# Spade-BDI Documentation

***Release 0.3.2***

**Sergio Frayle Pérez**

**Jan 03, 2025**

# CONTENTS:

CHAPTER

# ONE

# SPADE-BDI

Create hybrid agents with a BDI layer for the SPADE MAS Platform.

- Free software: MIT License
- Documentation: https://spade-bdi.readthedocs.io.

## 1.1 Features

- Create agents that parse and execute an ASL file written in AgentSpeak.
- Supports Agentspeak-like BDI behaviours.
- Add custom actions and functions.
- Send TELL, UNTELL and ACHIEVE KQML performatives.

## 1.2 Examples

basic.py:

```
import getpass
from spade_bdi.bdi import BDIAgent

server = input("Please enter the XMPP server address: ")
password = getpass.getpass("Please enter the password: ")

a = BDIAgent("BasicAgent@" + server, password, "basic.asl")
a.start()

a.bdi.set_belief("car", "blue", "big")
a.bdi.print_beliefs()
```

(continues on next page)

(continued from previous page)

```
print(a.bdi.get_belief("car"))
a.bdi.print_beliefs()

a.bdi.remove_belief("car", 'blue', "big")
a.bdi.print_beliefs()

print(a.bdi.get_beliefs())
a.bdi.set_belief("car", 'yellow')
```

basic.asl:

```
!start.

+!start <-
    +car(red);
    .a_function(3,W);
    .print("w =", W);
    literal_function(red,Y);
    .print("Y =", Y);
    .custom_action(8);
    +truck(blue).

+car(Color)
 <- .print("The car is ",Color).
```

## 1.3 Examples

basic.py:

```
import getpass
from spade_bdi.bdi import BDIAgent

server = input("Please enter the XMPP server address: ")
password = getpass.getpass("Please enter the password: ")

a = BDIAgent("BasicAgent@" + server, password, "basic.asl")
a.start()

a.bdi.set_belief("car", "blue", "big")
a.bdi.print_beliefs()

print(a.bdi.get_belief("car"))
a.bdi.print_beliefs()

a.bdi.remove_belief("car", 'blue', "big")
a.bdi.print_beliefs()

print(a.bdi.get_beliefs())
a.bdi.set_belief("car", 'yellow')
```

basic.asl:

```
!start.

+!start <-
    +car(red);
    .a_function(3,W);
    .print("w =", W);
    literal_function(red,Y);
    .print("Y =", Y);
    .custom_action(8);
    +truck(blue).

+car(Color)
 <- .print("The car is ",Color).
```

## 1.4 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

# INSTALLATION

## 2.1 Stable release

To install Spade-BDI, run this command in your terminal:

```
$ pip install spade_bdi
```

This is the preferred method to install Spade-BDI, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

## 2.2 From sources

The sources for Spade-BDI can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/javipalanca/spade_bdi
```

Or download the tarball:

```
$ curl  -OL https://github.com/javipalanca/spade_bdi/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

# CREATING A BDI AGENT IN SPADE

Belief-Desire-Intention (BDI) agents are a cornerstone of modern agent-based systems. In SPADE (Smart Python multi-Agent Development Environment), creating a BDI agent involves managing the agent's beliefs, desires, and intentions in a dynamic environment. This section provides a guide on setting up and managing a BDI agent in SPADE.

## 3.1 Initial Setup of a BDI Agent

1. **Agent Creation**: - To create a BDI agent, you need to define its Jabber Identifier (JID) and password. The agent is also associated with an AgentSpeak file that defines its initial behaviors.
   - **Initialization Code**:

```
from spade import BDIAgent
agent = BDIAgent("youragent@yourserver.com", "password", "initial_plan.asl")
await agent.start()
```

2. **Defining Initial Beliefs**:
   - The initial beliefs of the agent can be defined in the AgentSpeak file or programmatically set after the agent starts.

## 3.2 Managing Beliefs

1. **Setting Beliefs**: - Beliefs represent the agent's knowledge about the world and can be added or updated using the *set_belief* method. - **Example Code**:

```
agent.bdi.set_belief("key", "value")
```

2. **Retrieving Beliefs**: - To access the current beliefs of the agent, use methods like *get_belief* or *get_beliefs*. - **Example Code**:

```
current_belief = agent.bdi.get_belief("key")
all_beliefs = agent.bdi.get_beliefs()
```

3. **Removing Beliefs**: - Beliefs can be dynamically removed using the *remove_belief* method. - **Example Code**:

```
agent.bdi.remove_belief("key")
```

Creating a BDI agent in SPADE involves initializing the agent with its credentials and defining its initial set of beliefs and plans. The agent's beliefs are dynamically managed, allowing it to adapt to changes in the environment. SPADE's framework offers a flexible and powerful platform for developing sophisticated BDI agents in multi-agent systems.

CHAPTER

FOUR

# THE AGENTSPEAK LANGUAGE

The AgentSpeak language is a logic programming language based on the Belief-Desire-Intention (BDI) model. It is based on the `agentspeak` package, which is a Python implementation of the Jason language. The language is described in the following paper:

> Rao, A. S., & Georgeff, M. P. (1995). BDI agents: From theory to practice. ICMAS. https://cdn.aaai.org/ICMAS/1995/ICMAS95-042.pdf_

This section provides an overview of its syntax and semantics, focusing on how beliefs, desires, and goals are represented and managed in AgentSpeak.

## 4.1 Basic Semantics

- **Beliefs**: In AgentSpeak, beliefs represent the agent's knowledge about the world, itself, and other agents. They are often expressed in a simple predicate form. For example, `is_hot(temperature)` might represent the belief that the temperature is hot.
- **Desires and Goals**: Desires or goals are states or conditions that the agent aims to bring about. In AgentSpeak, these are often represented as special kinds of beliefs or through goal operators. For instance, `!find_shade` could be a goal to find shade.
- **Plans and Actions**: Plans are sequences of actions or steps that an agent will execute to achieve its goals. Actions can be internal (changing beliefs or goals) or external (interacting with the environment).

## 4.2 Syntax of AgentSpeak

AgentSpeak is a logic-based programming language used for creating intelligent agents in multi-agent systems. Understanding its syntax is crucial for effectively programming these agents. This section provides an overview of the key syntactic elements of AgentSpeak.

Basic Elements

- **Beliefs:**
  - Syntax: `belief(arguments)`.
  - Description: Represent the agent's knowledge or information about the world.
  - Example: `is_sunny(true), temperature(25)`.
- **Goals:**
  - Syntax: `!goal(arguments)`.
  - Description: Goals are states or outcomes the agent wants to bring about or information it seeks.

  - Example: `!find_shelter`.

- **Plans:**
  - Syntax: `TriggeringEvent : Context <- Body.`
  - Triggering Event: An event that initiates the plan, such as the addition (+) or deletion (-) of a belief or goal.
  - Context: A logical condition that must hold for the plan to be applicable.
  - Body: A sequence of actions or subgoals to be executed.
  - Example: `+is_raining : is_outside <- !find_umbrella; .print("Hello world").`
- **Actions**
  - Syntax: `.internal_action(arguments)`.
  - Description: Defined by the developer or the environment.
  - Example: `.print("Hello World")`.
- **Communication**
  - **Sending Messages:**
    - Syntax: `.send(receiver, illocution, content)`.
    - Illocutions: Include tell, achieve, askHow, etc.
    - Example: `.send(agentB, tell, is_sunny(true))`.
- **Comments**
  - Single Line Comment: // This is a comment
  - Multi-Line Comment: Not typically supported in standard AgentSpeak.

## 4.3 Creating Agents: Beliefs, Desires, and Goals

Agents are defined by their belief base, goal base, and plan library.

- Example of Beliefs:

```
is_sunny.
temperature(high).
```

This represents beliefs that it is sunny and the temperature is high.

- Example of Goals:

```
!stay_cool.
!drink_water.
```

These are goals to stay cool and to drink water.

- Plans and Actions

A plan in AgentSpeak is a rule that specifies what to do in a given context. Example of a Plan:

```
+!stay_cool : is_sunny & temperature(high) <-
    !find_shade;
    !drink_water.
```

This plan states that to achieve the goal `stay_cool`, if it is sunny and the temperature is high (`is_sunny & temperature(high)`), the agent should achive goals `!find_shade` and `!drink_water` sequentially.

Optionally, a plan may have a custom a name that is set with a tag beginning with a `@`. Example:

```
@my_custom_plan
+!stay_cool : is_sunny & temperature(high) <-
    !find_shade;
    !drink_water.
```

## 4.4 Practical Implications

Understanding these basic concepts is crucial for effectively programming in AgentSpeak. `spade_bdi` provides additional constructs and features, enhancing the basic capabilities of AgentSpeak. When designing agents in SPADE, it is essential to carefully consider the initial set of beliefs and goals, as they drive the agent's behavior through the plans. By grasping these fundamental concepts of AgentSpeak, developers can begin to design and implement sophisticated agents in SPADE, capable of complex decision-making and interactions in dynamic environments. The simplicity of AgentSpeak's syntax, combined with its powerful representational capabilities, makes it a suitable choice for a wide range of applications in multi-agent systems.

### 4.4.1 Variables and the '?' Operator in AgentSpeak

In AgentSpeak, variables are essential for dynamic information processing within an agent's logic. They are uniquely identified by starting with an uppercase letter, distinguishing them from constants and predicates. This section delves into the syntax and use of variables, focusing on the ? operator for retrieving belief values.

## 4.5 Syntax of Variables in AgentSpeak

**Uppercase Naming**: Variables in AgentSpeak are always denoted by names starting with an uppercase letter. This convention distinguishes them from other elements like predicates or constants. Example of Variable Declaration: `Location, Temp, X, Y`

### 4.5.1 Using the `'?'` Operator to Retrieve Belief Values

- **Purpose**: The ? operator in AgentSpeak is used to bind the current value of a belief to a variable. This operation is akin to querying the agent's belief base.
- **Syntax**: To use the ? operator, include it before the belief name and specify the variable in the belief's argument list. The format is typically `?Belief(Variable)`.
- **Example**: If an agent has a belief `location(office)`, and you want to bind the value office to a variable `CurrentLocation`, you would use the statement `?location(CurrentLocation)`.

## 4.6 Practical Application of Variables

- Retrieving and Using Belief Values:

Variables are particularly useful for capturing and utilizing the values of beliefs in plans and decision-making. Example:

```
+!check_current_location
: location(CurrentLocation) & CurrentLocation == "office" <-
.print("The agent is currently in the office").
```

Here, `CurrentLocation` is a variable that retrieves the value from the location belief.

- Dynamic Decision-Making in Contexts:

Variables enable plans to adapt their behavior based on the changing state of the world, as represented by the agent's beliefs. Example:

```
+temperature(Temp) : Temp > 30 <-
    .print("It's currently hot outside").
```

In this example, Temp is a variable that holds the current value of the temperature belief, triggering the plan if Temp exceeds 30.

### 4.6.1 Conclusion

Proper use of variables and the ? operator in AgentSpeak is fundamental for creating dynamic and responsive agents. Variables, identified by their uppercase starting letter, offer a way to handle changing information and make context-sensitive decisions. The ? operator is a key tool for querying and utilizing the agent's belief base, enhancing the agent's ability to interact intelligently with its environment.

CHAPTER

FIVE

# COMMUNICATION IN AGENTSPEAK

## 5.1 Sending Messages

In AgentSpeak and multi-agent systems, communication is a key aspect of agent interaction. This section covers the process and considerations for sending messages between agents in AgentSpeak, with a focus on the syntax, types of messages, and practical implementation.

### 5.1.1 Syntax for Sending Messages

AgentSpeak provides a simple and flexible syntax for sending messages. The general form includes specifying the type of communicative act (ilocution), the recipient agent, and the content of the message.

Basic Syntax:

```
.send(recipient, ilocution, content)
```

where recipient is the identifier of the target agent, ilocution is the type of communicative act, and content is the message content.

Types of Communicative acts: In AgentSpeak, communication between agents is achieved through illocutionary acts, often referred to as communicative acts. Unlike performatives, which are more general in speech act theory, AgentSpeak uses specific types of illocutions to facilitate clear and purpose-driven agent interactions. Here are the key illocutions used in AgentSpeak:

- `tell`: Used to inform another agent about a belief. This act is about sharing knowledge or facts. For example, an agent might tell another agent that a specific condition is true:

```
.send(agentB, tell, weather(raining));
```

- `achieve`: Sent to request another agent to perform some action or bring about a certain state of affairs. This is similar to a request or command in conventional communication:

```
.send(agentB, achieve, fix_the_leak);
```

- `tellHow`: This illocution is used when an agent wants to inform another agent about how to perform a specific action or achieve a goal. It's about sharing procedural knowledge:

```
.send(agentB, tellHow, "+!solve_problem <- !gather_data; !analyze_data.");
```

- `askHow`: When an agent needs to know how to perform an action or achieve a goal, it uses askHow to request this procedural knowledge from another agent.:

```
.send(agentB, askHow, learn_chess);
```

- `untell`: This is used to inform another agent that a previously held belief is no longer true. It's a way of updating or correcting earlier information:

```
.send(agentB, untell, weather(raining));
```

- `unachieve`: Sent to request that another agent cease its efforts to achieve a previously requested goal. It's like a cancellation or retraction of a previous achieve request:

```
.send(agentB, unachieve, fix_the_leak);
```

- `untellHow`: Used to inform another agent to disregard previously told procedural knowledge. This might be used if the procedure is no longer valid or has been updated:

```
.send(agentB, untellHow, "@plan_name");
```

Each of these illocutions plays a vital role in the communication protocol within a multi-agent system, allowing agents to share knowledge, coordinate actions, and update each other on changes in beliefs or plans. When designing AgentSpeak agents, it is crucial to implement these illocutions correctly to ensure effective and coherent agent interactions.

CHAPTER

SIX

# CREATING PLANS IN AGENTSPEAK

In AgentSpeak, plans are central to the behavior of agents. They define how an agent should react to certain events or changes in their environment or internal state. This section explores the syntax and structure of plans in AgentSpeak, providing examples and best practices.

## 6.1 Plan Syntax

**Basic Structure**: A plan in AgentSpeak typically consists of a triggering event, an optional context, and a sequence of actions. The general format is:

```
TriggeringEvent : Context <- Actions.
```

- Triggering Event: This is what initiates the plan. It can be the addition or removal of a belief (+belief or -belief), or the adoption or dropping of a goal (+!goal or -!goal).
- Context: The context is a condition that must be true for the plan to be applicable. It's written as a logical expression.
- Actions: These are the steps the agent will take, interacting with the environment or other agents.
- Tag (Optional): Before the triggering event, a plan may have a tag beginning with a @ and followed by the name of the plan.

## 6.2 Writing a Basic Plan

Example: Suppose an agent needs to respond to a high temperature reading. The plan might look like this:

```
@refresh_plan
+temperature(high) : is_outside <-
    !move_to_shade;
    !drink_water.
```

In this plan, `+temperature(high)` is the triggering event (a belief that the temperature is high). The context `is_outside` checks if the agent is outside. The actions `move_to_shade` and `drink_water` are executed in sequence.

## 6.3 Best Practices in Plan Creation

When designing plans in AgentSpeak, it is important to consider the following best practices:

- Modularity: Keep plans modular. Each plan should have a single, clear purpose.
- Reusability: Design plans that can be reused in different situations.
- Readability: Write clear and understandable plans, as AgentSpeak is a declarative language.

## 6.4 Handling Failures in Plans

Plans should account for potential failures. This can be done through alternative plans or by including failure-handling steps within the plan. Example with Failure Handling:

```
+!travel(destination) : car_is_functional <-
    drive(car, destination).
+!travel(destination) : not car_is_functional <-
    call_taxi(destination).
```

Here, there are two plans for the same goal `!travel(destination)`. The first plan is used if the car is functional, and the second plan (calling a taxi) is a backup if the car isn't functional.

CHAPTER

SEVEN

# MANAGING LISTS IN AGENTSPEAK

In AgentSpeak, lists are important data structures that enable agents to handle collections of items. While AgentSpeak does not offer the same list manipulation capabilities as imperative programming languages, it still provides ways to manage lists through pattern matching and recursion. This section explores how AgentSpeak handles lists.

## 7.1 List Structure in AgentSpeak

- **Representation**: Lists in AgentSpeak are represented as a collection of elements enclosed in brackets and separated by commas, e.g., `[element1, element2, element3]`.
- **Head and Tail**: Lists can be split into a "head" (the first element) and a "tail" (the remainder of the list). This is done using the pattern `[Head|Tail]`.

## 7.2 Basic Operations on Lists

1. **Accessing Elements**: - The first element of the list (head) and the rest (tail) can be accessed using list decomposition. - **Example**:

```
+!process_list([Head|Tail]) : true <-
    .print("Processing", Head);
    !process_list(Tail).
```

2. **Adding Elements**: - AgentSpeak does not have a direct operation for adding elements, but this can be achieved by updating a list. - **Example**:

```
+!add_element(Element) : list([List]) <-
    -+list([Element|List]).
```

3. **Removing Elements**: - Similar to adding elements, removing requires updating the list without the element to be removed. - **Example**:

```
+!remove_element(Element) : list([Element|Tail]) <-
    -+list([Tail]).
```

## 7.3 Recursion in List Handling

- **Recursive Processing**: To process lists, recursion is often used, where a plan calls itself with the list's "tail" until the list is empty.
- **Example of Recursion**:

```
+!process_list([Head|Tail]) : .length(Tail, X) & X > 0  <-
    .do_something_with(Head);
    !process_list(Tail).

+!process_list([LastElement]) : true  <-
    .do_something_with(LastElement).
```

Managing lists in AgentSpeak, although not as straightforward as in other languages, is feasible and effective through list decomposition, creating new lists for adding or removing elements, and recursive patterns to process lists. These methods enable agents to dynamically handle sets of data and are essential for developing complex behaviors in multi-agent systems.

CHAPTER

EIGHT

# CREATE CUSTOM ACTIONS AND FUNCTIONS

You must to overload the `add_custom_actions` method and to use the `add_function` or `add` (for actions) decorator. This custom method receives always the `actions` parameter:

```
import spade_bdi

class MyCustomBDIAgent(BDIAgent):

    def add_custom_actions(self, actions):
        @actions.add_function(".my_function", (int,))
        def _my_function(x):
            return x * x

        @actions.add(".my_action", 1)
        def _my_action(agent, term, intention):
            arg = agentspeak.grounded(term.args[0], intention.scope)
            print(arg)
            yield
```

**Hint:** Adding a function requires to call the `add_function` decorator with two parameters: the name of the function (starting with a dot) and a tuple with the types of the parameters (e.g. `(int, str)`).

**Hint:** Adding an action requires to call the `add` decorator with two parameters: the name of the action (starting with a dot) and the number of parameters. Also, the method being decorated receives three parameters: `agent`, `term`, and `intention`.

CHAPTER

NINE

# CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 9.1 Types of Contributions

### 9.1.1 Report Bugs

Report bugs at https://github.com/javipalanca/spade_bdi/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 9.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 9.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 9.1.4 Write Documentation

Spade-BDI could always use more documentation, whether as part of the official Spade-BDI docs, in docstrings, or even on the web in blog posts, articles, and such.

### 9.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/javipalanca/spade_bdi/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 9.2 Get Started!

Ready to contribute? Here's how to set up *spade_bdi* for local development.

1. Fork the *spade_bdi* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/spade_bdi.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv spade_bdi
$ cd spade_bdi/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 spade_bdi tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 9.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/javipalanca/spade_bdi/pull_requests and make sure that the tests pass for all supported Python versions.

## 9.4 Tips

To run a subset of tests:

```
$ py.test tests.test_spade_bdi
```

## 9.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

# TEN

# CREDITS

## 10.1 Development Lead

- Sergio Frayle Pérez <sfp932705@gmail.com>

## 10.2 Contributors

- Javi Palanca <jpalanca@gmail.com>

CHAPTER

# ELEVEN

# HISTORY

## 11.1 0.3.2 (2025-03-01)

- Updated to SPADE 4.0.1
- Updated dependencies
- License changed to MIT

## 11.2 0.3.1 (2024-01-06)

- Added new examples
- Added documentation
- Updated to SPADE 3.3.2

## 11.3 0.3.0 (2023-06-13)

- Updated to SPADE 3.3.0.

## 11.4 0.2.2 (2022-06-03)

- Added exception when belief is not initialized.
- Improved examples.
- Improved documentation.

## 11.5 0.2.1 (2020-04-13)

- Fixed a bug when updating beliefs.
- Upgraded spade version to 3.1.4.

## 11.6 0.2.0 (2020-02-24)

- Created add_custom_actions method.
- Added example for actions.
- Improved documentation.
- Added some helpers like pause_bdi, resume_bdi.
- Now the asl file in the constructor is mandatory.

## 11.7 0.1.4 (2019-07-10)

- Allow to send messages to JIDs stored as beliefs.

## 11.8 0.1.3 (2019-07-08)

- Allow .send to a list of receivers.
- Allow to receive messages with lists of lists.
- Fixed readme.

## 11.9 0.1.1 (2019-06-18)

- Moved from pyson to python-agentspeak
- Added some helpers like pause_bdi, resume_bdi.
- Now the asl file in the constructor is mandatory.
- Allow to send tell messages with no args.
- Allow sending messages with variables.
- Extended the examples.

## 11.10 0.1.0 (2019-03-09)

- First release on PyPI.

CHAPTER

TWELVE

# INDICES AND TABLES

- genindex
- modindex
- search