SUPERAGI
AGENTSPEAK TOOLKIT

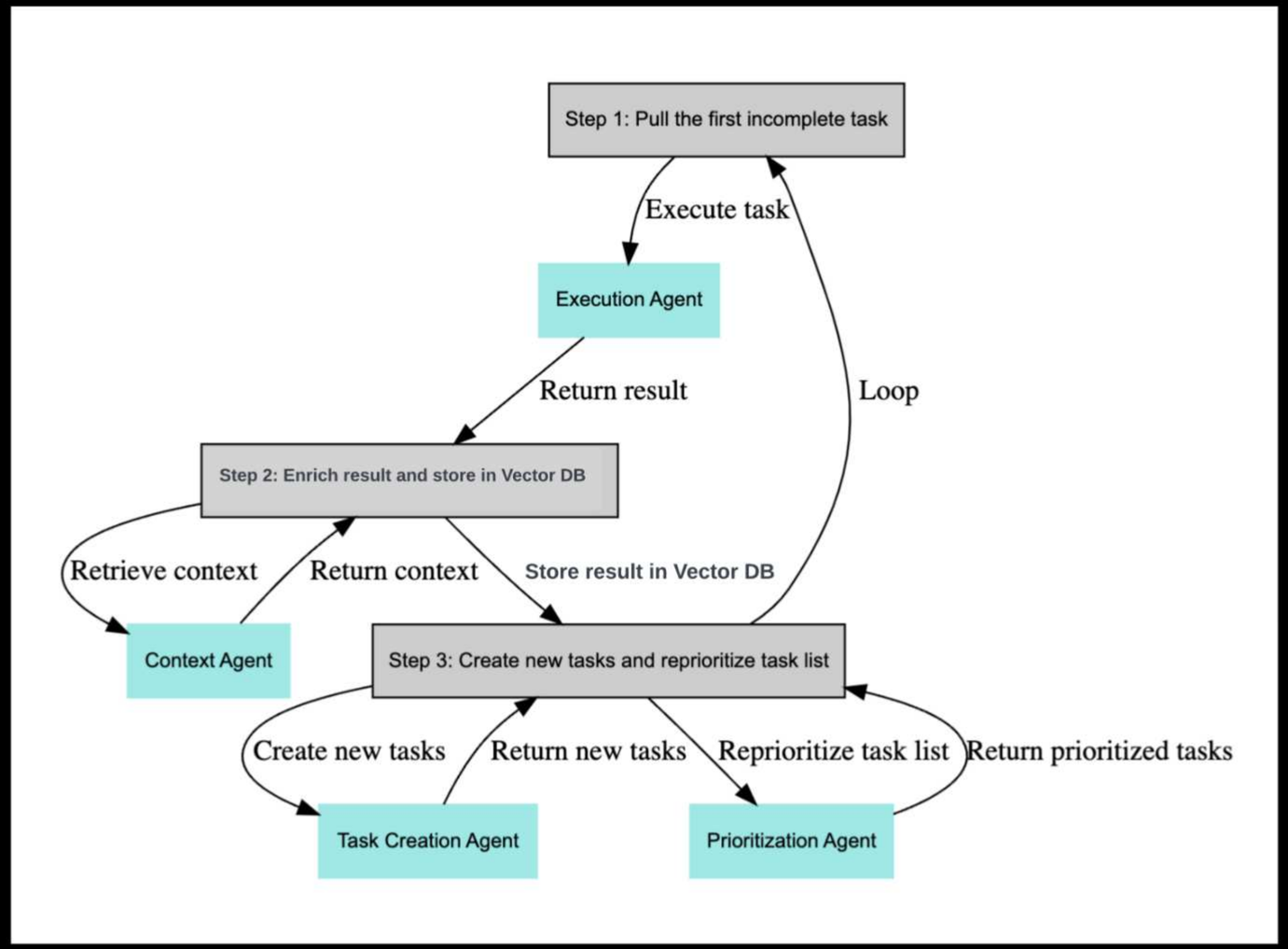
Before the advent of LLMs and LLM-based agents, there was much prior work on autonomous agency.

We felt it important not to neglect, but to instead incorporate, the preexisting research into autonomy.

Some features that I observed were lacking (when first hearing about and then tinkering with LLM-based autonomous agents):

- Sophisticated Control Flow / Event Loops
- Task Convergence
- Repeatability / Reproduciblity
- Verification / Formal Methods
- Sensing or Acting
- Declarative Programming

Example: Early Agent Control Flow:



We need something that can provide:

- Adherence: pin down the agent to keep it on task.
- Completeness: ensure that it doesn't drop the ball on any tasks.
- Versatility: allow developers more control over the agents.
  - for use cases like our DeployGPT system.
- Explainability: allow a formal representation.

AgentSpeak is a great language for reasoning with beliefs, desires and intentions.

It is great at adding and removing goals at run time, which a lot of planning technologies lack (like PDDL).

https://jason.sourceforge.net/wp

Jason
by Gustave Moreau (1865)

Oil on canvas, 204 x 115.5 cm.
Musée d'Orsay, Paris.
© Photo RMN. Photograph by: Hervé Lewandowski

Jason

a Java-based interpreter for an extended version of AgentSpeak

Home | Description | Documents | Examples | Demos | Teaching | Projects | Search

About *Jason*

Jason is an interpreter for an extended version of AgentSpeak. It implements the operational semantics of that language, and provides a platform for the development of multi-agent systems, with many user-customisable features. ***Jason*** is available Open Source, and is distributed under GNU LGPL. See more in the Description page.

Links

- Jason on Github (latest code);
- Jason on Sourceforge;
- Screenshots.

Authors

Jason is developed by Jomi F. Hübner and Rafael H. Bordini, based on previous work done with many colleagues, in particular Michael Fisher, Joyce Martins, Álvaro Moreira, Renata Vieira, Willem Visser, Mike Wooldridge, but also many others, as acknowledged in the manual (see the Documents page).

Download the latest version of Jason!

DOWNLOAD

Read the tutorial for installing Jason as an Eclipse plug-in!

ECLIPSE PLUG-IN

News

11/11/2019

Jason agents can also publish/subscribe to ROS topics using ROS Bridge. Check it out here.

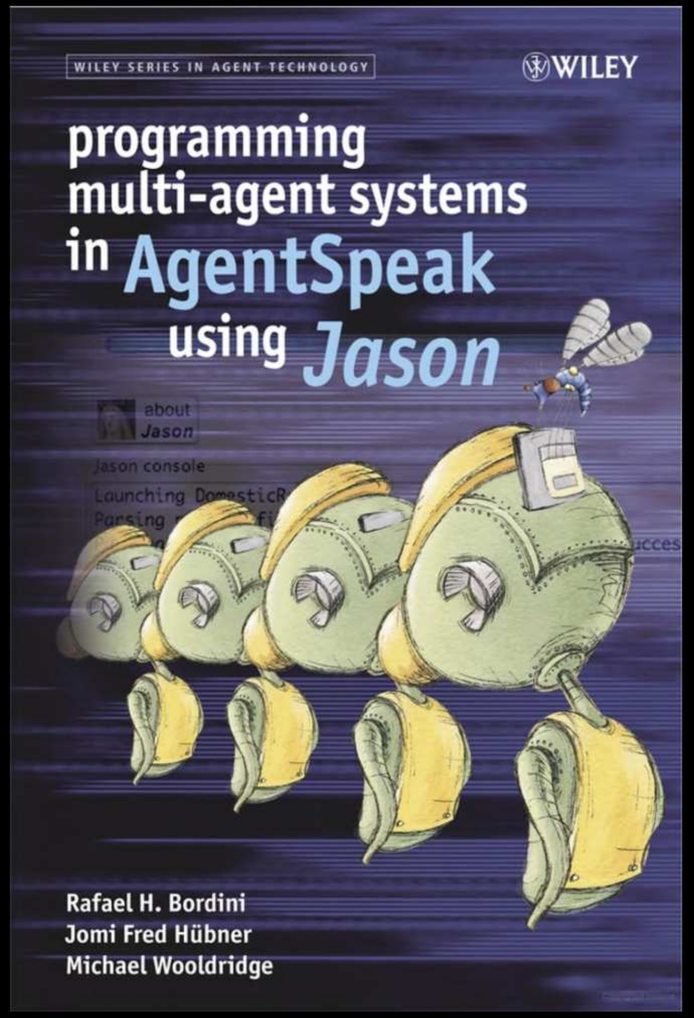
17/10/2019

The Multi-Agent Programming Contest 2019 had two teams using ***Jason***:

- **1st place:** LFC (using JaCaMo)
- **4th place:** TRG

Watch replays of the matches and

“Jason is an interpreter for an extended version of AgentSpeak. It implements the operational semantics of that language, and provides a platform for the development of multi-agent systems, with many user-customizable features.”


WILEY SERIES IN AGENT TECHNOLOGY
WILEY
programming multi-agent systems in AgentSpeak using Jason
about Jason
Jason console
Rafael H. Bordini
Jomi Fred Hübner
Michael Wooldridge

From: https://jason.sourceforge.net/wp/description
One of the best known approaches to the development of cognitive agents is the BDI (Beliefs-Desires-Intentions) architecture. In the area of agent-oriented programming languages in particular,

AgentSpeak has been one of the most influential abstract languages based on the BDI architecture. The type of agents programmed with AgentSpeak are sometimes referred to as reactive planning systems.

To the best of our knowledge, Jason is the first fully-fledged interpreter for a much improved version of AgentSpeak, including also speech-act based inter-agent communication.

Using Saci (for example), a Jason multi-agent system can be distributed over a network effortlessly. Various ad hoc implementations of BDI systems exist,

but one important characteristic of AgentSpeak is its theoretical foundation: it is an implementation of the operational semantics, formally given to the AgentSpeak language and most of the extensions available in Jason.

We therefore have begun work providing SuperAGI with a marketplace toolkit that will enable a DSL for control flow and event loops, by exposing an AgentSpeak interface to it.

AgentSpeak therefore can provide a skeleton or backbone, upon which to rig the LLM-based autonomous agent.

This will serve to both constrain and guide the actions of the autonomous agents.

Integrating AgentSpeak w/ SuperAGI can be likened to creating a system with a co-processor.

We envision installing the AgentSpeak marketplace toolkit, which will then hook itself into the event loop / control flow of SuperAGI, and provide developers with extra capabilities.

Both developers and autonomous agents can eventually generate and edit the AgentSpeak programs.

Furthermore, AgentSpeak programs can invoke and control all manner of LLM behavior and API functions.

Here is some sample Jason/AgentSpeak(L) code that was generated by GPT-4:

```
!evacuate(Person).
+!evacuate(Person) : true <-
        .print("Identify the safest evacuation route for ", Person);
        identify_route(Person, Route);!communicate_evacuation_plan(Pers
+!communicate_evacuation_plan(Person, Route) : true <-
        .print("Communicate the evacuation plan to ", Person);
        inform_person(Person, Route).
+!monitor_evacuation(Person) : true <-
        .print("Monitor the evacuation progress of ", Person);
        monitor_progress(Person).
+!provide_resources(Person) : true <-
        .print("Provide necessary resources to ", Person);
        provide_food(Person);
        provide_water(Person);
        provide_shelter(Person).
+!reassess_situation(Person) : true <-
        .print("Reassess the situation and provide further assistance t
        reassess(Person).
```

We made a somewhat unusual design choice: to enable the use of two AgentSpeak implementations simultaneously: python-agentspeak and Jason.

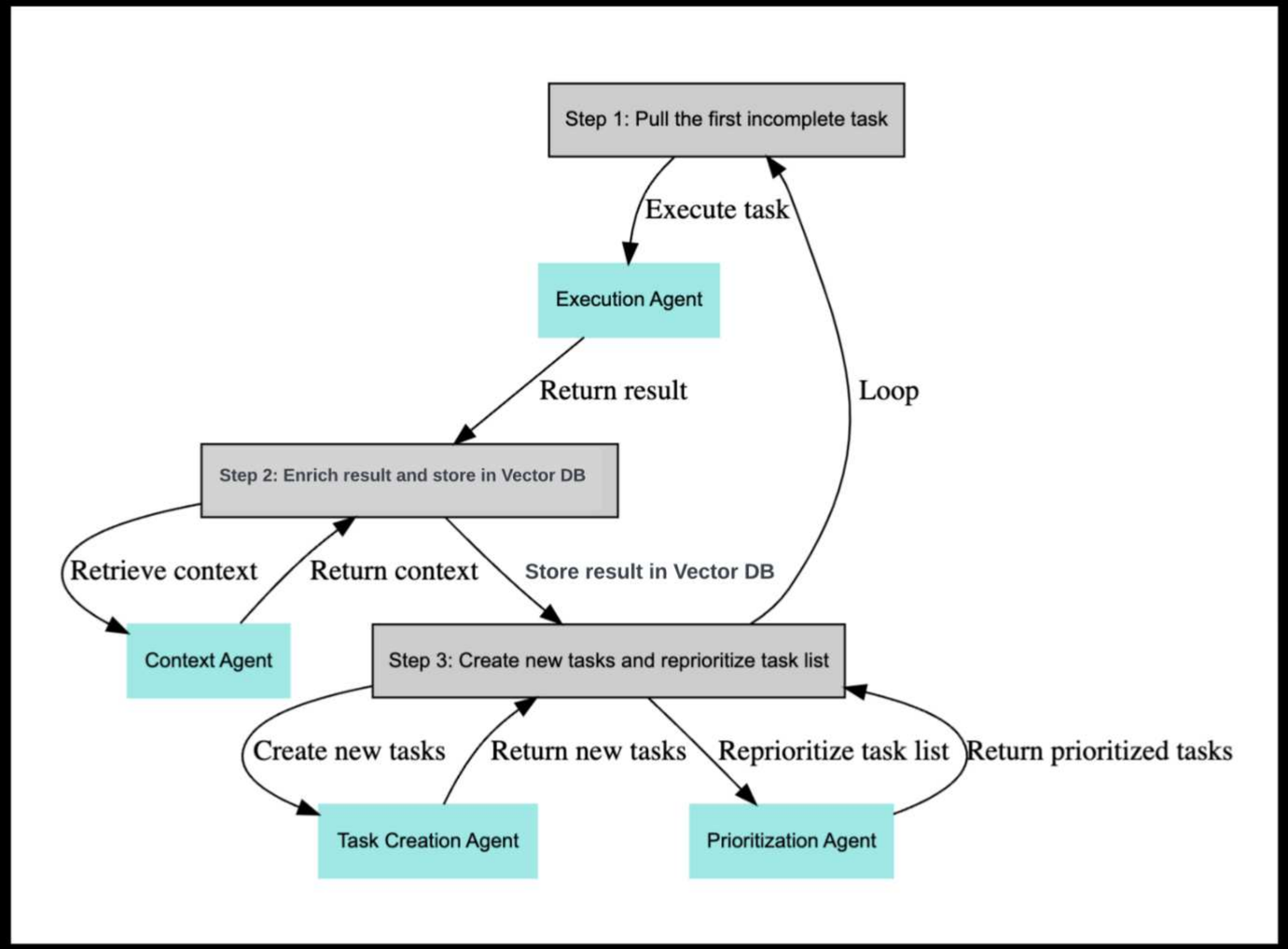
I figured that the python-agentspeak would have library functions necessary to cleanly discuss between Python and Java, and it adds some flexibility.

This might cause some confusion about which contexts are involved, although we intend to address that via naming conventions.

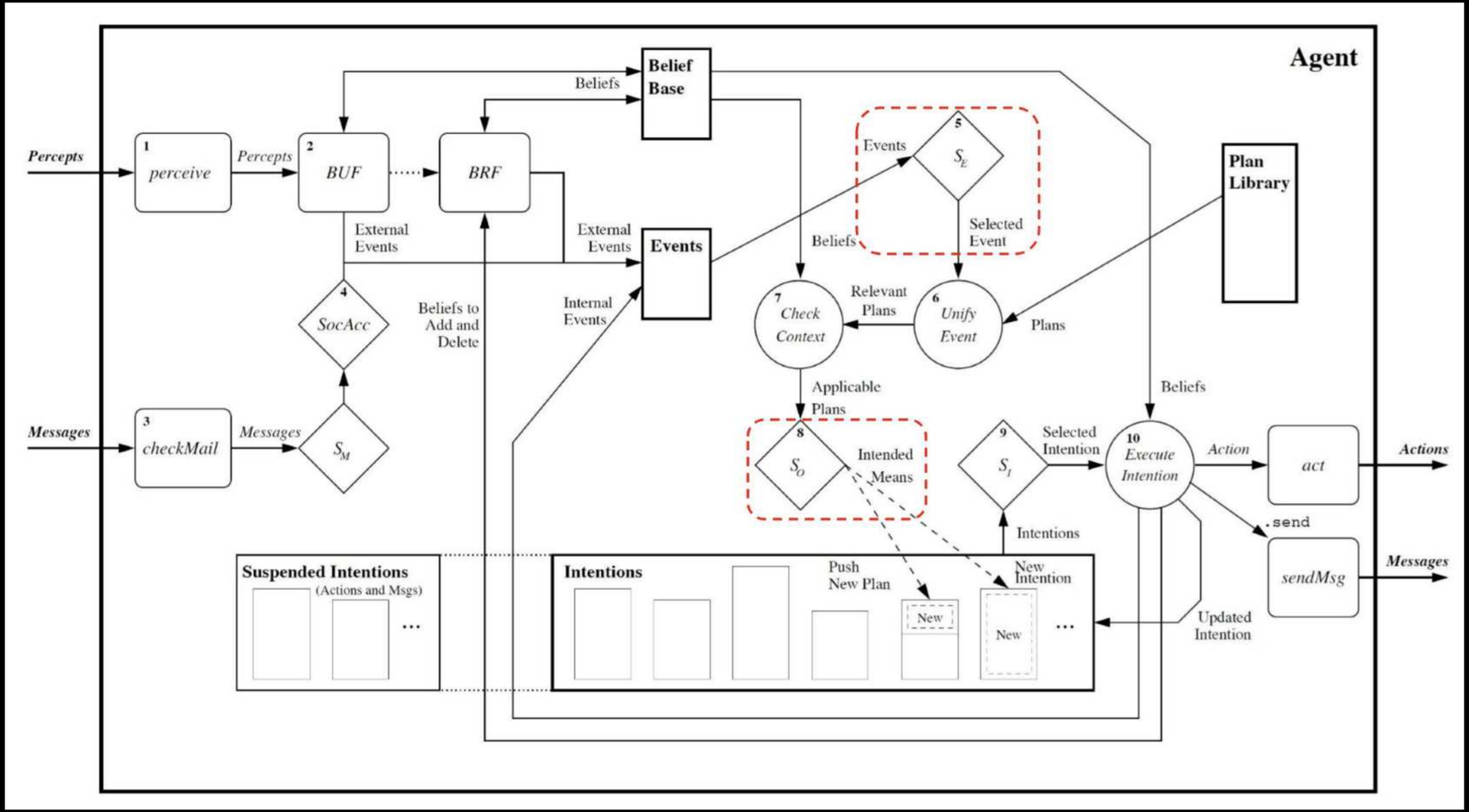
So, our SuperAGI<->Jason bridge uses:

- Jason/AgentSpeak(L)
  - Feature-complete
  - Jason is Programmed in Java
  - Reasons about AgentSpeak(L)
    - Uses a Prolog-like language
- python-agentspeak
  - Not feature-complete
  - Programmed in Python
  - Reasons about AgentSpeak(L)
    - Uses a Prolog-like language
- Py4J
  - Bridges from Python to Java
  - Connects our Python-based plugin with Jason

Recall the Earlier Early Control Flow:



Jason/AgentSpeak(L) Event Loop:



Simplified Jason/AgentSpeak(L) control-loop:

```
1: B ← B 0 ; PlanLib ← PlanLib 0 ; Ev ← {} ; I ← {}
2: loop
3:       ρ ← SENSE_ENV()
4:       BEL_UPDATE(ρ, B, Ev)
5:       if Ev is not empty then
6:          ev ← FETCH_EVENT(Ev)
7:          p ← SELECT_PLAN(ev, B, PlanLib)
8:          if ev is an env change or a new goal to achieve then
9:             I ← I ∪ {NEW_INT(p, ev)}
10:         else if ev is a sub-goal to achieve then
11:            PUSH_PLAN(currInt, p, ev)
12:         end if
13:      end if
14:      if I is not empty then
15:         currInt ← SELECT_INTENTION(I)
16:         a ← FETCH_NEXT_ACTION(currInt)
17:         EXEC_ACTION(a, currInt, B, I, PlanLib)
18:      end if
19: end loop
```

Next we show an example of a Python program running both python-agentspeak and Jason/AgentSpeak(L).

We will eventually insert code like this into the toolkit plugin.

We achieved Python<->Java integration using Py4J.

```
#!/usr/bin/env python
import agentspeak
import agentspeak.runtime
import agentspeak.stdlib
import os
import pprint
from py4j.java_gateway import JavaGateway, java_import
gateway = JavaGateway()
java_import(gateway.jvm, "py4j.examples.JavaAgentSpeakClient")
actions = agentspeak.Actions(agentspeak.stdlib.actions)
.add_function(".call_java_agentspeak", (int, ))
defcall_java_agentspeak(x):
    stack = gateway.entry_point.getStack();
    pprint.pprint("Adding to stack: "+str(x));
    stack.push(str(x));
    return 1;
env = agentspeak.runtime.Environment()
withopen(os.path.join(os.path.dirname(__file__), "agent.asl")) as sourc
    agent = env.build_agent(source, actions)
if__name__ == "__main__":
    env.run_agent(agent)
```

```
!start.
// +!start <-//      .custom_action(3, X);
//       .print('X =', X);
//       .print('I LOVE THIS!').
+!start <-+goal(solveUserProblem(andrewDougherty,problem1),3).
+goal(X,Y) <-
        .call_java_agentspeak(Y,Z);
        .print('I LOVE THIS!: ',X,Z).
// +!call_language_model(Model
```

The next video is hard to follow:

- Shows our successful technology integration experiment
- We separately launch Jason and Python
- The Jason environment has code to startup the Java end of the Py4J bridge
- The Python script loads the python-agentspeak library
- Declares a a python-agentspeak internal action call_java_agentspeak/2
- Proceeds to run the agent specified by agent.asl
- Agent defines a plan to add a belief goal/2
- A trigger for adding a belief which defines a plan to call the call_java_agentspeak/2 with some args.
- The call_java_agentspeak/2 internal action then uses Py4J to obtain the Java stack object
- The call_java_agentspeak/2 proceeds to push its first argument onto the Java stack object


andrewdo@ai2: /home/andrewdo
Must guard, guard, guard, guard, guard...
-UUU:**--F1 *shell*   All L2   (Shell:run WK edna Projectile)

What kind of use-cases are enabled by this new bridge between Python and AgentSpeak?

The Free Life Planner

(DRAFT - Do NOT Distribute!!):

A Virtual Secondary Social Safety Net

Andrew John Dougherty
FRDCSA Project

Abstract

Could free software artificial intelligence be uniquely positioned to help alleviate poverty, hunger, disease, and a slew of other aptly-termed "wicked problems?" There are a number of positive indicators. Free software such as Linux has the benevolent property that it may be copied ad infinitum for essentially zero cost, enabling world-wide distribution. Moreover, modern artificial intelligence software on commodity hardware is capable of solving an increasingly wide range of problems, often with superhuman performance. Additionally, the digital divide is disappearing, with intermediaries able to reach the rest. These factors mean that essentially everyone may soon have access to or benefit from free superhuman intelligence. If so, a virtual secondary social safety net system could be developed (at near-zero development and marginal cost), which could help people to be more self-reliant and less dependent on a primary social safety net. This could help to improve quality of life, reduce suffering, and save lives. We propose the Free Life Planner (FLP), a systematic life-planning system intended to help fulfill this opportunity.

Introduction

Purpose of the Software

To End the "Information Dark Ages". The late Dr. Jaime Carbonell had the praiseworthy goal of "getting the right information to the right people at the right time in the right language in the right medium with the right level of detail."[27] Anywhere this goal remains unsatisfied we consider to belong by definition to what we term the "information dark ages."

The author recently heard of a situation in which a patient with a pacemaker, whose medical records had not been transferred to the appropriate physician, was scheduled to undergo medical treatment involving electrical shocks. This might have proven serious or fatal, had not a confidant serendipitously discovered the issue.

Such situations underscore several facts: that some people still live in the "information dark ages," that some people must self-advocate, that life-and-death matters are best not left to chance, and that systematic or catch-all solutions are needed.

https://freelifeplanner.org/doc

https://altruisticsoftware.org/frdcsa

Through Jason, SuperAGI and the FRDCSA and Free Life Planner systems can pass messages to each other.

This is a massive force multiplier, as it enables both to control each other.

However, we found setting up Py4J, given our lack of experience with either language, to be difficult and time consuming.

This is why we weren't able to get further along with the AgentSpeak Toolkit.

I still need to decouple FLP's Jason<->SWIPL interface from the Python interface, in order to have a public release.

However the code as is can be found here:

https://github.com/aindilis/jason

In particular, see:

- https://github.com/aindilis/jason/tree/master/examples
  - python-adapter
  - python-adapter/scripts
  - python-adapter/src/java

!THE_END.