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

June 19, 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

We have implemented GAMYGDALA as a plugin in the existing GOAL source code. To do this, we first need to have a good overview of the GOAL architecture. 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 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 plugin) 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 needed to make sure that such an action exists for each of our custom actions, and that it is created when necessary.

All of our custom actions share the same format, that is to say they all take a list of values as their parameters instead of some predicate. Since these actions all share this format, it makes sense to generalise them. This is exactly what we have done. There is an abstract 'parameter action', and to create a new action in the grammar we only need to create a new class for it in a certain package.

The classes that handle the 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 agent's execution code, and also that the parameters that are passed to the action are preserved in the process.

To make sure that this agent validator class did not become much larger than it already was, we created a parameter action factory that handles the creation of the parameter actions. Using a class loader, this factory creates the specific parameter action that matches the operator. This is very easy, because every parameter action executor has an 'Action Token' annotation that tells the class loader for which operator it is used. To allow for easy class loading, the reflections package was used.

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

2.2 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 were made to allow for

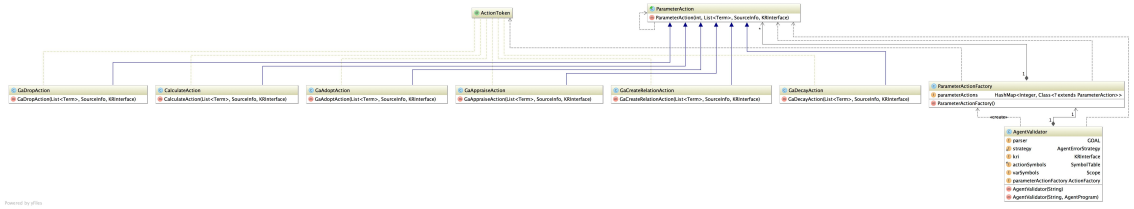

Figure 1: The UML-diagram for the grammar.

interpretation of the custom actions, and the actual interpretation is hooked to the interface between GOAL and GAMYGDALA.

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 needed to make sure that such an action executor exists for each of our custom actions.

Since all actions still share the same format, generalisation could also be applied in the runtime. We created an abstract 'parameter action executor', which all specific action executor implementations extended. Each action executor specifies for which action it is an executor.

All these action executors are put together in the same package. That way, the runtime can easily find the action executor that matches that action is has come across. Again, a factory that uses the reflections package was used, although this time annotations were not necessary. In this case, we could simply use type parameters, since we were now trying to match an action type instead of an operator integer. This way, the code also became cleaner, because by using type parameters a parameter action executor could never be used for the wrong type of parameter action.

Once the action executor is found and actually executes, the call is forwarded towards the GOAL-GAMYGDALA. Since this interface is built so that GOAL does not have to know anything at all about GAMYGDALA except for the actions it provides, the raw GOAL terms are passed to the interface.

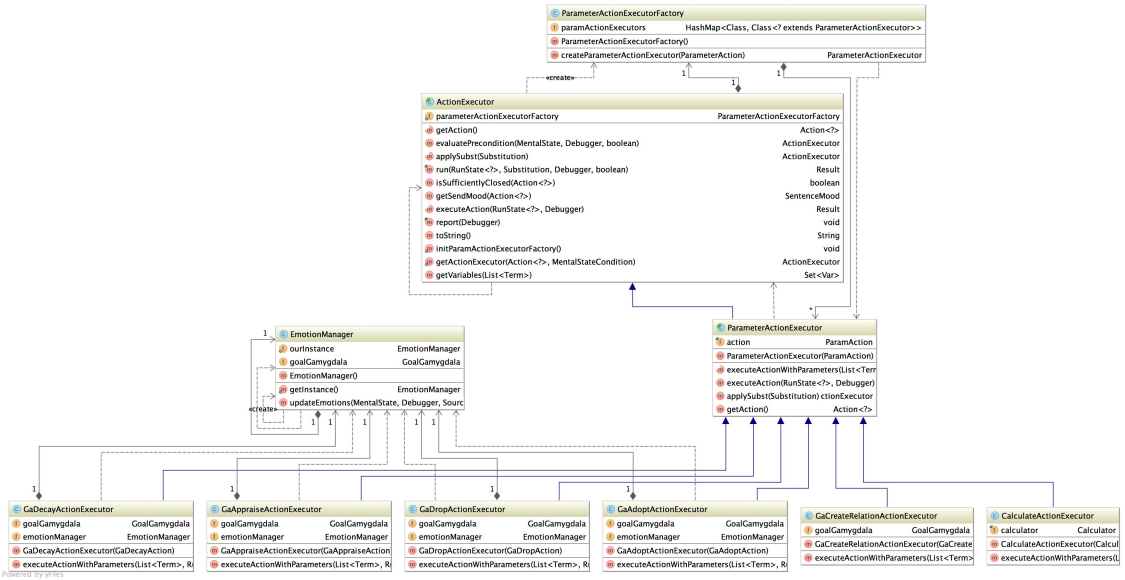

Figure 2: The UML-diagram for the runtime.

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 Agent's 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.

To support our claim that we improved the code on the software engineering part, we made two UML-diagrams. The first diagram is made using our first version of the port, this was an almost exact copy of the original GAMYGDALA code. The second diagram is made using our current code, it is not perfect yet but it is a big improvement over the original code. The UML-diagrams can be found on the next two pages.

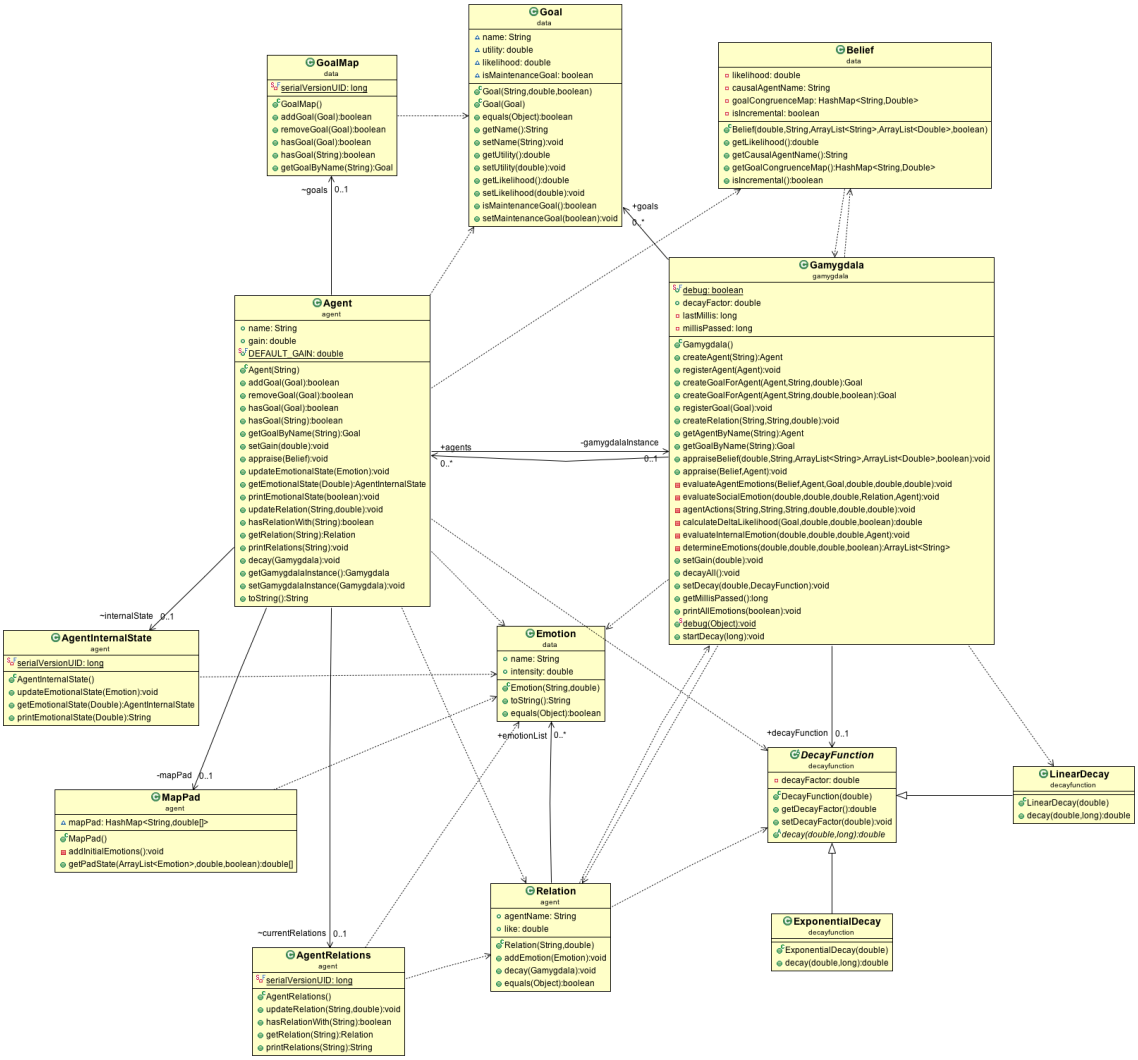

Figure 3: The UML-diagram for the initial port.

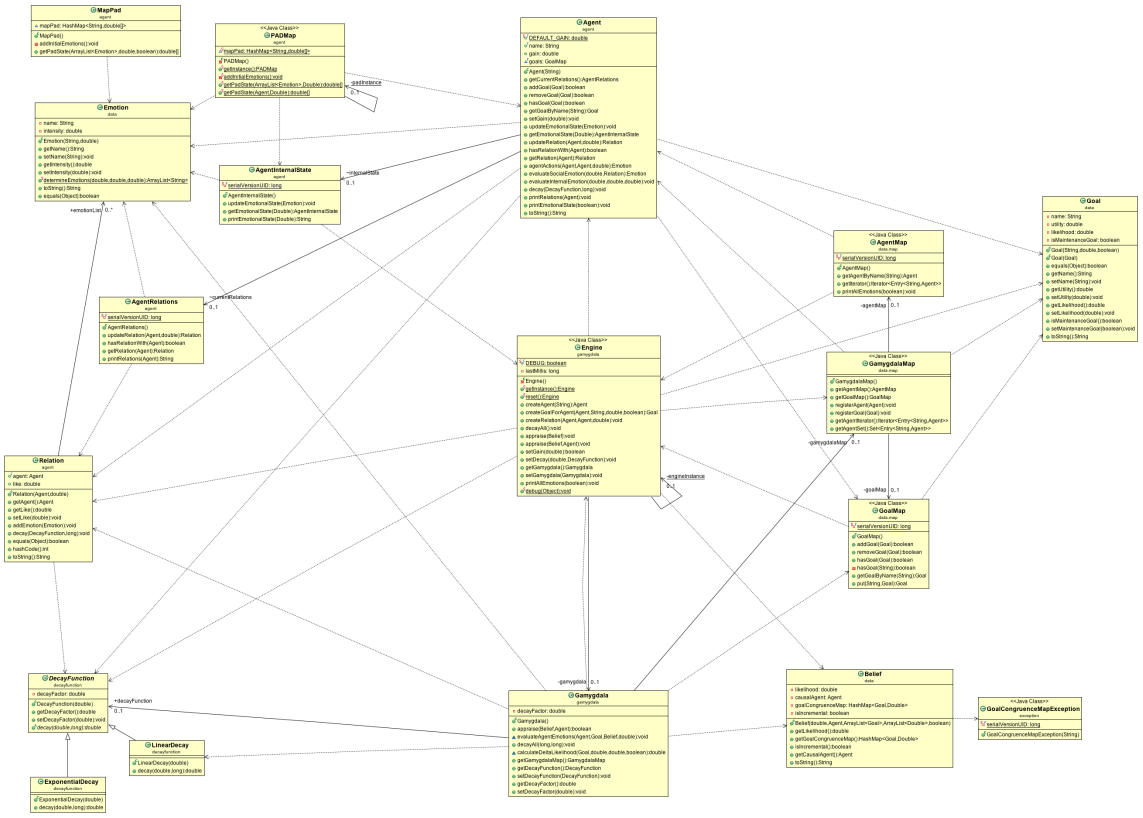

Figure 4: The UML-diagram for the current port.

4 GOAL-GAMYGDALA Interface

Now that you have an understanding of how GOAL works and how GAMYGDALA works, it is possible to understand the architecture of the interface between the two, and the considerations that were made in the process of creating this architecture. There are some significant differences between the way GOAL works and the way GAMYGDALA works. Understanding these differences is fundamental in the process of designing the interface between the two. After all, the goal of the interface is to cross these differences the best it can.

4.1 Term Parser

When a call is made in GOAL, this is done in the form of terms. Terms define the basic structure of a GOAL value, though the only current implementation is in Prolog. Since our GAMYGDALA port is in java, it expects basic java values, so we need to transform these Terms to primary values or lists. This is done by using a term parser. We have used the Strategy pattern for the term parser: we defined a term parser interface that specifies all the types of values that the parser should be able to get from a Term, and we have created the specific prolog term parser implementation. This allows us to easily change the term parser in the case that another language might be supported by GOAL in the future.

4.2 Main interface

Since all calls need to be forwarded to the GAMYGDALA Engine, we have created a singleton called GoalGamygdala that mimics the essential actions of the Engine, though the names might be changed to better match GOAL conventions (since the interface is GOAL-GAMYGDALA and not GAMYGDALA-GOAL). These actions then make use of the term parser to parse the terms into values that GAMYGDALA can understand, and then they forward the calls to GAMYGDALA with the parsed values.

4.3 Agent interface

There are some actions that are agent-specific. Since it seemed rather faulty to lay the responsibility of forwarding these agent-specific actions with the main interface class, which is mainly responsible for forwarding agent calls, it made more sense to create an interface for the agents specifically. The only responsibility that is laid with the main interface is then allowing GOAL to get to the desired agent interface that matches the agent it wants to work with.

4.4 Relation management

Relations are the one specific case where differences between GOAL and GAMYGDALA really show themselves. The main problem is that all GOAL agents work on their own, and they might not be active at the same time. However, agents should be able to define relations to each other regardless of whether they are active or not. To achieve this, relation creation calls are not simply forwarded to GAMYGDALA. First, a check is done to see if the relation target is actually registered in GAMYGDALA. If it is not, the

relation is saved and will be applied the second the target gets registered. Because of this, agents don't have to worry about loading times or sheer inactivity of other agents, and they can simply define relations whenever they want. Once these relations become applicable, they will be applied.

4.5 Agent management

Because both the relation manager and the main interface need to be able to access the agents, to create the relations and do other things respectively, we have created an agent manager singleton that keeps track of all agents. In that way, both the relation manager and the main interface are capable of accessing the agents when they so desire.

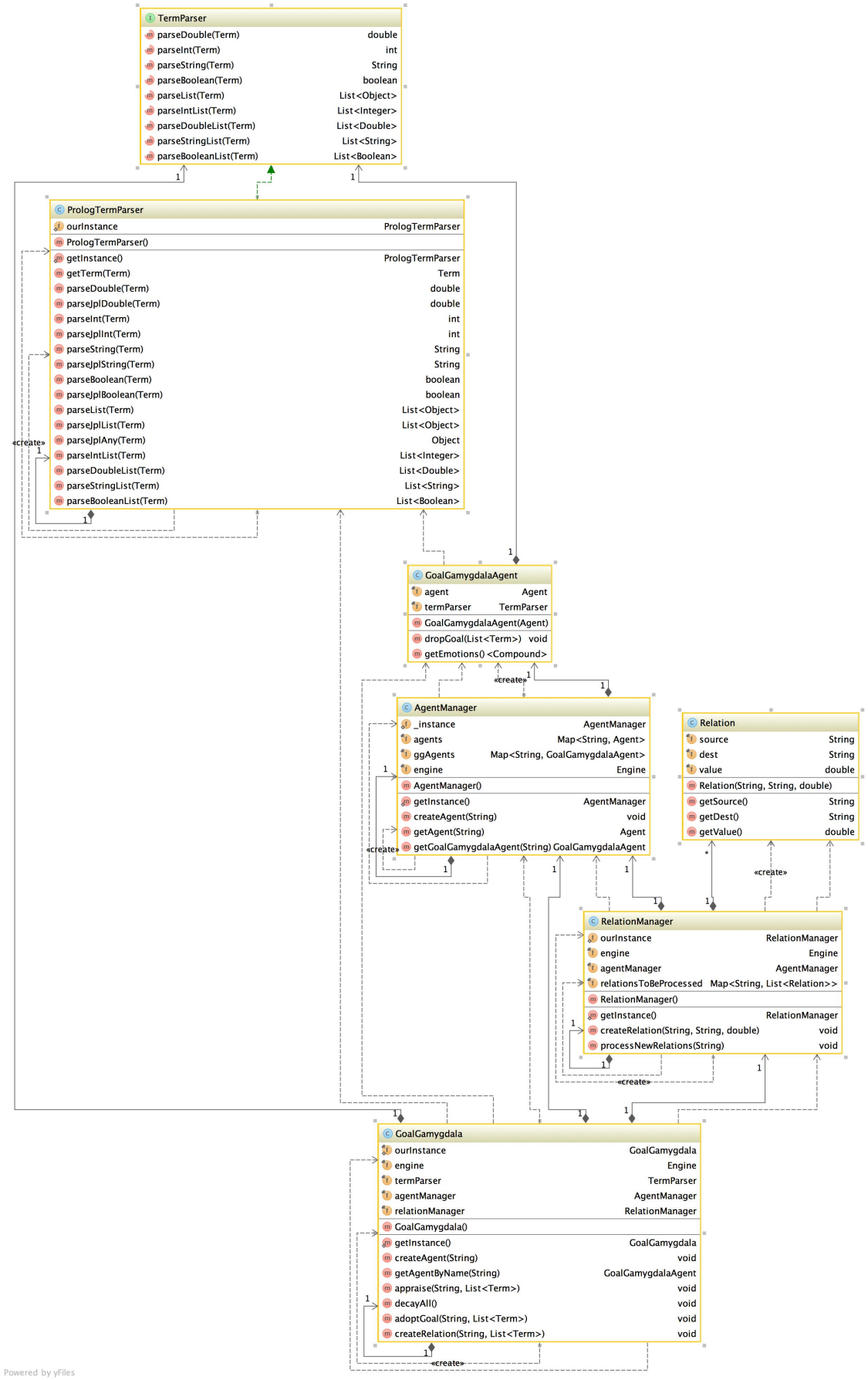

Figure 5: The UML-diagram for the GOAL-GAMYGDALA interface.