DREAM: Deferred Restructuring of Experience in Autonomous Machines
H2020-FETPROACT-2014

Deliverable D6.2

CAFER Release (M12)

Due date of deliverable: December, 31st, 2015
Actual submission date: December, 18th, 2015
Partner responsible: UPMC
Name of participants:

- Stéphane Doncieux (UPMC, main author)
- Carlos Maestre (UPMC)
- Léni Le Goff (UPMC)
- José Antonio Becerra Permuy (UDC)
- Luis Calvo Varela (UDC)

Grant Agreement Number: 640891
Contract Start Date: January, 1st, 2015
Duration: 48 months
Project coordinator: UPMC
Partners: UDC, QML, ARMINES, VU/VUmc

Document validated by the Integration Committee on December, 18nd, 2015.

CONTENTS

PREAMBLE

This document presents the first release of the source code of the CAFER framework. It also introduces its main motivations and requirements and the functionalities introduced in this release. An organisation has been created on GitHub to host all the public code to be released during the DREAM project, see https://github.com/robotsthatdream/. The CAFER framework is in a repository of this organization: https://github.com/robotsthatdream/cafer. The first release has been created as a `v0.1` tag. It can be accessed here: https://github.com/robotsthatdream/cafer/releases/tag/v0.1.

This document has been validated by the integration committee that includes the members of the project listed on the front page of this document. This committee will be updated in the following months to include at least one member per partner.

1 INTRODUCTION

The CAFER framework (Cognitive Architecture Framework based on Evolutionary Robotics) is intended to help implementing the cognitive architecture to be developed in the DREAM project.

1.1 REQUIREMENTS

Validation experiments have been proposed to demonstrate the adaptation abilities developed during the DREAM project (see deliverable D6.1). These experiments require to integrate the source code from the different project members in a single cognitive architecture. This cognitive architecture is not fully defined yet. It will draw inspiration from MDB, the cognitive architecture developed at UDC [1], and will take into account the developments made by the other members.

Given that the requirements of this cognitive architecture are not fully identified yet, we have decided to use an agile software development method. The architecture will be defined in a bottom-up approach in which we start from simple experiments and progressively add the functionalities required to go one step further in the development process. This approach requires a lightweight framework that can easily follow the modifications of the cognitive architecture requirements. It should make its implementation easier while not constraining it.

We have then decided to implement CAFER as a minimal set of services and functions to help implementing the different cognitive architectures that will be proposed during the project. The minimality of the framework is important to ensure an easy adaptation.

It is important for this framework to rely on an existing middleware for three reasons: to make it lighter by minimizing the code to develop, to allow an integration with existing modules and to facilitate its adoption by the robotics community.

1.2 IMPLEMENTATION CHOICE

We have decided to implement CAFER on top of ROS (`http://wiki.ros.org`), a standard middleware in robotics. ROS has a modular structure

that allows to run asynchronous nodes and let them exchange information through topics. Nodes can also offer services to be called by other nodes.

This modular structure of ROS with nodes running in parallel and asynchronously make it a tool of choice to implement cognitive architectures. Furthermore, many nodes already exist for classical robotics functionalities like map-based navigation or robot arm control.

2 METHODOLOGY

2.1 REQUIREMENT DEFINITIONS AND MODIFICATIONS

Several different experiments have already been implemented by the different partners in the project. These experiments have either been implemented in the partner's own software framework, or they may also have been implemented as a separate set of software programs that are manually integrated by the experimenter.

The methodology we have proposed consists in starting from these concrete experiments and start implementing the nodes, services and topics required to integrate them and launch them in a common software framework.

We have started with some of the experiments of two partners: UPMC and UDC, and we will progressively add other experiments from different partners during the course of the project, for the next releases of CAFER.

The goal is to make the learning and restructuring algorithms callable from a central cognitive module.

The CAFER core will be updated if necessary each time a new set of experiments is integrated. First integrations are expected to lead to the most significant updates and later integrations are expected to be done with only minor modifications of the framework, if any.

2.2 SOFTWARE DEVELOPMENT

The source code is documented. Tests are provided to ensure its quality and examples are also provided to help users understand how it works and how to use it. GitHub facilities are used to propose enhancement, identify bugs and follow their resolution.

3 FIRST FEATURES IMPLEMENTED

CAFER is organized around the notion of a `component`, which is an interface that allows to connect to CAFER. As for now, it offers a single service, called `getid`. Last, the template of a memory manager has been implemented.

In the following, we provide a high level description of these features and of their motivations. For a more detailed description, please consult the documentation of CAFER, its source code and the provided examples.

3.1 COMPONENT

The component provides several basic functions. The connection of a source code to CAFER is expected to be done through this element. A particular node can be connected to CAFER through several different components. A component can be seen as a facet of a node. The idea is to create a separate component for each connection to a node or set of nodes. For instance, with a learning algorithm that tests in parallel several controllers, there may be a component for communicating with each controller in the node implementing the learning module. Each of those controllers would connect to the corresponding simulator (or robot), motivation engine, etc. The corresponding component in the learning module would give an easy access to each of those modules, independently from the others.

The component is provided as a template parameterized by an experiment-specific component type. This mechanism is intended to allow future modifications of the component while minimizing their impact on specific code.

A component has a unique identifier whose unicity is guaranteed by the `getid` service.

In the current release, it has the following features:

- watchdog and related services
- change frequency message
- node termination messages
- launch file call

The first three features rely on a `management` topic that connects components between them and allow them to communicate. The management

topic is not expected to be unique in CAFER. It is expected to be shared only between components that are expected to be in direct contact (for instance a controller with the simulation to which it is connected).

3.1.1 Watchdog and related services

Each component sends a message on the management topic at a given frequency. These messages allow other components connected to the same management topic to know what components are up now and when their last message was sent. This is intended to help detecting when a node is initialized and when a node has crashed.

A map is updated each time a message is received. It allows to implement several simple functions, like finding what component is up now, checking if a component is still up or not, waiting for a given component to be up and running, etc.

3.1.2 Change frequency message

The component includes a `ros::rate` attribute (actually a shared pointer on `aros::rate`). A message allows to change its frequency.

Future methods could allow to dynamically update the frequency depending on the current component load and to propagate this frequency adaptation to all connected nodes.

3.1.3 Node termination messages

A message allows to ask a component to stop. Two alternatives have been proposed: a local kill message and a global one. The local kill message only sets a `terminate` parameter to true and relies on implementation specific code to take it into account. The idea is it use it to shutdown only the component (that is the motivation for the "local" term). The global kill message calls a `ros::shutdown` and thus terminates the whole node (that may contain several components).

3.1.4 Launch file call

In ROS, a launch file is an XML file that contains the description of a set of nodes and allows to launch all of them at the same time. This is the most convenient way for launching nodes. A component has a method to call a launch file and thus create new nodes.

3.2 GETID SERVICE

The getid service proposes a service that provides a unique identifier for a given string. This service is used each time such a unique identifier is required.

3.3 DATA MANAGER

The data manager is intended to serve as a basis for any episodic memory like module. It listens to a given topic and stores the messages thus received in a map that can be consulted through dedicated services. It is intended to serve as a basis for all modules responsible for storing data to be used by a redescription process. It has been implemented as a template parameterized by the type of message to store. The template provides the basic functionalities expected from this kind of module, i.e. storing messages in the local database and accessing stored messages through either a random accessor or through a simple search engine.

4 FUTURE WORK

The features described in the previous chapter have been implemented and tested. Examples have also been provided. They will be used in the following months to implement currently available experiments and to implement new ones. These first tests of CAFER will be made in between UDC and UPMC, who are the main developers of CAFER right now. Once CAFER is stable enough, it will be open to other members of the consortium so that they can start to test it and to use it for their own experiments.

BIBLIOGRAPHY

[1] F. Bellas, R.J. Duro, A. Faina, and D. Souto. Multilevel darwinist brain (mdb): Artificial evolution in a cognitive architecture for real robots. *IEEE Transactions on Autonomous Mental Development*, 2(4), 2010. (Cited page 7.)