Toward a Robust Software Architecture for Generally Intelligent Humanoid Robotics

Ben Goertzel
OpenCog Foundation
G/F, 51C Lung Mei Village
Tai Po, N.T., Hong Kong
Email: ben@goertzel.org

David Hanson
Hanson Robotics
25th Floor, Workington Tower
78 Bonham Strand
Sheung Wan, Hong Kong
Email: david@hansonrobotics.com

Gino Yu
School of Design
Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
Email: mcgino@polyu.edu.hk

***Abstract*—This brief "position paper" summarizes the authors' thinking regarding the design of a software framework for interfacing between general-intelligence-oriented software systems and complex mobile robots, including humanoid robots. It is presented from the perspective of soliciting input from the community while the software framework in question is in the phase of design and early implementation, so that this feedback may be appropriately incorporated. An initial case study motivating this work is the use of the OpenCog AGI (Artificial General Intelligence) software framework to help control humanoid robots created by Hanson Robotics.**

I. Introduction

One approach to achieving intelligent behavior in complex mobile robots such as humanoids, is to integrate a general-purpose cognitive architecture with an appropriate conglomeration of robotics-specific software. This is by no means the only possible approach, e.g. purely subsumption-oriented [1] or biomorphic [2] approaches are also interesting and potentially effective. However, the integrative cognitive architecture approach is also worthy of exploration.

In this brief position paper we present some ideas regarding the design and engineering of a software system for connecting cognitive software with robotics software to achieve an integrated cognitive architecture for intelligent robotics. The initial motivation for this system is a current project involving the use of the OpenCog Artificial General Intelligence (AGI) software system [1] [3], [4] to control humanoid robots created by Hanson Robotics. However, the goal is to create a general-purpose architecture, not restricted to this particular application.

II. High Level Requirements

The creation of a requirements specification for a software framework connecting cognitive architectures to robots, is itself a complex pursuit, and it is hoped that discussion of this position paper will facilitate collection of ideas in this regard from the community, which can then be used to formulate a detailed requirements list. What is presented here is merely an outline of high level requirements, intended to evoke the basic intentions underlying the software framework:

- Perception
  - Intake of perceptual data from a variety of different robotic sensors, each with their own drivers and preprocessors
  - Synthesis of data from multiple sensors into an overall perceptual world-view, or (depending on the nature of the data) a collection of overlapping partial world-views
  - Export of processed, synthesized versions of perceptual data to cognitive systems
- Action
  - Intake of high-level or medium-level action plans from cognitive systems, and transformation of these into detailed action plans suitable for enaction by robotic systems
  - Intake of detailed action plans from external systems (e.g. telerobotics controllers)
  - Delivery of detailed action plans to specific robotic systems, for driving robot actions
  - Synthesis of multiple high or medium level action plans, intended for simultaneous or overlapping action, into a single detailed action plan suitable for driving action of a specific robotic system
- Control / Coordination
  - Determination of which actions a robot should take, given its perceptions at a given time and also conditional on the understanding of the current goals and context, where two important cases are:
    - the understanding of goals and context is provided by reference to an external cognitive system
    - the understanding of goals and context is provided by relatively simple ruled supplied directly within the robot control/coordination framework (hence allowing for rapid response in the case of relatively simple "reflex" behaviors or other behaviors that are desired to be preprogrammed in a specific context)
  - Receipt of robot control signals from remote controls or other user interfaces
    - In some cases, control signals from UIs may be treated as absolute commands; in other cases

[1] http:\\opencog.org

they may be treated as special perceptual signals, to be considered alongside other perceptions in formulating appropriate action plans

III. Reference Tools

Among the numerous tools needed to fulfill the above broad requirements are:

- A cognitive architecture, capable of ingesting perceptual data, forming a model of the current context of a robot therefrom, and suggesting appropriate actions
- A robot simulator
- A software framework for interacting with robots
- Optionally, software for generating robot movements, e.g. a 3D graphic art program and/or motion capture program

To the extent these various tools are already able to work together, development will be easier. For example, one option would be to make use of the Blender infrastructure, i.e.

- MORSE, integrated with Blender, for robot simulation
- ROS, integrated with MORSE, for robot control
- BGE, Blender Game Engine, for scripting interactions involving simulated robots
- Blender for art asset creation (including perhaps additional plugins like MakeHuman)

For the cognitive architecture, as stated above our current work involves OpenCog but the framework sketched here doesn't depend on OpenCog specifically.

IV. Architecture Sketch

At a very high level, we suggest to fulfill the requirements posited above via creation of the following software components:

- Action Orchestrator (AO)
- Action Creator (AC)
- Perception Synthesizer (PS)
- Robot Controller (RC)
- Action Data Source (ADS), an interface that might wrap up more than one such data source
- Robot Control User Interface (RCUI)

There are plausible deployment scenarios in which the AC, ADS or RCUI might be implemented on a variety of platforms (e.g. Windows, Mac; or for the RCUI, Android or iOS). On the other hand, we can assume for the foreseeable future that the RC, PS and AO will be run on Linux. Hence one deployment possibility is to deploy the RC, PS and AO as ROS nodes; but implement communication with the other components via some more platform-independent method like ZeroMQ.

A. Action Orchestrator

The AO receives signals from the RC, indicating what set of high level actions to carry out at a given point in time. It then directs the robot how to carry out the actions. In general the signals received by the AO would be lists of action descriptions. Each action description would contain an action name and optionally some (discrete or quantitative) action parameters. The actions usable by the AO should be specified in a configuration file, not hard coded, so it is easy to add new actions or change the parameter-sets of existing ones.

The AO could be connected to a physical robot, or else to a simulated robot in the chosen simulation platform. This would be determined in a configuration file. Ideally the commands sent from the AO to the physical or simulated robot would be identical.

In an initial implementation of the AO, each action type would simply come with a hard-coded mapping into a set of detailed motor commands for a particular robot. A slightly more advanced implementation would handle blending, i.e. would have a way of dealing with multiple simultaneous action requests by blending the corresponding lists of motor commands.

Ultimately, a sophisticated AO would contain an internal hierarchical representation of actions, representing each action as a hierarchically structured set of trajectories of body parts. This would allow flexible blending of actions that started at different times and involved overlapping parts of the body. But this represents a significant AI undertaking, specification of which goes beyond the scope of this document. (Some early thoughts in this regard are given in OpenCog documentation, e.g. in Engineering General Intelligence.)

B. Action Creator

The AC translates movement scripts created externally, into robot action sequences to be enacted in a robot simulator or physical robot.

As one example, the AC would translate animations created in an art program (e.g. Blender, Maya, etc.) into robot action sequences. If the bone structure of the animated character and the robot are identical or very similar, this is relatively straightforward (though still not trivial as it requires knowledge of the parameters of the robot motors, etc.). If the bone structures are different this requires some fancier mathematical translation. The action sequences created in the AC are then exported to the AO, which can invoke them as needed.

C. Perception Synthesizer

The PS receives data from the robot, e.g. sound from a microphone, 3D visual data from a Kinect, visual data from a webcam, etc. It then processes this data into an appropriate form for further utilization. For example, the PS might overlay webcam visual data with Kinect visual data. Or the PS might be configured to pass along to the RC only those parts of the visual data stream that have changed since the last message passed along to the RC. Basic signal processing could also be done in the PS. In essence the PS cleans up and filters data for passing along to the RC (and potentially on to other sources as well, e.g. a data store or OpenCogs Atomspace).

The PS could receive information from the physical robot, or the simulated robot. Ideally the data received in the two cases would be identical in format. (Of course the particulars of e.g. visual data obtained from a simulation would not be the same as from actual visual sensors, though.)

In a future version the AO and PS could work in collaboration, but initially they will be separately operating pieces of software.

Assuming one utilizes ROS as an infrastructure, the AO, PS and RC could be three separate ROS nodes, in frequent communication. The AO and PS would then communicate with other ROS nodes representing particular aspects of the robot or associated software.

D. Action Data Source

An Action Data Source comprises an external piece of software that produces descriptions of actions to be sent to the Action Creator.

One example would be a Telerobotics Data Source, containing a mechanism for observing the movement of a human face, and packaging these movements as a stream to pass to the PS for perceptual understanding; and in a tele-operation scenario, to the AC as well.

E. Robot Control UI

In many cases it will be useful to have a human being directly control a robot, via manipulating a User Interface that sends messages to the RC.

E.g., one possibility is to make such a UI in the form of an Android phone app. This avoids the need to use special hardware, but makes the UI mobile, so that a person can hold the phone in their hand like a remote control and manipulate the robot. It also leaves the opportunity open for advanced manipulation of control parameters as well as simple controls.

F. Robot Controller

The RC contains the logic for choosing actions based on perceptions and based on user controls. Initially, for a standalone application, this logic could be fairly simple. For instance, instances of non user control driven action logic might include:

- Move the head to face someone who is detected to be talking
- Move the head to look at the nearest human face, or at a human face that has recently become mobile, etc.
- Imitate the movements of the person being tracked by the ADS
- Imitate the movements of the person being tracked by the ADS; or if this person is looking at a certain person, then look at that person

The best way to formalize this action logic is not clear. One option is to leverage OpenCog, and:

- Use the OpenCog AtomSpace to contain symbolic abstractions of perceptual data, and action commands
- Use Implication relations in OpenCogs AtomSpace to encode (potentially probabilistically weighted) action logic rules

OpenCog provides a highly general and flexible formalism here, including probability weighting of actions; and also provides a framework in which learning of action logic rules can naturally take place (although the initial action logic rules will be hard coded).

The RC must be able to receive control signals from external sources as well – from humans via the RCUI, or from other software programs. The right messaging protocol must be chosen carefully here; ZeroMQ is one candidate currently under consideration.

V. Toward a Simple, Robust, Extensible Robot Control API

As one possible form of input, we are considering the possibility of implementing the high-level robot control language specified in Jamie Diproses paper Human-Centric API for Programming Socially Interactive Robots [5] as a means of sending high level commands to the RC. Without replicating the description of this API here, we nevertheless will give some simple examples of what the extension of the API we propose might look like. The syntax and semantics are simple enough that the examples should be fairly transparent. Mainly, what is needed for the present purposes is to create classes that represent the various actions the robot performs, and tell the robot to execute them commands like *robot.do(...)* .

E.g. one could say

```
robot = Robot()

greet = SayTo()
smile = GestureAt(Gesture.Smile, person)
wave = GestureAt(Gesture.Wave, person)

greet.add_gesture_start(smile)
greet.add_gesture_start(wave)
greet.add_text("Hello")
greet.add_gesture_end(smile)
greet.add_gesture_end(wave)

robot.do(greet)
```

Or, one could say

```
robot = Robot()
person = Person()

point = GestureAt(Gesture.Point, person)
frown = GestureAt(Gesture.Frown, person)
robot.do(point, frown)
```

It will also sometimes be necessary to say what parts of the body are going to execute a gesture. E.g. whether its the left or right arm that will point. You could do this two ways, via

```
robot = Robot()
person = Person()
point = GestureAt(Gesture.Point, person,
    body_part = robot.left_arm)
robot.do(point)
```

or

```
robot = Robot()
```

```
person = Person()
point = GestureAt(Gesture.Point, person)
robot.left_arm.do(point)
```

This sort of simple syntax, appropriately fleshed out and extended, could be used to enable relatively simple yet functionally sophisticated interfacing with the RC component of the proposed architecture.

VI. Conclusion

A rough sketch of a software architecture for interfacing between cognitive architectures and robotics software systems has been outlined. This represents current work in progress, on which feedback is solicited from others working in the area. Feedback will be incorporated in the ongoing design and engineering work, hopefully resulting in a system with broad utility beyond the initial application of flexibly interfacing OpenCog with Hanson Robotics robots.

Acknowledgment

The authors would like to thank Jamie Diprose, David Hanson, Lake Watkins and Alex van der Peet for discussions that were integral in formulating the ideas presented here.

References

[1] R. A. Brooks, *Flesh and Machines, Pantheon Books, New York*. NY, 2002.

[2] B. Hasslacher and M. W. Tilden, "Living machines," *Robotics and Autonomous Systems*, p. 143169, 1995.

[3] B. Goertzel, C. Pennachin, and N. Geisweiller, *Engineering General Intelligence, Part 1: A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*. Springer: Atlantis Thinking Machines, 2013.

[4] ——, *Engineering General Intelligence, Part 2: The CogPrime Architecture for Integrative, Embodied AGI*. Springer: Atlantis Thinking Machines, 2013.

[5] J. Diprose, B. Plimmer, B. MacDonald, and J. Hosking, "A human-centric api for programming socially interactive robots," *University of Auckland technical report*, 2014.