference order with which the planner will choose the plan to be adopted in case more than one way of achieving an objective is possible. The defined language provides the following constructs for plan definition:

```
<plans> -> <plans><plan_declarator> |
             <plan_declarator> |

<plan_declarator> ->
    plan IDENTIFIER(<objective>)
        if(<conditions>)
        priority INTEGER_CONSTANT {
            <action_call_list>
        }
        pos(<conditions>)
```

4.1.7. Reactions

The RA component functionality is defined by a set of reactions. Each reaction is defined by an identifier for the reaction, the name of the perception that will trigger it and the response action in case the specified perception is received. The reserved word "reconsider" is intended to denote the lack of the necessity to reconsider the objectives if the reaction is activated. This functionality is defined in the language through the following constructs:

```
<reactions> -> <reactions><reaction>; |
            <reaction> |

<reaction> -> <reaction_modifier>
    IDENTIFIER if (IDENTIFIER)
        <action_call>

<reaction_modifier> -> reconsider |
```

4.2. Example of language use

In this section, a simple agent described using IADL is presented to exemplify its usage. The agent describes a simple soldier. He has knowledge of how much ammunition he has, the distance covered, in which direction he is facing and whether he is hiding or not. He makes simple actions, like moving, shooting and taking cover. His purpose is the following: if he has ammunition, he tries to kill the player, if not, he tries to hide.

```
agent Soldier{
    belief:
        int direction=0, ammo=5;
        bool cover = false;
        int distance = 0, lastTurn = -1;
    action:
        shoot(){
            --ammo;
        }

        turn(int dir){
            direction += dir;
            lastTurn = dir;
        }

        move(int size){
            distance += size;
        }

        takeCover(){
            cover = true;
        }

    objective:
        kill()
            pre(ammo>0)
            pos(false)
        hide()
            pre(ammo==0)
```

```
        pos(false)

    plan huntLeft(kill)
        if(lastTurn == 1) priority 1 {
        move(1);
        turn(-1);
    }

    plan huntRight(kill)
        if(lastTurn == (-1)) priority 1
            {
        move(1);
        turn(1);
    }

    plan run(hide)
        if(true) priority 1 {
        move(5);
        takeCover();
    }

    reaction turnLeft if(noiseLeft)
        turn(-1);
    reaction turnRight if(noiseRight)
        turn(1);
    reaction shootFront if(noiseFront)
        shoot();
}
```

5. Case studies

As a mean to validate the ICE architecture and language, a few case studies were proposed. In this section we describe one of them, which is an unimplemented conceptual example, that should provide an illustration of the language used.

5.1. Command base

The command base agent can be considered as an operative center in a strategy game. In this context, the base is in charge of ordering units to gather resources, or defend the base, and it still can heal a damaged unit. Upon notice of the scarcity of resources, the base can order a unit to start gathering it to keep its supply. When the base is under attack, it can request support from nearby units to help fend off the attack. In case a unit is badly damaged, it can order a retreat in order to be healed. In the IADL language, such agent is described as the following:

```
composite Unit {
    bool orderResource;
    bool orderDefend;
    bool orderRecover;
    int health;
};

agent Base {
    belief:
        int resources=100;
        int damage=0;
        bool defend = false;
        Unit unit;
        Unit soldier;
    action:
        orderResource(Unitcollector) {
            collector.orderResource =
                true;
            collector.orderDefend =
                false;
            collector.orderRecover =
                false;
        }

        orderDefend(Unit soldier) {
            soldier.orderResource =
                false;
            soldier.orderDefend = true;
            soldier.orderRecover = false
                ;
        }

        orderRecover(Unit soldier) {
            soldier.orderResource =
                false;
            soldier.orderDefend = false;
            soldier.orderRecover = true;
        }

    objective:
        saveLife()
            pre(defend == true)
            pos(false);
        getResources()
            pre(resources < 20)
            pos(false);
        saveUnit(Unit unit)
            pre(unit.health < 5)
            pos(false);

    plan defense(saveLife)
        if(true) priority 1 {
        orderDefend(soldier);
    }

    plan resources(getResources)
        if(true) priority 1 {
        orderResource(collector);
    }

    plan help(saveUnit)
        if(true) priority 1 {
        orderRecover(unit);
    }
}
```

6. Concluding Remarks

This paper has described a AI engine and its corresponding implementation along with examples of its usage. ICE is by no means the only alternative to generalized behaviorial description available, as was shown by Section 2. Nevertheless it provides a lightweight AI engine implementation coupled with a non-

scripted agent language that can be used in the development of individual game entity behavior decoupling this aspect of content development from the target language. Currently the ICE compiler only generates C++ code, but a Java class generation module is currently being developed and is expected to provide an interesting extension to our work. Compared to existing AI toolkits and architectures, ICE lacks the development tools of SimBionic [4] and the sophisticated motivational processing of SOAR [5], on the other hand, ICE agents provide an efficient implementation of a simple BDI architecture that can be integrated into a gaming project without the overhead of a runtime environment.

References

[1] Bordini, R. H., Fisher, M., Pardavila, C., and Wooldridge, M. Model checking AgentSpeak. In *Proceedings of the 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)* (Melbourne, Australia, July 2003), ACM Press, pp. 409–416.

[2] Braubach, L., Pokahr, A., Lamersdorf, W., and Moldt, D. Goal representation for bdi agent systems. In *Proceedings of the Second International Workshop on Programming Multiagent Systems Languages and tools (PROMAS 2004)* (2004), R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, Eds., pp. 7–9.

[3] d'Inverno, M., and Luck, M. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation 8*, 3 (1998), 233–260.

[4] Fu, D., and Houlette, R. Putting ai in entertainment: An ai authoring tool for simulation and games. *IEEE Intelligent Systems 17*, 4 (2002), 81–84.

[5] Henninger, A. E., Jones, R. M., and Chown, E. Behaviors that emerge from emotion and cognition: implementation and evaluation of a symbolic-connectionist architecture. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (2003), ACM Press, pp. 321–328.

[6] Meneguzzi, F. R., de Souza Schneider, P. H., and Santos, T. C. W. D. Ice: Kernel de comportamento para jogos. Trabalho de Conclusão, Dezembro 2001.

[7] Nareyek, A. Intelligent agents for computer games. In *Computers and Games, Second International Conference, CG 2000, LNCS 2063* (Hamamatsu, Japan, 2000), T. A. Marsland and I. Frank, Eds., Springer, pp. 414–422.

[8] Nareyek, A. Ai in computer games. *ACM Queue 1*, 10 (2001), 58–65.

[9] Nareyek, A. Intelligent agents for computer games. In *Computers and Games*, T. A. Marsland and I. Frank, Eds., vol. 2063 of *LNCS*. Springer Verlag, 2002, pp. 414–422.

[10] Rao, A. S. AgentSpeak(L): BDI agents speak out in a logical computable language. In *7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, R. van Hoe, Ed., vol. 1038 of *Lecture Notes on Computer Science*. Springer Verlag, Eindhoven, Netherlands, 1996, pp. 42–55.

[11] Tozour, P. *AI Game Programming Wisdom*. Charles River Media, Hingham, Massachusetts, 2002, ch. 1, pp. 3–15.

[12] Weiss, G. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, MA, 1999.

[13] Wooldridge, M. *Intelligent Agents*. The MIT Press, 1999, ch. 2.