TI2806 Contextproject - Emergent Architecture Design Draft

Virtual Humans - Group 4 - Port

https://github.com/tygron-virtual-humans

Sven van Hal - 4310055

Tom Harting - 4288319

Nordin van Nes - 4270029

Sven Popping - 4289455

Wouter Posdijk - 4317149

April 28, 2015

Contents

1 Introduction

This is the draft version of our emergent architecture design. We just programming this week. During the first sprints/weeks, our project is divided in two separate sub-projects. These are the GAMYGDALA port and the Goal plugin. This is why this report has two sections, containing an explanation for both sub-projects.

During the next weeks will keep improving, expending and updating this report so that it becomes more complete. We also think that we will combine this report with the emergent architecture designs of the other groups within our context project.

2 GOAL

Our goal is to implement a plugin in the existing source code. To do this, we need to have an overview of both the relevant parts of the GOAL architecture and the architecture of our own plugin.

2.1 GOAL Architecture overview

The GOAL source consists of two major parts: the grammar and the runtime. These parts can be characterized as the parser and interpreter respectively.

2.1.1 Grammar overview

In this section, the parts of the grammar that are relevant to the implementation of our custom action will be covered.

The grammar uses the antlr4 library to create clear grammar in .g4 files. It then uses this grammar to parse the program. We need to make sure that our custom action (also referred to as âĂŸpluginâĂŹ) is correctly inserted in the existing grammar.

The parsing consists of multiple levels. The first level is converting the plain text to collections of very basic expressions. The second level turns these basic expressions into clearer expressions. In this level, every action has its own class, for example. We need to make sure that such an action exists for our custom action, and that it is created when necessary.

The classes that handle this second level of parsing are called validators, because they not only parse the program but also make sure that everything is correct. The grammar has four different validators for the four parts of the goal language: agents, multi-agent systems, modules and tests. Most relevant for us are the agents; we need to make sure that our custom action passes the validation of the agentâĂŹs execution code, and also that the parameters that are passed to the action are preserved in the process.

The grammar is not yet responsible for any communication with the actual implementation of the plugin; it just makes sure that an action is created, and the parameters, though also not yet interpreted, are passed to it.

2.1.2 Runtime overview

In this section, the parts of the runtime that are relevant to the implementation of our custom action will be covered. In the runtime changes will be made to allow for interpretation of the custom action, and the actual interpretation will be hooked to the implementation of the plugin.

The runtime traverses the rules of the agent program, and when it crosses an action, it creates an action executor for said action. This action executor is then executed. We need to make sure that such and action executor exists for our custom action.

Then, in this action executor, we need to implement the hook with the plugin implementation. We adjust the parameters so that they are in the correct format for the plugin, and then pass it to the plugin, which then handles the parameters and generates output This output must then be passed to the belief base.

2.2 Calculator Plugin overview

This section explains how the implementation of the calculator plugin is set up.

The program consists of many operations (e.g. plus, minus) that all derive from the Operation class. They have a string that represents their operator, and a function calc that gets a list of doubles as an input and produces a single double as the output. Each operation is in charge of setting its own precondition for the amount or the format of the parameters that are passed.

There is one main class, Calculator, that is responsible for finding the right operation and calling it, when being passed an operator string and the aforementioned list of doubles. It uses a class loader to find the correct operation, so that new operations can be added to the program simply by creating a new class that extends Operation in the right package.

3 GAMYGDALA

In the Gamygdala world, Agents have goals they want to achieve. Based on their beliefs, they assess whether or not they have achieved their goals, and for example what's the likelihood to achieve their goals. Based on these properties and based on the relation with other agents, the agent's emotion is calculated.

Each Agent has a separate Goal base which they update every evaluation cycle. Whenever an event happens, the impact on a particular goal has to be provided. New agent-specific values like the goal likelihood, change in likelihood, the overall utility of the goal (how useful is it to achieve the goal) are then recalculated. Finally, a new emotion is distilled from all these particular properties.

Because each goal is unique and interchangeable, the engine itself also keeps track of all goals.

In the process to porting the Gamygdala engine from Javascript to Java, we have implemented a number of design patterns to better structure the code. Since Gamygdala was originally developed as a gaming engine, we had to separate the core logic from other facilitator functions. The core logic is also separated into classes with a single-responsibility and coupling is avoided wherever possible.

During an analysis of the code, we found out that he five main classes of the engine are:

- **Agent** The agent which interacts with the environment.

- **Goal** Goals which Agents want to achieve.

- **Belief** Beliefs which alter the AgentâĂŹs view on the likelihood of achieving the goal.

- **Emotion** The Emotion which an Agent has after processing Beliefs (based on its Goals).

- **Relation** The Relation between two Agents.