



SPADE – Agent Behaviours

Agentes e Sistemas Multiagente
Guilherme Barbosa, José Machado

SPADE Agent Behaviours

- A behaviour is a task that an agent can execute using repeating patterns
- SPADE provides predefined several behaviour types:
 - *OneShotBehaviour & TimeoutBehaviour* – applied to perform casual tasks
 - *PeriodicBehaviour & CyclicBehaviour* - applied for performing repetitive tasks
 - *Finite State Machine (FSMBehaviour)* – applied for more complex behaviours to be built
- Each SPADE agent can run several behaviors simultaneously

SPADE Agent Behaviours - OneShotBehaviour

- *OneShotBehaviour* is an atomic behaviour that executes just once
- This abstract class can be extended by application programmers to create behaviours for operations that need to be done just one time.

```
$ python sender.py
SenderAgent started
InformBehav running
Message sent!
Agent finished with exit code: Job Finished!
```

```
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour
from spade.message import Message

class SenderAgent(Agent):
    class InformBehav(OneShotBehaviour):
        async def run(self):
            print("InformBehav running")
            msg = Message(to="receiver@your_xmpp_server")      # Instantiate the message
            msg.set_metadata("performative", "inform")          # Set the "inform" FIPA performative
            msg.set_metadata("ontology", "myOntology")          # Set the ontology of the message content
            msg.set_metadata("language", "OWL-S")               # Set the language of the message content
            msg.body = "Hello World"                            # Set the message content

            await self.send(msg)
            print("Message sent!")

        # set exit_code for the behaviour
        self.exit_code = "Job Finished!"

        # stop agent from behaviour
        await self.agent.stop()

    async def setup(self):
        print("SenderAgent started")
        self.b = self.InformBehav()
        self.add_behaviour(self.b)
```



SPADE Agent Behaviours - TimeoutBehaviour

- *TimeoutBehaviour* is a behaviour which is run once (like *OneShotBehaviour*)
 - However, its activation is triggered at a specified *datetime*

```
$python timeout.py
TimeoutSenderAgent started at 18:12:09.620316
TimeoutSenderBehaviour running at 18:12:14.625403
Message received with content: Hello World
Did not received any message after 10 seconds
Agents finished
```

```
import datetime
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour, TimeoutBehaviour
from spade.message import Message

class TimeoutSenderAgent(Agent):
    class InformBehav(TimeoutBehaviour):
        async def run(self):
            print(f"TimeoutSenderBehaviour running at {datetime.datetime.now().time()}")
            msg = Message(to=self.get("receiver_jid")) # Instantiate the message
            msg.body = "Hello World" # Set the message content

            await self.send(msg)

        async def on_end(self):
            await self.agent.stop()

        async def setup(self):
            print(f"TimeoutSenderAgent started at {datetime.datetime.now().time()}")
            start_at = datetime.datetime.now() + datetime.timedelta(seconds=5)
            b = self.InformBehav(start_at=start_at)
            self.add_behaviour(b)

class ReceiverAgent(Agent):
    class RecvBehav(CyclicBehaviour):
        async def run(self):
            msg = await self.receive(timeout=10) # wait for a message for 10 seconds
            if msg:
                print("Message received with content: {}".format(msg.body))
            else:
                print("Did not received any message after 10 seconds")
                self.kill()

        async def on_end(self):
            await self.agent.stop()

        async def setup(self):
            b = self.RecvBehav()
            self.add_behaviour(b)
```



SPADE Agent Behaviours - PeriodicBehaviour

- *PeriodicBehaviour* is a behaviour that is executed periodically with an interval (set in seconds)
- Can also delay its startup by setting a *datetime* in the *start_at* parameter (similar to *TimeoutBehaviour*)

```
$ python periodic.py
ReceiverAgent started
RecvBehav running
PeriodicSenderAgent started at 17:40:39.901903
PeriodicSenderBehaviour running at 17:40:45.720227: 0
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:46.906229: 1
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:48.906347: 2
Message sent!
```

```
class PeriodicSenderAgent(Agent):
    class InformBehav(PeriodicBehaviour):
        async def run(self):
            print(f"PeriodicSenderBehaviour running at {datetime.datetime.now().time()}: {self.counter}")
            msg = Message(to=self.get("receiver_jid")) # Instantiate the message
            msg.body = "Hello World" # Set the message content

            await self.send(msg)
            print("Message sent!")

            if self.counter == 5:
                self.kill()
            self.counter += 1

        async def on_end(self):
            # stop agent from behaviour
            await self.agent.stop()

        async def on_start(self):
            self.counter = 0

        async def setup(self):
            print(f"PeriodicSenderAgent started at {datetime.datetime.now().time()}")
            start_at = datetime.datetime.now() + datetime.timedelta(seconds=5)
            b = self.InformBehav(period=2, start_at=start_at)
            self.add_behaviour(b)

class ReceiverAgent(Agent):
    class RecvBehav(CyclicBehaviour):
        async def run(self):
            print("RecvBehav running")
            msg = await self.receive(timeout=10) # wait for a message for 10 seconds
            if msg:
                print("Message received with content: {}".format(msg.body))
            else:
                print("Did not receive any message after 10 seconds")
                self.kill()

        async def on_end(self):
            await self.agent.stop()

        async def setup(self):
            print("ReceiverAgent started")
            b = self.RecvBehav()
            self.add_behaviour(b)
```



SPADE Agent Behaviours - CyclicBehaviour

- *CyclicBehaviour* is a behaviour that is executed cyclically until it is stopped
- Normally every agent has at least 1 *CyclicBehaviour* active responsible to process receiving messages

```
$ python periodic.py
ReceiverAgent started
RecvBehav running
PeriodicSenderAgent started at 17:40:39.901903
PeriodicSenderBehaviour running at 17:40:45.720227: 0
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:46.906229: 1
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:48.906347: 2
Message sent!
```

```
class PeriodicSenderAgent(Agent):
    class InformBehav(PeriodicBehaviour):
        async def run(self):
            print(f"PeriodicSenderBehaviour running at {datetime.datetime.now().time()}: {self.counter}")
            msg = Message(to=self.get("receiver_jid")) # Instantiate the message
            msg.body = "Hello World" # Set the message content

            await self.send(msg)
            print("Message sent!")

            if self.counter == 5:
                self.kill()
                self.counter += 1

        async def on_end(self):
            # stop agent from behaviour
            await self.agent.stop()

        async def on_start(self):
            self.counter = 0

        async def setup(self):
            print(f"PeriodicSenderAgent started at {datetime.datetime.now().time()}")
            start_at = datetime.datetime.now() + datetime.timedelta(seconds=5)
            b = self.InformBehav(period=2, start_at=start_at)
            self.add_behaviour(b)

class ReceiverAgent(Agent):
    class RecvBehav(CyclicBehaviour):
        async def run(self):
            print("RecvBehav running")
            msg = await self.receive(timeout=10) # wait for a message for 10 seconds
            if msg:
                print("Message received with content: {}".format(msg.body))
            else:
                print("Did not received any message after 10 seconds")
                self.kill()

        async def on_end(self):
            await self.agent.stop()

        async def setup(self):
            print("ReceiverAgent started")
            b = self.RecvBehav()
            self.add_behaviour(b)
```

SPADE Agent Behaviours – Finite State Machine Behaviour (*FSMBehaviour*)

- Agents can also have complex behaviours, which has registered states and transitions between states
 - Enables the sequential execution of several *OneshotBehaviours*
- *FSMBehaviour* is a behaviour composed of states (*OneshotBehaviours*) that may transition from one state to another

SPADE Agent Behaviours – Finite State Machine Behaviour (*FSMBehaviour*)

- *FSMBehaviour* class is a container behaviour that implements the methods:
 - *add_state(name, state, initial)* -> adds a new state to the FSM
 - Every state of the FSM must be registered in the behaviour with a string name and an instance of the *State* class – representing a node/state of the FSM
 - A FSM can only have 1 initial state- to mark a *State* as the FSM initial's state, set the *initial* parameter to *True*, e.g., *add_state(name, state, initial=True)*
 - A *State* defines its transit to another *State* by using the *set_next_state(state_name)* method
 - Dynamically expresses to which state it transits when it finishes
 - *add_transition(source, dest)* -> Adds a transition from one state to another
 - Transitions define from which state to which state it is allowed to transit

SPADE Agent Behaviours – Finite State Machine Behaviour (*FSMBehaviour*)

FSMBehaviour

```
from spade.agent import Agent
from spade.behaviour import FSMBehaviour, State
from spade.message import Message

STATE_ONE = "STATE_ONE"
STATE_TWO = "STATE_TWO"
STATE_THREE = "STATE_THREE"

class ExampleFSMBehaviour(FSMBehaviour):
    async def on_start(self):
        print(f"FSM starting at initial state {self.current_state}")

    async def on_end(self):
        print(f"FSM finished at state {self.current_state}")
        await self.agent.stop()
```

FSMBehaviour States

```
class StateOne(State):
    async def run(self):
        print("I'm at state one (initial state)")
        msg = Message(to=str(self.agent.jid))
        msg.body = "msg_from_state_one_to_state_three"
        await self.send(msg)
        self.set_next_state(STATE_TWO)

class StateTwo(State):
    async def run(self):
        print("I'm at state two")
        self.set_next_state(STATE_THREE)

class StateThree(State):
    async def run(self):
        print("I'm at state three (final state)")
        msg = await self.receive(timeout=5)
        print(f"State Three received message {msg.body}")
        # no final state is setted, since this is a final state
```

Agent Configuration

```
class FSMAgent(Agent):
    async def setup(self):
        fsm = ExampleFSMBehaviour()
        fsm.add_state(name=STATE_ONE, state=StateOne(), initial=True)
        fsm.add_state(name=STATE_TWO, state=StateTwo())
        fsm.add_state(name=STATE_THREE, state=StateThree())
        fsm.add_transition(source=STATE_ONE, dest=STATE_TWO)
        fsm.add_transition(source=STATE_TWO, dest=STATE_THREE)
        self.add_behaviour(fsm)
```



SPADE Agent Behaviours – *on_start/on_end*

- In some cases, we may need to execute tasks before or after the *Behaviour's run()* method is executed
- To do this, SPADE behaviour classes provide the methods called *on_start()* and *on_end()*:
 - *on_start()* method is executed before the *run()* method of the respective behaviour is started
 - *on_end()* method is executed after the *run()* method of the respective behaviour is finished

```
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour

class DummyAgent(Agent):
    class MyBehav(CyclicBehaviour):
        async def on_start(self):
            print("Starting behaviour . . .")
            self.counter = 0

        async def run(self):
            print("Counter: {}".format(self.counter))
            self.counter += 1
            if self.counter > 3:
                self.kill(exit_code=10)
                return
            await asyncio.sleep(1)

        async def on_end(self):
            print("Behaviour finished with exit code {}".format(self.exit_code))

    async def setup(self):
        print("Agent starting . . .")
        self.my_behav = self.MyBehav()
        self.add_behaviour(self.my_behav)

$ python killbehav.py
Agent starting . . .
Starting behaviour . . .
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Behaviour finished with exit code 10.
```





SPADE Agent Behaviours – Finishing a Behaviour

- To finish a behaviour, the `kill(exit_code)` method is available (executed inside the respective behaviour)
- This method marks the behaviour to be finished at the next loop iteration and stores the `exit_code` to be queried later

```
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour

class DummyAgent(Agent):
    class MyBehav(CyclicBehaviour):
        async def on_start(self):
            print("Starting behaviour . . .")
            self.counter = 0

        async def run(self):
            print("Counter: {}".format(self.counter))
            self.counter += 1
            if self.counter > 3:
                self.kill(exit_code=10)
                return
            await asyncio.sleep(1)

        async def on_end(self):
            print("Behaviour finished with exit code {}".format(self.exit_code))

        async def setup(self):
            print("Agent starting . . .")
            self.my_behav = self.MyBehav()
            self.add_behaviour(self.my_behav)
```

```
$ python killbehav.py
Agent starting . . .
Starting behaviour . . .
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Behaviour finished with exit code 10.
```





SPADE Agent Behaviours – Finishing an Agent

- To finish an Agent, the `stop()` method is available
 - Can be executed inside the respective behaviour via `self.agent.stop()`
 - Can be executed inside the Agent class via `self.stop()`
- This method informs the server to stop this agent

```
class PeriodicSenderAgent(Agent):
    class InformBehav(PeriodicBehaviour):
        async def run(self):
            print(f"PeriodicSenderBehaviour running at {datetime.datetime.now().time()}: {self.counter}")
            msg = Message(to=self.get("receiver_jid")) # Instantiate the message
            msg.body = "Hello World" # Set the message content

            await self.send(msg)
            print("Message sent!")

            if self.counter == 5:
                self.kill()
                self.counter += 1

        async def on_end(self):
            # stop agent from behaviour
            await self.agent.stop()

        async def on_start(self):
            self.counter = 0

        async def setup(self):
            print(f"PeriodicSenderAgent started at {datetime.datetime.now().time()}")
            start_at = datetime.datetime.now() + datetime.timedelta(seconds=5)
            b = self.InformBehav(period=2, start_at=start_at)
            self.add_behaviour(b)
```

SPADE Agent Behaviours – Assess Agent's global variables

- *Behaviours* can also:
 - Set/Get values from their respective Agent's global variables
 - Re-use functions defined in the Agent class
- For this, inside the behaviour class, use the `self.agent.*` method

```
runner.py > SellerAgent > setup
import random
import datetime
from spade import agent

from Behaviours.profReview_Behav import ProfReviewBehav
from Behaviours.receiveRequests_Behav import ReceiveRequestBehav

class SellerAgent(agent.Agent):

    products_sold = {}
    products_value = {}

    async def setup(self):
        print("Agent {}".format(str(self.jid)) + " starting...")

        # Initialise quantity of products sold per product in the list provided
        for i in self.get("products"):
            self.products_sold[i] = 0                                # all products yet not sold, i.e., equal to 0 for each product
            self.products_value[i] = random.randint(1, 10)           # define a random cost for each product

        a = ReceiveRequestBehav()
        b = ProfReviewBehav(period=10)                            # CyclicBehav to verify buy requests from clients
                                                                # Every 10 seconds, PeriodicBehav will calculate profit

        self.add_behaviour(a)
        self.add_behaviour(b)
```

```
Behaviours > profReview_Behav.py > ...
1  from spade.behaviour import PeriodicBehaviour
2
3  class ProfReviewBehav (PeriodicBehaviour):
4      async def run(self):
5          # Initiate profit value
6          profit = 0
7
8          # Calculate profit based on products value and products sold
9          for i in self.agent.products_sold:
10              profit += (self.agent.products_sold[i] * self.agent.products_value[i])
11
12          print("-" * 20)
13          print("Agent {}".format(str(self.agent.jid)) + " Profit = {}".format(profit))
14          print("-" * 20)
15
16          print("\n\n")
```



SPADE Agent Behaviours – Create an Agent from within another Agent

- Agents can also be created from within another Agent's behaviour
- For this special case, you can use the start() method as usual
- Since Agents and behaviours are asynchronous methods, the start() method MUST be called with an *await* statement

```
import spade
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour

class AgentExample(Agent):
    async def setup(self):
        print(f"{self.jid} created.")

class CreateBehav(OneShotBehaviour):
    async def run(self):
        agent2 = AgentExample("agent2_example@your_xmpp_server", "fake_password")
        await agent2.start(auto_register=True)

async def main():
    agent1 = AgentExample("agent1_example@your_xmpp_server", "fake_password")
    behav = CreateBehav()
    agent1.add_behaviour(behav)
    await agent1.start(auto_register=True)

    # wait until the behaviour is finished to quit spade.
    await behav.join()

if __name__ == "__main__":
    spade.run(main())
```

SPADE Agent Behaviours – Waiting for a Behaviour

- In some cases, we may need to wait for a behaviour to finish
- To do this, SPADE behaviour classes provide a method called *join()*
 - Makes a behaviour instance wait for another behaviour instance to be finished

```
import asyncio
import getpass

import spade
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour

class DummyAgent(Agent):
    class LongBehav(OneShotBehaviour):
        async def run(self):
            await asyncio.sleep(5)
            print("Long Behaviour has finished")

    class WaitingBehav(OneShotBehaviour):
        async def run(self):
            await self.agent.behav.join() # this join must be awaited
            print("Waiting Behaviour has finished")

    async def setup(self):
        print("Agent starting . . .")
        self.behav = self.LongBehav()
        self.add_behaviour(self.behav)
        self.behav2 = self.WaitingBehav()
        self.add_behaviour(self.behav2)
```



SPADE – Agent Behaviours

Agentes e Sistemas Multiagente
Guilherme Barbosa, José Machado