MULTI AGENT SYSTEM DESIGN

Assignment 1

2APL

Alvaro Armada Ruiz
alvaro.armada@estudiantat.upc.edu

Benedikt Janik Krauss
benedikt.janik.krauss@estudiantat.upc.edu

Hajar Lachheb
hajar.lachheb@estudiantat.upc.edu

Ruben Vera Garcia
ruben.vera@estudiantat.upc.edu

Zixi Chen
zixi.chen@estudiantat.upc.edu

10/04/2023

1 Introduction

2APL (A Practical Agent Programming Language) is a programming language designed for developing multi-agent systems. It is based on the BDI (Belief-Desire-Intention) model of agency, which allows agents to reason about their beliefs, desires, and intentions in a goal-directed manner. In 2APL, agents are defined as collections of beliefs, desires, and intentions. Beliefs represent the agent's knowledge of the world, desires represent the agent's goals or objectives, and intentions represent the agent's plans for achieving its goals.

2APL agents communicate with each other through message passing, which allows them to exchange information and coordinate their actions. The language also supports the use of external libraries for tasks such as communication, perception, and action. One of the unique features of 2APL is its support for programming with norms. Norms are rules that define expected behavior and can be used to regulate the behavior of agents. For example, a norm might specify that agents should not steal from each other or should always help other agents in need.

Overall, 2APL provides a flexible and powerful platform for developing intelligent multi-agent systems. Its BDI-based architecture and support for programming with norms make it a popular choice for researchers and practitioners in the field of multi-agent systems.

2 Element in the language and Formalism

2APL, an agent-oriented programming language, offers a range of abstractions for defining agent behavior. Until stated otherwise, we will focus specifically on the **hybrid version of 2APL**. This version of 2APL is a robust and feature-rich language that incorporates various updates and enhancements. It provides essential abstractions that enable the specification of agent behavior. Some of the key abstractions offered by 2APL include:

- ***B*eliefs:** Beliefs represent the agent's knowledge or information about the world. They are expressed as logical statements or propositions that the agent holds to be true. Beliefs can be updated and modified based on incoming information or changes in the environment.

  ```
  beliefs:
      counter(X).
  ```

  Beliefs in agent systems are not limited to being solely positive; they can also be negative, representing the absence or negation of a certain condition. To illustrate this, let's consider the following example. Let's say we have an agent that represents a bomb disposal robot in a dangerous environment. The agent's belief is defined as follows:

  ```
  beliefs:
      not bomb(X,Y).
  ```

  In this case, the belief "not bomb(X,Y)" represents the absence of a bomb at location (X,Y). The agent maintains this belief until it receives information indicating the presence of a bomb at a specific location.

- ***A*ctions:** Actions represent the executable behaviors or operations that an agent can perform. They can include physical actions, communication actions, or any other relevant

operations. Actions are triggered by plans or events in the environment. There are indeed different types of actions in the Hybrid 2APL, we can mention some :

- **Belief Update Actions:** These actions are used to update the agent's beliefs based on new information or changes in the environment. They can add or remove beliefs from the agent's belief base.

```
+available;
-status(busy);
```

  We can also "automate" the process, by creating belief update actions. They have a precondition, the name of the update, and then the postcondition. When we use them, if the precondition is true, it will update the belief to match the postconditions:

```
{ bomb(X,Y) } RemoveBomb(X,Y) { not bomb(X,Y) }
```

- **Test Actions:** Test actions are used to check the truth value of a particular belief or condition. They allow agents to evaluate the state of the world or verify specific conditions before making decisions or taking further actions. For the example below, in case we have in our beliefs "start(0,1)", it will result in the substitution [X/0 , Y/1].

```
B(start(X,Y))
```

- **Goal Dynamics Actions:** These actions allow agents to manipulate their goals dynamically. Agents can add, remove, or modify their goals based on changing circumstances or internal decision-making processes.

```
adoptz(subgoal);
dropgoal(subgoal);
```

- **Abstract Actions:** Abstract actions are used to represent higher-level behaviors or complex actions that can be decomposed into a sequence of primitive actions or subactions. They provide a way to modularize agent behavior and promote reusability.

```
goto(X, Y);
```

- **Communication Actions:** Communication actions enable agents to interact and exchange information with other agents in the system. They facilitate the coordination, cooperation, and negotiation between agents.

```
send(Receiver, Performative, Language, Ontology, Content)
```

- **External Actions:** External actions represent interactions with the environment outside the scope of the agent system. They can include physical actions, interactions with sensors or actuators, or any external operations required by the agent.

```
@blockworld( enter( X, Y, blue ), _ );
```

Regarding intentions, it is important to note that in hybrid 2APL, there are no explicit

intentions. The agent executes all PG-rules applicable at any time without relying on intentions.

- ***R*ules:** Rules define the decision-making and reasoning processes of an agent. They consist of conditions and actions that guide the agent's behavior. Rules help agents make choices, update beliefs, achieve goals, or respond to external events. In hybrid 2APL, rules play a significant role in agent behavior. Let's explore the three types of rules:

  - **PG-Rules (Percept-Goal Rules):** These rules match the percept (or external events) and the current goals of the agent. The body of the rule is executed when both the percept and the goals match the rule's head.

    ```
    <query>? "<-" <query> "|" <plan>
    ```

    The code fragment below is an example of a planning goal rule of Harry indicating that a plan to achieve the goal clean(blockworld) can be generated if the agent believes there is a bomb at position (X,Y). Note that goto(X,Y) is an abstract action the execution of which replaces a plan for going to position (X,Y). After performing this plan, Harry performs a pickup action in the blockworld, modifying his beliefs such that he now believes he carries a bomb, modifying his beliefs that there was a bomb at (X,Y), going to position (0,0), where the bomb should be dropped in a dustbin, perform the drop action in the blockworld, and finally modifying his beliefs that he does not carry a bomb anymore.

    ```
    pgrules:
    clean( blockworld ) <- bomb( X, Y ) |
    {
        goto( X, Y );
        @blockworld( pickup( ), _ );
        PickUp( );
        -bomb( X, Y );
        goto( 0, 0 );
        @blockworld( drop( ), _ );
        Drop( );
    }
    ```

  - **PC-Rules (Percept-Condition Rules):** Procedural rules generate plans as a response to 1) the reception of messages sent by other agents, 2) events generated by the external environment, and 3) the execution of abstract actions.

    ```
    <atom>? "<-" <query> "|" <plan>
    ```

    The code fragment below shows an example of procedural call rules that is used for implementing Harry. This rule indicates that if Harry receives a message from Sally informing him that there is a bomb at position (X,Y), then his beliefs will be updated with this new fact, and the goal to clean the blockworld is adopted, if he does not believe that there is already a bomb at a position (A,B). Otherwise, he updates his beliefs with the received information without adopting the goal. This is because Harry is assumed to this goal as long as he believes there is a bomb somewhere.

    ```
    pcrules:
    message( sally, inform, La, On, bombAt( X, Y ) ) <- true |
    ```

```
    {
        if B( not bomb( A, B ) )
            { +bomb( X, Y );
                adoptz( clean( blockworld ) );
            }
        else
            { +bomb( X, Y );
            }
    }
```

– **PR-Rules (Plan Rule):** PR-rules match the remaining part of a plan that has not been executed yet. The head of the rule matches the remaining part of the plan, and the body is executed when the head matches.

```
<planvar> "<-" <query> "|" <planvar>
```

The code fragment below shows an example of a plan repair rule of harry. This rule is used for the situation in which the execution of a plan that starts with the external action @blockworld(pickup(), _); fails. Such an action fails in case there is no bomb to be picked up (e.g., when it is removed by another agent). The rule states that the plan should be replaced by another plan consisting of an external action to sense its current position after which it removes the bomb from its current position. Note that in this case the rest of the original plan denoted by REST is dropped, as it is not used anymore within the rule.

```
prrules:
@blockworld( pickup(), _ ); REST; <- true |
    {
        @blockworld( sensePosition(), POS );
        B(POS = [X,Y]);
        RemoveBomb( X, Y );
    }
```

- ***G*oals:** Goals define the desired states or objectives that an agent aims to achieve. They represent the agent's intentions and provide a direction for its actions. Goals can be dynamic and change as the agent's priorities or circumstances evolve.

```
goals:
    clean ( blockWorld ) .
```

- ***P*lans:** Plans are a set of predefined sequences of actions that an agent can execute to achieve its goals. They represent the agent's strategies or courses of action in response to specific situations. Plans are associated with conditions or triggers that activate them when certain conditions are met.

```
plans:
    B ( start (X , Y ) ) ;
    @blockworld ( enter ( X , Y , blue ) , _ ) ;
```

In summary, 2APL provides a range of abstractions for representing the various aspects of an

agent's behavior, including the agent's beliefs, actions, goals, plans, and tasks. These abstractions can be used to define the behavior of complex multi-agent systems, as well as to reason about the behavior of individual agents in different scenarios.

3 Operational Semantics

3.1 Reasoning cycle

As stated previously, 2APL is a BDI-based language. In other words, an agent initial (cognitive) state is set with Beliefs, the information that the agent has, Goals, the objectives of the agent, and Plans, what the agent needs to perform to achieve the goals.

This initial state of the agent will be modified during its execution with a deliberation cycle process, as shown in Figure 1.

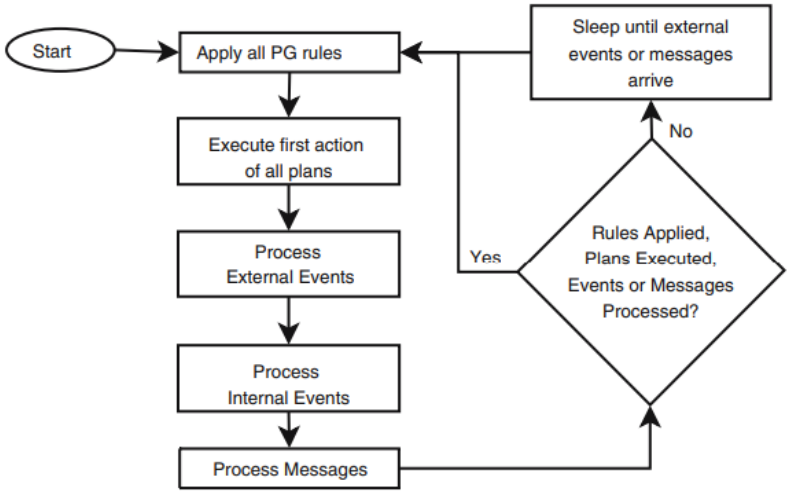

Fig. 1: The deliberation cycle of a 2APL agent. Extracted from [**2**]

This process starts with the agent already having its internal state ready and programmed. When the execution is started, the agent will execute the deliberation cycle with the following steps:

- **Apply all Planning Goal rules (PG-rules) possible** to execute new plans based on current Goals and Beliefs of the agent. In other words, all PG-rules are matched if possible with their own goal base and, thus, only those plans are started.
- Afterward, the agent proceeds by **executing only the first action of all the plans** started in the previous step, so each plan has a chance to be executed and not only the first available plan in the list.
- Next, the agent will process all **external events** from the environment with Procedure Call rules (PC-rules), following with the **internal events**.
  - The external events will process all the events in which the agent knows that the environment has changed, and has to potentially change its internal state in response.

  - The internal events have the objective of trying to repair those plans that their execution has failed, applying Plan Repair rules (PR-rules).

- Lastly, from the main cycle, the agent will process the messages received from other agents with Procedure Call rules. It is important to notice that the agent will always apply the first applicable rule in these processing sections.

- At this moment, the agent needs to check if it will **perform another deliberation cycle** or **sleep** until external events from the environment or messages from other agents are received. The agent will decide to sleep when no rule could be applied, no plan could be executed and no events or messages had been processed, in other words, if the last cycle had no changes for the agent.

3.2 Semantic elements

There is the necessity to state in detail the most important aspects that the agent can execute and perform or receive during the deliberation cycle. Furthermore, the elements will be explained by answering the questions provided by the assignment.

- ***Is there any Belief revision?*** *Yes, the Belief Update Action.* The agent can execute this action when the pre-conditions are inside the belief base and when finished, its post-conditions are added or removed from the belief base. In other words, this action will update the belief base of the agent.

- ***Is there any Goal/Intention revision?*** *Yes, the Goal Dynamics Actions.* The agent can utilize these actions to add or remove a certain goal from its base. There are five distinct Goal Dynamic actions:

  - *adopta(ϕ):* Add a new goal at the beginning of the goal base.
  - *adoptz(ϕ):* Add a new goal at the end of the goal base.
  - *dropgoal(ϕ):* Drop a specific goal of the goal base.
  - *dropsubgoals(ϕ):* Drop all goals that are subgoals from a specific goal.
  - *dropsupergoal(ϕ):* Drop all goals that need the specific goal to be achieved.

- ***Is there any Plan/Task revision?*** *Yes, the Practical Reasoning Rules.* They enable 2APL agents to generate new plans to add to their base. There are three distinct types:

  - *Planning Goal rules (PG-rules):* As explained in previous sections, PG-rules create new plans based on the current Goals and Beliefs base of the agent.
  - *Procedure Call rules (PC-rules):* PC-rules create a new plan when messages from other agents or events from the environment are processed by the agent.
  - *Plan Repair rules (PR-rules):* When a plan fails, the agent can repair it if it has specific belief/s inside its base.

- ***Is there any way to make agents react to sudden changes in the environment?*** *Yes, the events* can be utilized to pass information from the environment to agents. The agent can react to sudden changes and special events in the environment already specified by the programmer. When an event is activated, the method *notifyEvent()* will be called from the environment's constructor and sent to the agents in the list.

- ***Is there any special events' handling?*** *Yes, the exceptions* are received by the agent and then used to apply Plan Repair rules when the execution of a plan was not successful. The exception will contain the identifier of the failed plan such that it can be determined the plan that needs to be repaired.
- ***Is there any Meta-Level reasoning?*** *Yes, agents in 2APL can query their goal and belief bases to retrieve the information in order to reason about its own knowledge and motivations plus modify them based on their reasoning. Furthermore, agents can inspect their own plans to reason their decision-making processes and change them depending on the current situation at runtime, facilitating meta-level reasoning.*

4 Formal semantics

The operational semantics of 2APL are defined in terms of configurations and a transition system.

In 2APL, there are two main types of configurations:

1. **Agent Configuration**: This represents the current state of a single agent. An agent configuration is a tuple represented as (B, G, P, PC, C, E), where:
   - B is the agent's belief base, a set of ground atoms representing what the agent currently believes about the world.
   - G is the agent's goals, which represent the states of the world the agent aims to bring about.
   - P is the set of plans, where each plan is a sequence of actions that the agent is currently executing.
   - PC is the set of plan candidates, which are sequences of actions that the agent could potentially execute if the preconditions are satisfied.
   - C is the set of capabilities, which are the actions the agent can perform.
   - E is the set of events, which are occurrences that the agent perceives from the environment or from other agents.
2. **Multiagent Configuration**: This represents the current state of a multi-agent system. A multiagent configuration in 2APL is a tuple represented as (A, Env), where:
   (a) A is the set of agent configurations, representing all the agents in the system.
   (b) Env is the shared environment in which the agents operate. This could be a physical environment (e.g., a map of a city) or a more abstract environment (e.g., a market).

The operational semantics also include a transition system, which specifies how the configurations can change over time. Each transition rule in the system specifies a condition under which a configuration can change, and the effect of the change. Here's an example of a transition rule:

Rule (Belief Revision):
Condition: There is an event e in E, a belief b in B, and a rule r in the agent's program such that r specifies that when event e occurs, belief b should be added to B.
Effect: The event e is removed from E, and the belief b is added to B.

5 Connecting with the Environment

5.1 The 2APL Platform

A complete tool that aids in the construction of multi-agent systems is the 2APL platform. It offers a wide range of functionalities and development tools that help by creating sophisticated multi-agent systems. Some main characteristics of the 2APL platform include the following:

- *Agent Management:* The platform enables programmers to control numerous agents in a single environment. By abstracting away the intricate complexities of agent management, it offers a single interface for agents APIs, which streamlines the development process.
- *Execution Modes:* The platform supports running programs on multiple agents in two different execution modes: stand-alone and distributed. The distributed mode uses an interface to the JADE platform (see 'JADE Interface'). This mode is useful when one wants to run a multi-agent program in a distributed manner on different machines. If not, the stand-alone mode is sufficient, as it relies entirely on its own features and components to create, manage, and execute multi-agent systems. This means the agent configurations, multiagent configurations, and transition systems are handled solely within the 2APL system. In standalone mode, agents can communicate with each other directly, and the environment is simulated by the 2APL system itself.
- *Data Exchange:* Agents can import and export data from outside sources using the platform. Agents can use data that is not present in the current environment and therefore interact with the outside world.
- *Inter-Agent Communication:*The platform provides a message-passing mechanism that enables sending and receiving messages between each other and therefore communicating.
- *Interface Description:* The platform supports describing interfaces for agents.
- *GUI-Based Editor:* The platform offers a GUI-based editor for the 2APL programming language, which makes it easier to write 2APL programs.
- *Jade Interface:* If the execution mode is set to 'distributed' then the platform provides an interface to Jade. This allows communication between agents, which run on different machines, in the same network. Furthermore, programs written in 2APL can be easily translated to Jade script with the platform. It is even possible to playback 2APL programs in the Jade scripting console. For instance, an agent can use the Jade ImportAgent interface to import data from external sources by specifying a source and a method of receiving the data.
- *Visual Debugger:* Another functionality provided by the 2APL platform is a visual debugging tool. This allows monitoring the execution of multi-agent programs, halting their execution by means of breakpoints, inspecting the mental states of individual agents, and analyzing the interactions among individual agents.
- *Crash Reporter:* This enables programmers to locate and analyze issues that arise during the multi-agent system's operation. The system is interrupted when an agent crashes, and it is frequently difficult to identify the crash's primary cause. This is where the Crash Report helps.

For planning and building multi-agent systems, the 2APL platform offers a strong platform and developing tool. Due to its comprehensive collection of capabilities, it serves as the perfect basis for creating advanced multi-agent systems in 2APL.

5.2 FIPA-compliant Messages

The 2APL interpreter is built on the JADE platform. This interpreter is used in both execution modes ('stand-alone' and 'distributed'). The JADE (Java Agent Development Framework) platform is an open-source platform built for peer-to-peer agents. It is fully implemented in Java. The platform complies with the specifications of FIPA. As JADE complies with all FIPA specifications and the agent communication action of 2APL is interpreted by an interpreter, which is built on top of the JADE platform, the 2APL message are FIPA-compliant.

5.3 Ontologies in 2APL Messages

The message structure in 2APL does allow the addition of an ontology to the message. The communication action send is responsible for sending a message in 2APL. This action can have up to five different parameters: send(Receiver, Performative, Language, Ontology, Content). Receiver indicates the name of the agent, who receives this message. The Performative parameter is responsible for defining the speech act name, e.g. inform, request. Language defines the language used to express the content of the message. Ontology is the name of the ontology used to give a meaning to the symbols in the Content expression. The parameter Content is an expression representing the content of the actual message. It is often the case that agents assume a certain Language and Ontology such that it is not necessary to pass them as parameters of their communication actions. Therefore, the shorter version of the communicative action is: send(Receiver, Performative, Content)

6 Development Tools

There are three different execution monitoring tools that can be used in the 2APL platform: Overview, State Tracer, and Log. These tools can be accessed through the 'Debug' toolbar inside the 2APL platform and are activated by default.

- *Overview:* The respective tab consists of three panels called Beliefbase, Goalbase, and Planbase. The beliefs, goals, and plans of the current state of the selected agent are presented in the Beliefbase, Goalbase, and Planbase panels, respectively. The Figure 2 illustrates the use of the Overview tab which presents the beliefs, goals, and plans of agent harry from its initial state directly after loading the multi-agent program. Executing the multi-agent program will present updated information about the agent harry.

- *State Tracer:* The respective tab is a temporal version of the Overview tab and stores the beliefs, goals, and plans of all agents during execution. Because an execution generates a trace of agents' states (beliefs, goals, and plans), the tools and its corresponding tab are called state tracer. This tool allows a user to execute a multi-agent program for a while, pause the execution, and browse through the execution of each agent. With the buttons in the upper part of the tab one can navigate through the state trace. The user can select how many states (one, two, or three) to show on one screen and whether to show the beliefs, plans, goals, and log with the menu on the lower part of the tab. The Figure 3 illustrates the use of the state trace tool for monitoring the execution of the harry agent.

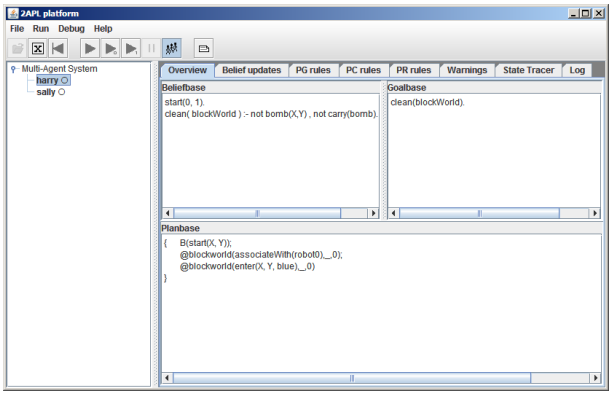

Fig. 2: Overview

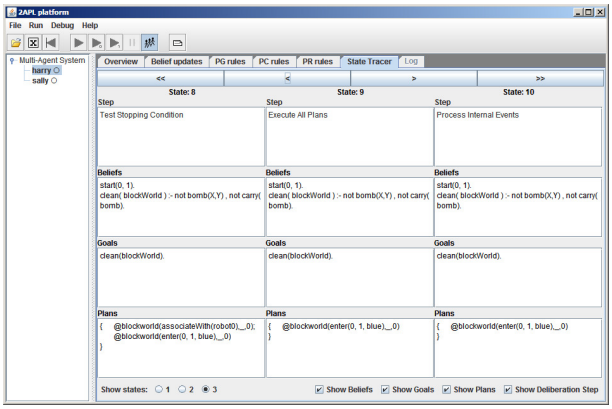

Fig. 3: State Tracer

This tab shows the execution of the multi-agent program at states 8, 9, and 10. It also shows that harry has performed an action in state 8 of the execution.

- *Log:* this tool presents information about the deliberation steps of individual agents. As illustrated in Figure 4, the Log tab indicates whether Harry has executed an action, processed an internal event, a message, or an external event. The user can browse through this window to see which deliberation steps have been performed. Each of the logging options can be disabled by deselecting it in the menu item 'Debugging'.

Furthermore, there is an Eclipse-based plugin, which adds the 2APL platform to Eclipse. The plugin runs on Eclipse Xtext. Eclipse Xtext is a framework for the development of programming languages and domain-specific languages. With Xtext the user defines his language using a powerful grammar language. As a result, the user gets a full infrastructure, including a parser, linker, typechecker, and compiler as well as editing support for Eclipse, any editor that supports the Language Server Protocol, and all web browsers. Through the plugin, a user can develop 2APL projects in three ways. He can create a new project based on existing multi-agent program files. He can also create a new project from scratch. Or he can load and edit an existing 2APL project. The 2APL IDE plugin provides syntax highlighting as well. The user can decide the highlighting style on his own, which is illustrated in Figure 5 below. The Figure 6 illustrates how 2APL files can be edited in Eclipse. There are other libraries available for developing multi-agent programs in 2APL. For example, the APLBenchmarker library. This is a generic

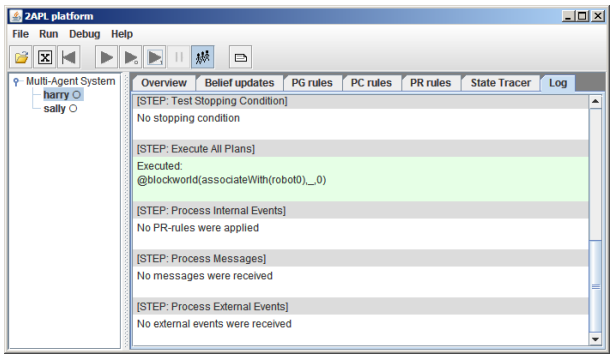

Fig. 4: Log

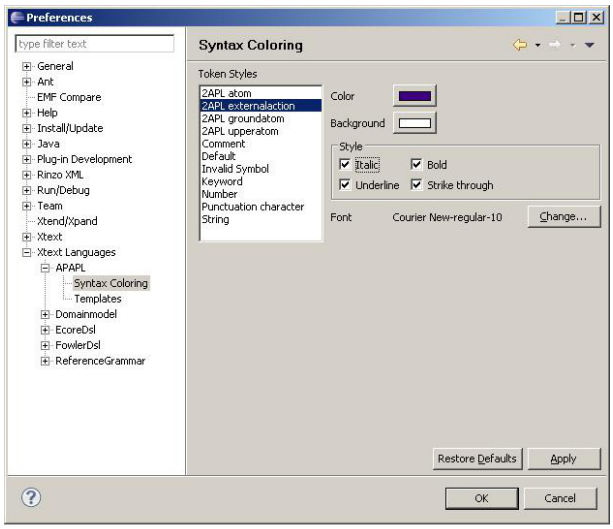

Fig. 5: Syntax Highlighting Options

benchmarking tool for 2APL. It comes with six operations that are being benchmarked. There is also the possibility to extend those operations individually. For each operation that is being benchmarked, the APLBenchmarker will print three results. Firstly, a count of the number of times that the operation has been performed. Secondly, the total execution time in milliseconds. And lastly, the average execution time for one single execution in milliseconds. Those results can be printed for each agent individually, or for all agents together.

7 Execution example

7.1 Hybrid 2APL

In order to run an example, first, Java version 9 or fewer needs to be installed (a pretty old version, as the most recent one is 19). Then, download the code from the 2APL repository [6]. Finally, run the APAPL file. After that, the window in Figure 7 will open. In order to run an example, click on "file", and then "open". Finally, select the desired *".mas"* file.

For this case, we will show the example that can be found on the repository "Harry & Sally".

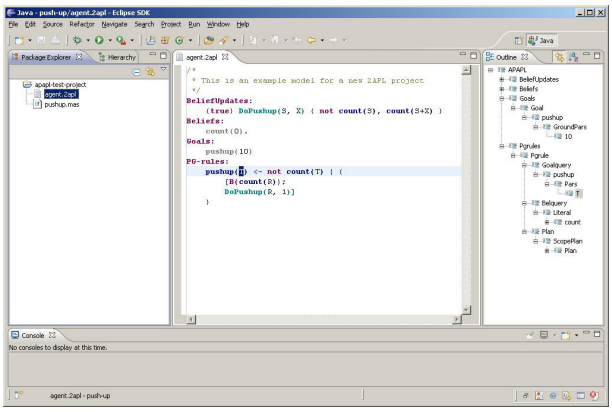

Fig. 6: Editing Files

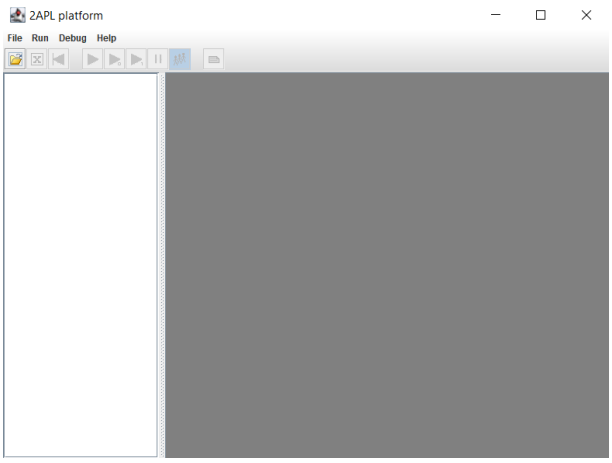

Fig. 7: Hybrid 2APL platform

The first file that we'll take a look at is the *"harrysally.mas"*.

```
<apaplmas>
  <environment name="blockworld" file="blockworld.jar">
    <parameter key="gridWidth" value="18"/>
    <parameter key="gridHeight" value="18"/>
    <parameter key="entities" value="2"/>
  </environment>
  <agent name="harry" file="harry.2apl"/>
  <agent name="sally" file="sally.2apl"/>
</apaplmas>
```

We can differentiate two parts, the first one is the environment, blockworld for our case. Then we find the agents' name. The blockworld environment is implemented as a Java program that creates a graphical user interface (GUI) for the environment and includes the necessary code to handle the agents and their interactions. The implementation of Blockworld in Java involves creating classes and methods that define the behaviour of the agents, the rules of the environment, and the graphical interface that displays the environment. As this is implemented in Java, it is out of the scope of this work, so no further details we'll be given. On the agent

name, two agents are declared, "harry", on the *"harry.2apl"* file, and "sally", on the *"sally.2apl"* file. Let's start by taking a look at the *"harry.2apl"* file. The first line:

```
include: person.2apl
```

includes the rule defined in the person.2apl file so that it can be used by harry. To be more precise, this is the included rule:

```
pcrules:
  goto( X, Y ) <- true |
  {
    @blockworld( sensePosition(), POS );
    B(POS = [A,B]);
    if (A > X) then
    { @blockworld( west(), L );
      goto( X, Y );
    }
    else if (A < X) then
    { @blockworld( east(), L );
      goto( X, Y );
    }
    else if (B > Y) then
    { @blockworld( north(), L );
      goto( X, Y );
    }
    else if (B < Y) then
    { @blockworld( south(), L );
      goto( X, Y );
    }
  }
```

This rule indicates that the execution of the abstract action *goto(X,Y)* causes this action to be replaced with the plan from the body of the rule. This plan implements a movement toward position (X,Y) by moving one square at a time. Note the use of recursion in this PC-rule. The rule starts with the definition of the *goto(X,Y)* abstract action, which takes two arguments: X and Y, representing the target position the agent wants to reach. The rule then uses the @blockworld construct to interact with the blockworld environment. The *sensePosition()* method is called to retrieve the current position of the agent, which is stored in the POS variable. The rule then uses the B construct to break down the position into its X and Y coordinates (A and B, respectively). The rule then uses a series of conditional statements (if-else) to check the current position of the agent and determine which direction the agent should move in order to reach the target position. For example, if the X coordinate of the current position is greater than the target X coordinate, the agent needs to move west, so the *west()* action is called using @blockworld, and the *goto(X,Y)* action is called again recursively with the same target position.

The next thing we find on Harry's file is:

```
beliefupdates:
  { bomb(X,Y) }          RemoveBomb(X,Y) { not bomb(X,Y) }
  { true }               AddBomb(X,Y)    { bomb(X,Y) }
  { carry( bomb ) }      Drop( )         { not carry( bomb ) }
  { not carry( bomb ) }  PickUp( )       { carry( bomb ) }
```

Belief update rules specify how an agent's beliefs should change in response to certain events or actions. For example, the first belief update rule specifies that if the agent believes there is a bomb at position (X,Y), then it should remove that belief (i.e., update its belief to *not bomb(X,Y)*) when it executes the *RemoveBomb(X,Y)* action. Similarly, the second belief update rule specifies that whenever the agent executes the *AddBomb(X,Y)* action, it should add the belief that there is a bomb at position (X,Y). The third one, specifies that if the agent believes it is currently carrying a bomb (*carry(bomb)*), then it should update its belief to indicate that it is no longer carrying the bomb (*not carry(bomb)*) when it executes the *Drop()* action. Finally, the last rule specifies that if the agent believes it is not currently carrying a bomb (*not carry(bomb)*), then it should update its belief to indicate that it is now carrying the bomb (*carry(bomb)*) when it executes the *PickUp()* action.

Then we come into the beliefs section:

```
beliefs:
  start(0,1).
  clean( blockWorld ) :- not bomb(X,Y) , not carry(bomb).
```

The first belief is that the agent's starting position is at coordinates (0,1). The second belief is that the block world environment is clean, but only if there are no bombs in the environment (*not bomb(X,Y)*) and the agent is not currently carrying a bomb (*not carry(bomb)*).

For the plans, we find:

```
plans:
  B(start(X,Y));
  @blockworld( enter( X, Y, blue ), _ );
```

The previous code fragment illustrates the implementation of the initial plan base of Harry, which first tests where he believes to enter, then enters the blockworld at the believed position.

Then we have the implementation of the initial goal base:

```
goals:
  clean( blockWorld ).
```

Harry wants to achieve a desirable situation in which the blockworld is clean.

The next code fragment we find is:

```
pgrules:
  clean( blockWorld ) <- bomb( X, Y ) |
  {
    goto( X, Y );
    @blockworld( pickup( ), _ );
        PickUp( );
        RemoveBomb( X, Y );
    goto( 0, 0 );
        @blockworld( drop( ), _ );
    Drop( );
  }
```

It indicates that a plan to achieve the goal *clean(blockworld)* can be generated if the agent believes there is a bomb at position (X,Y). Note that *goto(X, Y)* is an abstract action, the execution of which replaces a plan for going to position (X,Y). After performing this plan, Harry performs a pickup action in the blockworld, modifying his beliefs such that he now believes he carries a bomb, modifying his beliefs that there was a bomb at (X,Y), going to position (0,0), where the bomb should be dropped in a trash can perform the drop action in the blockworld, and finally modifying his beliefs that he does not carry a bomb anymore.

Then we find:

```
pcrules:
  message( sally, inform, La, On, bombAt( X, Y ) ) <- true |
  {
    if B( not bomb( A, B ) )
    {
      AddBomb( X, Y );
      adoptz( clean( blockWorld ) );
    }
    else
        {
          AddBomb( X, Y );
        }
  }
```

This rule indicates that if Harry receives a message from Sally informing him that there is a bomb at position (X,Y), then his beliefs will be updated with this new fact, and the goal to clean the blockworld is adopted, if he does not believe that there is already a bomb at a position (A,B). Otherwise, he updates his beliefs with the received information without adopting the goal. This is because Harry is assumed to this goal as long as he believes there is a bomb somewhere.

To finish up with Harry, we encounter:

```
prrules:
  @blockworld( pickup(), _ ); REST; <- true |
  {
    @blockworld( sensePosition(), POS );
    B(POS = [X,Y]);
    RemoveBomb( X, Y );
  }
```

This rule is used for the situation in which the execution of a plan that starts with the external action *@blockworld(pickup(), _)*; fails. Such an action fails in case there is no bomb to be picked up (e.g. when it is removed by another agent). The rule states that the plan should be replaced by another plan consisting of an external action to sense its current position, after which it removes the bomb from its current position. Note that in this case, the rest of the original plan denoted by REST is dropped, as it is not used anymore within the rule.

This was the last part of the *"harry.2apl"*. Let's now take a look at *"sally.2apl"*. In this case, its initial goals are:

```
goals:
```

```
  search( blockWorld ).
```

Which specifies that the goal of sally is to search for bombs in the environment.

The only rule that this file has is:

```
pgrules:
  search( blockWorld ) <- true |
  {
    B(is( X, int( random( 15 ) ) ));
    B(is( Y, int( random( 15 ) ) ));
    goto( X, Y );
    @blockworld( senseBombs(), BOMBS );
    if B( BOMBS = [[default,X1,Y1]|REST] ) then
    {
      send( harry, inform, bombAt( X1, Y1 ) );
    }
  }
```

This plan generation rule describes how an agent can randomly move through the block world environment and inform other agents if it detects the presence of a bomb. This rule can be used to generate plans for an agent that is responsible for detecting bombs in the environment, as is the case for Sally.

8 Different Versions (OO2APL and Net2APL)

In this section, we will study other versions of APL, OO2APL, and Net2APL. First, we look into the intent and motivation for these versions. Then we study the abstractions provided by APL in an object-oriented pattern and how these participants collaborate with each other. Finally, an execution example will be shown to illustrate how this language works.

8.1 Intent and Motivation

Based on the idea that Agents as a concept can make a valuable contribution to real-world applications, the 2APL programming language was refurbished as an Object Oriented Design Pattern and subsequently reimplemented in JAVA as OO2APL, which is object-oriented 2APL. Net2APL is a continuation of the OO2APL framework, which can be distributed over multiple physical systems.

The intent of the OO agent pattern is to separate the business logic of decision-making and processing of triggers from the rest of an application and thus promote the modular development of autonomous behavior. It allows developers to concentrate all the decision-making in a few classes that expose exactly what can be observed, what actions can be made, and how the connection between observations and action is realized. These classes can be stored and extended from a library to facilitate reuse.

8.2 Elements in OO2APL

Now let's explore the various elements that the OOAPL language offers for defining the behavior of agents:

- **Trigger.** An object that represents an (internal and external) event, message or goal to be processed.
- **Goal.** Persistent trigger that represents a goal of the agent.
- **Plan.** Specifies the business logic for processing a trigger.
- **PlanScheme.** Specifies when a plan is applicable.
- **Context.** Exposes all required information for determining whether a goal is achieved and the execution of plans.
- **Messenger.** Allows the agent to send messages to other agents.
- **DeliberationStep.** Selects and/or executes relevant plans.
- **DeliberationRunnable.** Contains a run method that executes a sequence of deliberation steps (called deliberation cycle).
- **RuntimeConfiguration.** This class is the main data container for a single agent.
- **TriggerListener.** Exposes the functionality to add to the agent triggers (messages, events, percepts, etc.).
- **KillSwitch.** Contains a method to cause the agent to stop executing.

The structure of the OO Agent Pattern in OO2APL is shown in Figure 8.

8.3 Operational Semantics

The runtime configuration is the core component of an agent. It stores and returns objects to collaborate with other components during the execution time. Most collaborations occur when an agent receives a trigger.
Trigger listeners can add triggers to the runtime configuration. This will cause the agent to call upon its deliberation runnable to execute a deliberation cycle. A deliberation cycle is a sequence of deliberation steps.
Each deliberation step contains an execute method that given a runtime configuration progresses the agent. In our library, we instantiated two types of deliberation steps that we consider to be basic progressions of an agent.

The first is depicted as step 'a', and progresses the runtime configuration by obtaining the plan schemes, current triggers, and context to instantiate plans schemes, and storing the potentially returned instantiated plans in the runtime configuration. A second step 'b' obtains the instantiated plans from the runtime configuration and executes them with the configuration's context and messenger (for communicating with other agents).
In our library, we chose to put the agent to sleep if it contains no more active plans or triggers, otherwise, it will execute the cycle again. A kill switch can halt an agent. The execution method of a plan may retrieve the context and messenger of the agent and adopt new triggers, including goals.

We note that the depicted classes in Figure 8 serve to specify the agent and need to collaborate with a surrounding system in order to gain execution time. In our library, we opted to enrich a

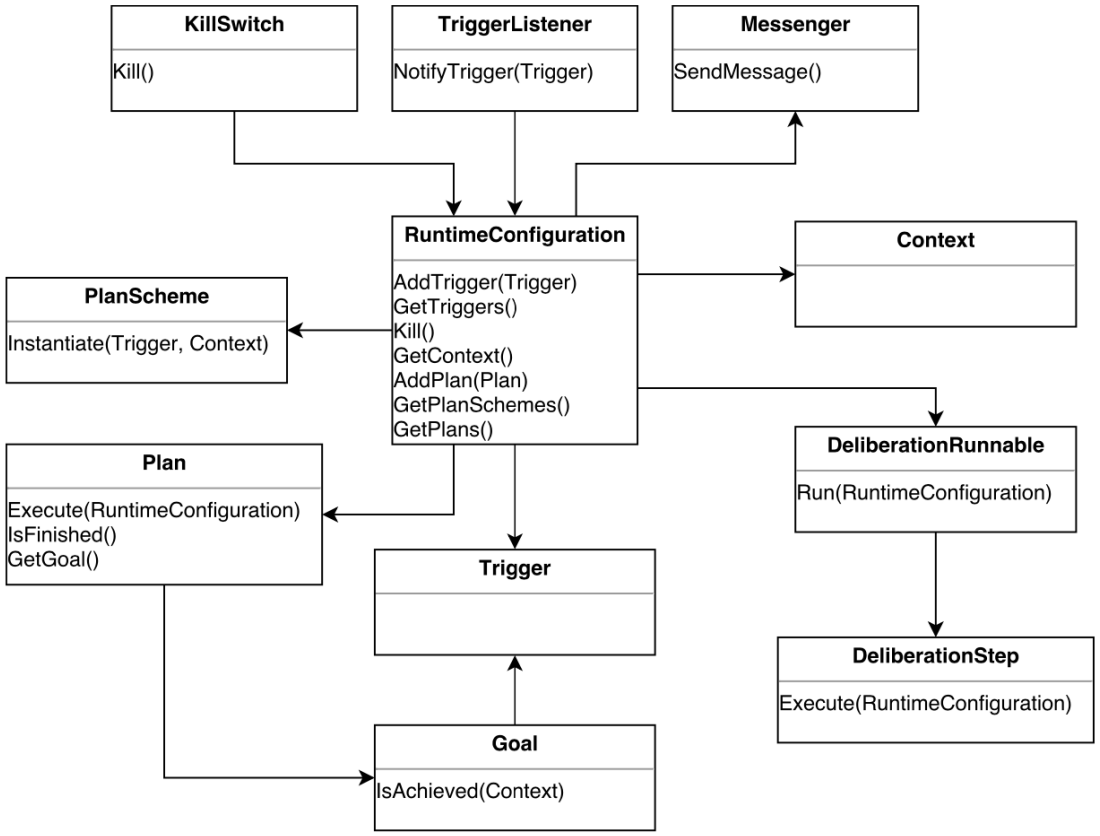

Fig. 8: Structure of the elements that form an Agent in OO2APL

standard executor service that maintains a thread pool for executing agents, the kills switch, a messenger infrastructure, and agent initialization factories.
This collaboration process is illustrated in Figure 9.

8.3.1 Semantic elements

There is the necessity to state in detail the most important aspects that the agent can execute and perform or receive during the deliberation cycle. Furthermore, the elements will be explained by answering the questions provided by the assignment.

- ***Is there any Belief revision?*** *Yes.* Belief, knowledge about the world, is in the case of NetAPL Context, which can be retrieved for inspection, writes and updates during the execution of a plan via the 'getContext' method of PlanToAgentInterface.
- ***Is there any Goal/Intention revision?*** *Yes.* Since triggering message can be sent and received during execution time, it could be the new goal for Agent.
- ***Is there any Plan/Task revision?*** *Yes.*The plan.builtin sub-package hosts some convenience-classes, so the implementor won't need to reprogram these types of Plans each time they are needed. Some of them (like FunctionalPlanScheme and SubPlanInterface) are involved in the current recommended way for implementors to implement plans. Self.rescheduler is a class also for Agent to reschedule the execution of their own plan.

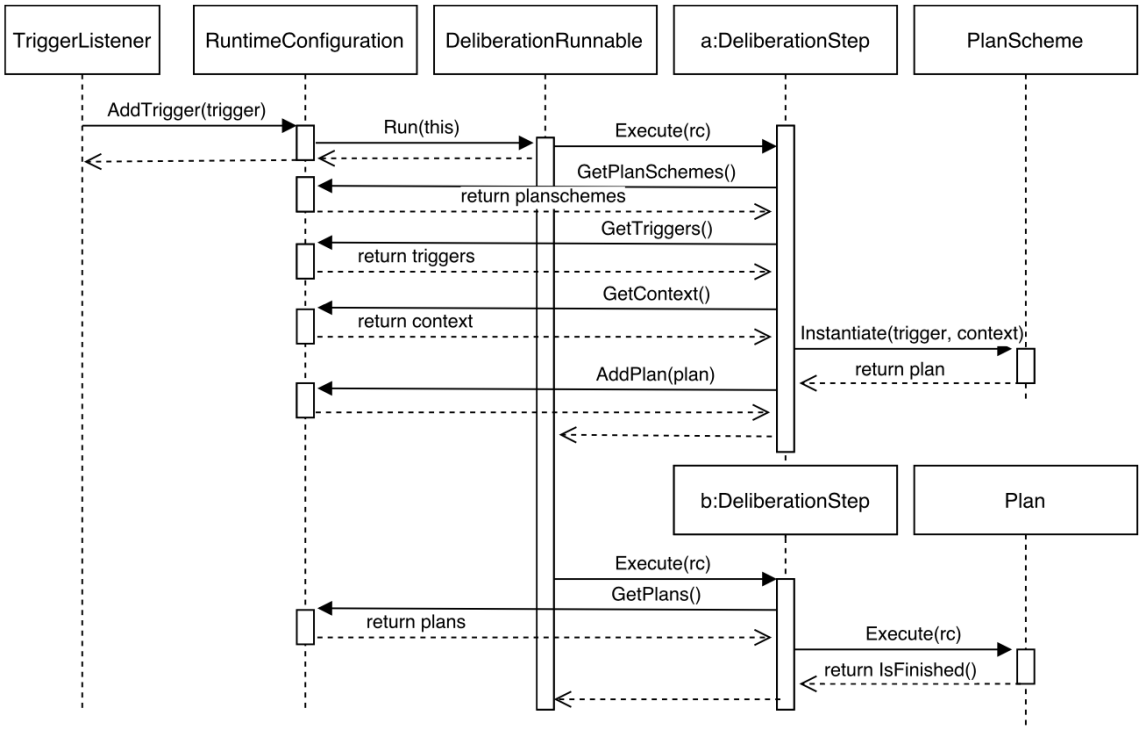

Fig. 9: The core interactions of the agent components.

- ***Is there any way to make agents react to sudden changes in the environment?*** *Yes, the events* can be utilized to pass information from the environment to agents. The agent can react to sudden changes and special events in the environment already specified by the programmer. When an event is activated, the method *notifyEvent()* will be called from the environment's constructor and sent to the agents in the list.
- ***Is there any special events' handling?*** *Yes.* There are ways to handle various kinds of exceptions, like AgentCreationFailedException, MessageReceiverNotFoundException, DeliberationStepException, PlanExecutionError...
- ***Is there any Meta-Level reasoning?*** Not found.

8.4 Execution Example

In order to execute this example, Java 11 or higher is needed. The first step consists of downloading 2APL code and installing it as a Maven repository on your computer. Then, we can download the example and run in it in our IDE. Once we execute, we see the same environment we saw before, blockworld.

For this case, we start by analyzing the *Main.java* file. Firstly, we define the position where both agents will appear:

```
private static final Point HARRY_START_POSITION = new Point(0,0);
private static final Point SALLY_START_POSITION = new Point(8, 8);
```

Then, we create the agents by the following lines:

```
Main.createAgent(
        getHarryArguments(startingPosition),
        startingPosition,
        AgentColor.BLUE,
        Main.nHarries > 1 ? String.format("Harry-%d", i) : "Harry"
);

Main.createAgent(
        getSallyArguments(startingPosition),
        startingPosition,
        AgentColor.RED,
        Main.nSallies > 1 ? String.format("Sally-%d", i) : "Sally",
        new SearchGridWorldGoal()
);
```

As we can see, Sally's creation has an extra line, "new SearchGridWorldGoal()". That will be the plan the agent will adopt.

This is how the goal is implemented:

```
public class SearchGridWorldGoal extends Goal {
    @Override
    public boolean isAchieved(AgentContextInterface
       agentContextInterface) {
        return false;
    }

    public String toString() {
        return "search( GridWorld )";
    }
}
```

As we can see, it will always return false, as we want that Sally constantly looks for bombs and bins. As long as this is Sally's goal (which should be always), she will execute the following plan:

```
@Override
public void execute(PlanToAgentInterface planToAgentInterface) throws
   PlanExecutionError {
    try {
        this.context = planToAgentInterface.getContext(SallyContext.
           class);
        this.planToAgentInterface = planToAgentInterface;
        Environment env = context.getEnvironment();

        try {
            env.senseBombs(planToAgentInterface.getAgentID()).forEach(
               this::informBomb);
            env.senseTraps(planToAgentInterface.getAgentID()).forEach(
               this::informTrap);
        } catch (ExternalActionFailedException e) {
            System.err.println(e.getMessage());
```

```
        }

        Point randomPoint = new Point(
                (int) Math.round(Math.random() * (env.getWidth() - 1)),
                (int) Math.round(Math.random() * (env.getHeight() - 1))
        );
        GoToGoal newGoToGoal = new GoToGoal(randomPoint);

        // Make sure no new GoToGoal is adopted if one is already in the
            goal base. Otherwise, Sally would try to
        // move to various random points at the same time, thus getting
           nowhere
        GoToGoal.adoptIfNoOther(newGoToGoal, planToAgentInterface);
    } catch(Exception e) {
        e.printStackTrace();
    }

    // Unlike other plans, this plan is never marked as finished, since
       it should continue executing indefinitely
}
```

As we can see, we sense for bombs and traps, and in case we find one we execute the *informBomb* or *informTrap* that can be seen below. After that, we select another random point to move to.

```
private void informBomb(Point p) {
    if(this.context.addInformedBomb(p))
        inform("bombAt", p);
}

private void informTrap(Point p) {
    if(this.context.addInformedTrap(p))
        inform("trapAt", p);
}
```

Then, in order to send the messages we have the inform method, where we create a message, set the parameters and send it:

```
private void inform(String directive, Point p) {
    ACLMessage newMessage = new ACLMessage(Performative.INFORM);
    newMessage.setSender(this.planToAgentInterface.getAgentID());
    newMessage.setContent(String.format("%s(%d,%d)",directive, p.x, p.y)
       );
    newMessage.setReceivers(this.context.getOtherAgents());
    for (AgentID agentID : this.context.getOtherAgents()) {
        this.planToAgentInterface.sendMessage(agentID, newMessage);
}
```

Finally, in order to match the goal with the plan, a "PlanScheme" is needed:

```
public class SallyGoalPlanScheme implements PlanScheme {
    @Override
    public Plan instantiate(Trigger trigger, AgentContextInterface
       agentContextInterface) {
        Plan plan = Plan.UNINSTANTIATED;
```

```
        if(trigger instanceof SearchGridWorldGoal) {
            plan = new SearchGridWorldPlan();
        }

        if(trigger != null && plan != Plan.UNINSTANTIATED)
            plan.setPlanGoal(trigger);

        return plan;
    }
}
```

We also have one more plan to handle messages, and their corresponding "PlanScheme".

```
public class SallyMessagePlanScheme implements PlanScheme {

    @Override
    public Plan instantiate(Trigger trigger, AgentContextInterface
       agentContextInterface) {
        Plan plan = Plan.UNINSTANTIATED;

        if(trigger instanceof ACLMessage) {
            plan = new SallyProcessMessagePlan((ACLMessage) trigger);
        }

        return plan;
    }
}
```

As we can see, if we receive an ACL Message, we take as a plan to process it.

```
@Override
public void execute(PlanToAgentInterface planToAgentInterface) throws
   PlanExecutionError {
    this.context = planToAgentInterface.getContext(SallyContext.class);

    if(this.message.getPerformative().equals(Performative.REQUEST)) {
        if(this.message.getContent().equalsIgnoreCase("traps"))
            // Harry informed Sally it does not have any trap locations
            this.context.clearTraps();
        else if (this.message.getContent().equalsIgnoreCase("bombs"))
            // Harry informed Sally it does not know any other bomb
               locations
            this.context.clearBombs();
    } else {
        handleNoBombPattern();
        handleNoTrapPattern();
    }

    setFinished(true);
}
```

In case Harry informed Sally it does not have any trap locations, or it does not know any other bomb location, Sally will remove the bomb or the trap from its belief base.

Finally, let's show the structure of the agent. We find different structures where we can store information, for instance:

```
private final Set<AgentID> otherAgents = new HashSet<>();
```

Where we can store the other agents that we will inform? Then, in order to get any of these attributes, a function like this one is used:

```
public Set<AgentID> getOtherAgents() {
    synchronized (this.otherAgents) {
        return new HashSet<>(this.otherAgents);
    }
}
```

9 Conclusion

In conclusion, 2APL (A Practical Agent Programming Language) offers a comprehensive framework for developing multi-agent systems. With its foundation in the BDI model of agency, agents in 2APL can reason about beliefs, desires, and intentions to achieve their goals. The language provides elements such as beliefs, desires, intentions, messages, plans, and norms to structure the behavior of agents and facilitate communication among them. The operational semantics of 2APL define how agents execute language constructs, ensuring a well-defined and coherent behavior. The formal semantics provide a mathematical foundation for precisely interpreting the language constructs, enabling formal reasoning about agent behavior.

2APL agents interact with the environment through perception and action, allowing them to gather information and affect the world. The language can be connected to external libraries and tools, providing additional functionalities for communication, perception, and action. One of the distinguishing features of 2APL is its support for programming with norms, enabling the specification and enforcement of expected agent behavior. This feature contributes to the regulation and coordination of agents in multi-agent systems.

Overall, 2APL serves as a flexible and powerful platform for developing intelligent multi-agent systems. Its BDI-based architecture, support for programming with norms, and integration with the environment make it a popular choice among researchers and practitioners in the field. The language provides a solid foundation for designing and implementing complex agent interactions, fostering the development of innovative solutions in the realm of multi-agent systems.

References

[1] A. Bracciali et al. "The KGP model of Agency for global computing: Computational model and prototype implementation". In: *Global Computing* (2005), pp. 340–367. DOI: 10.1007/978-3-540-31794-4_18.

[2] Mehdi Dastani. "2APL: A practical agent programming language". In: *Autonomous Agents and Multi-Agent Systems* 16.3 (2008), pp. 214–248. DOI: 10.1007/s10458-008-9036-y.

[3] Mehdi Dastani. "Design Patterns for Multi-Agent Programming". In: *International Journal of Agent-Oriented Software Engineering* (2016), pp. 167–202.

[4] *Transition system*. Apr. 2023. URL: https://en.wikipedia.org/wiki/Transition_system%5C#%5C~:%5Ctext=In%5C%20theoretical%5C%20computer%5C%20science%5C%2C%5C%20a,potential%5C%20behavior%5C%20of%5C%20discrete%5C%20systems..

[5] *2APL Publications*. URL: http://www2.projects.science.uu.nl/Net2APL/publications.html.

[6] *Hybrid 2APL repository*. URL: https://bitbucket.org/goldenagents/2apl.git.