Practical Assignment 1: 2APL A practical Agent Programming Language

MultiAgent System Design

Master in Artificial Intelligence

Students:
- Bonifacio Marco Francomano
- Marc Monfort
- Michelangelo Stasi
- Mario Rosas

Professors:
- Javier Vazquez Salceda

Academic Year: 2023-24

April 8th, 2024

Contents

2APL A practical Agent Programming Language

Disclaimer

There exist two versions of 2APL language: the Java-based version (Net2APL) and the Hybrid version. This report focuses primarily on the Hybrid 2APL version, with a brief description of the Java-based version for context.

Java-based 2APL (Net2APL)

Net2APL is a Java-based BDI Agent Framework library, continuing from the OO2APL framework. Key features include:

- **Origins**: Built upon the OO2APL framework.
- **Design Pattern**: Utilizes the Object-Oriented Agent Pattern as described in "*Design Patterns for Multi-Agent programming*" by Dastani & Testerink (2016).
- **BDI Approach**: Emphasizes the Believes, "*Desires, Intentions*" (BDI) paradigm for agent behaviors.
- **Core Components**:
  - **Plan Class**: Represents agent intentions.
  - **Goal Class**: Directly signifies agent desires.
  - **Trigger Class**: Superclass of Goal, encapsulating agent objectives.
- **Distribution**: Supports agent distribution across multiple physical systems.

1 | Declarative Abstractions: Beliefs and Goals

In the design of multiagent systems, the language employs a combination of declarative and imperative abstractions to enable seamless agent interactions and decision-making capabilities.

Beliefs and goals provide the foundational declarative constructs, in order to support the reasoning process of the agents. On the other hand, plans, events, and exceptions operate within the imperative paradigm, facilitating actionable tasks, communication, and error management.

1.1 | Beliefs

Beliefs are fundamental to an agent's perception and understanding of its environment. The belief base of an agent is a repository of information that encompasses both the agent's self-awareness and its understanding of the external world.

Example of Beliefs in Prolog Format

```
Beliefs:
pos(1,1).
hasGold(0).
```

In the provided example:

- `pos(1,1)` denotes the position of the agent.
- `hasGold(0)` indicates that the agent currently possesses no gold.

These beliefs are expressed as Prolog facts or rules, with the critical characteristic that all facts are ground, providing a definitive and unambiguous representation of the agent's state.

1.2 | Goals

Goals drive an agent's actions and decision-making processes. A 2APL agent's goals are implemented through its goal base, which comprises a collection of formulas.

Example of Goals in Prolog Format

```
Goals:
hasGold(5) and clean(blockworld), hasGold(10).
```

In this example:

- `hasGold(5) and clean(blockworld)` represents a conjunctive goal where the agent aims to possess 5 units of gold and clean a "blockworld".
- `hasGold(10)` signifies an additional goal of acquiring 10 units of gold.

Each goal expression is a conjunction of ground atoms, providing clear directives to the agent's decision-making process.

2 | Imperative Abstractions: Plans and Actions

While beliefs and goals operate in a declarative paradigm, the implementation of plans and interactions with the external environment is rooted in an imperative programming style.

2.1 | Plans

Plans serve as actionable blueprints, guiding the agent through various tasks and interactions. These plans are constructed using operators such as conditional choice, iteration, and sequence.

- **Sequence Operator (;)**: This binary operator links two plans, designating that the first plan must be completed before commencing the second.
- **Conditional Choice Operator (if then else)**: Enables branching within plans based on conditions evaluated against the agent's belief and goal bases.
- **Conditional Iteration Operator (while do)**: Facilitates the repetition of a plan as long as a specific condition remains true.
- **Non-interleaving Operator ([ ])**: Marks a plan as atomic, ensuring its execution without interruption. This guarantees that certain critical actions within a plan are executed in an uninterrupted sequence, essential for tasks requiring consistency or atomicity.

Example of Goals in Prolog Format

```
Plans:
[@blockworld(enter(5,5,red),L); ChgPos(5,5)],
send(admin,request,register(me)).
```

In the provided examples:

- `[@blockworld(enter(5,5,red),L); ChgPos(5,5)]` illustrates a sequence of two actions where the agent enters a 'blockworld' at position (5,5) with a red attribute and then changes its position to (5,5).
- `send(admin,request,register(me))` depicts a singular action plan where the agent sends a registration request to an administrator.

2.2 | Events and Exceptions

Events and exceptions play vital roles in facilitating communication and managing unforeseen circumstances within the multiagent system.

- **Events**: Events are conduits for the flow of information from the environment to the agents.
- **Exceptions**: In the context of 2APL, exceptions are leveraged to signal plan execution failures, triggering plan repair rules.

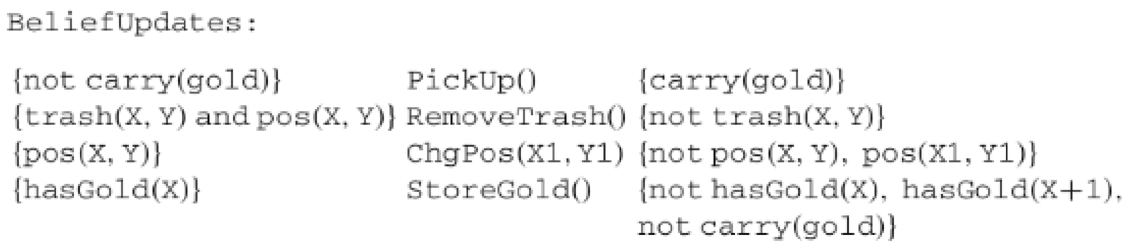
```
Events:
notifyEvent(AF event, String... agents).
Exceptions:
There is no explicit 2APL construct for exceptions. Formal semantic entities are
    employed to manage exceptional scenarios effectively.
```

3 | Basic Actions in Multiagent System Design

In the field of multiagent systems, basic actions are needed by the agents to interact and adapt within their environments. Below is a list of the primary types of basic actions:

3.1 | Belief Update Actions

Belief Update Actions enable the agents to modify and update their belief bases dynamically, ensuring alignment with evolving environmental conditions and new information. The structure of the action is Precondition, action, effect.



```
BeliefUpdates:

{not carry(gold)}           PickUp()       {carry(gold)}
{trash(X, Y) and pos(X, Y)} RemoveTrash()  {not trash(X, Y)}
{pos(X, Y)}                 ChgPos(X1, Y1) {not pos(X, Y), pos(X1, Y1)}
{hasGold(X)}                StoreGold()    {not hasGold(X), hasGold(X+1),
                                            not carry(gold)}
```

Figure 1.1: BeliefUpdates

For instance, in the first example, the precondition of the action PickUp() is that the agent doesn't already carry the gold in order to perform the action and the effect is to carry the gold after.

3.2 | Communication Actions

Communication actions enable agents to exchange messages, facilitating coordination, information sharing, and negotiation among agents in a MAS.

Three-Parameter Version

- `send(Receiver, Performative, Content)`: This simplified version is used when agents share a common language and ontology, focusing on the recipient, the type of message being sent (performative), and the message content.

Five-Parameter Version

- `send(Receiver, Performative, Language, Ontology, Content)`: This version includes specifications of the language and ontology used in the message, in addition to the recipient, performative, and content.

3.3 | External Actions

External Actions enable agents to perform tasks and operations within their external environments, ranging from navigation and exploration to manipulation and interaction with objects and entities.

Agents can learn about the consequences of their external actions through several means:

- **Sense Actions**: Specific external actions designed to gather information about the state of the environment.
- **Events**: Notifications generated by the environment in response to changes.
- **Return Parameters**: Data returned by the environment following the execution of an external action

Expression Format: An external action is expressed in the form `@env(ActionName, Return)`, where:

- `env` is the identifier for the external environment
- `ActionName` is the specific action to be executed
- `Return`is a list of values returned by the action, which can be empty or providing feedback.

3.4 | Abstract Actions

Abstract Actions involve the planning and execution of high-level, goal-oriented actions, allowing agents to formulate strategies, optimize decision-making processes, and achieve complex objectives efficiently.

3.5 | Test Actions

Test actions allow an agent to verify specific beliefs and goals before proceeding with certain parts of its plan. These actions are crucial for conditional decision-making within an agent's operational logic.

We have two types of test actions:

- **Belief Query Expression ($B(\phi)$)**: Queries the belief base using a formula ϕ made up of literals combined through conjunction or disjunction.
- **Goal Query Expression ($G(\phi)$)**: Similar to belief queries but targets the goal base, checking for goals that satisfy the given formula ϕ.

3.6 | Goal Dynamics Actions

Goal Dynamics Actions facilitate the management, prioritization, and adaptation of goals, empowering agents to adjust their objectives in response to changing environmental conditions, constraints, and priorities.

Types:

- Adopt Goal Actions:
  - `adopta(`ϕ`)`: Adds a goal at the beginning of the goal base.
  - `adoptz(`ϕ`)`: Adds a goal at the end of the goal base.
- Drop Goal Actions:
  - `dropgoal(`ϕ`)`: Removes a specific goal from the goal base.
  - `dropsubgoals(`ϕ`)`: Removes all goals that are logical subgoals of ϕ
  - `dropsupergoals(`ϕ`)`: Eliminates all goals for which ϕ is a logical subgoal

4 | Practical Reasoning Rules

The 2APL programming language provides constructs to implement practical reasoning rules that can be used to implement the generation of plans during an agent's execution.

Planning Goal Rules (PG-rules)

PG-rules enable an agent to generate plans based on its current goals and beliefs.

Format

- `[goal query] ← [belief query] | [plan].`

The head (goal query) is optional, allowing plans to be generated based solely on belief conditions.

Example

- **Scenario:** Generating a plan for an agent to move to a location (X2,Y2) from (X1,Y1) to remove trash, given the goal to clean a specific space (R) and the belief about its current position and the trash's location.
- **Rule:** `clean(R) ← pos(X1, Y1) and trash(X2, Y2) | {goTo(X1, Y1, X2, Y2) ; RemoveTrash()}.`

Procedure Call Rules (PC-rules)

Enable agents to respond to external events, messages from other agents, or the completion of abstract actions by generating appropriate plans.

Format

- `[atom] ← [belief query] | [plan].`

Head Examples

- **Message Reception:** A rule specifies generating a plan for getting and storing gold upon receiving a message about its location, provided the agent does not currently carry gold.
- **Event Handling:** Similarly, a plan is generated in response to an environmental event indicating gold's presence, with the same condition regarding the agent not carrying gold.

- **Abstract Action Execution:** Describes the sequence of actions for the abstract action `getAndStoreGold`, including moving to the gold's location, picking it up, moving to the depot, and storing it. This sequence is tightly controlled to prevent undesirable effects from other actions' interleaving.

Plan Repair Rules (PR-rules)

Plan repair rules are designed to respond to the failure of plan execution. These rules enable agents to replace a failed plan with a new, potentially more successful plan.

Application Conditions

A PR-rule is applied under three conditions:

- A plan's execution fails.
- The failed plan matches the abstract plan expression in the rule's head.
- The agent's beliefs satisfy the rule's belief condition.

Criteria for Failure

The execution of a plan is considered **failed if the execution of its first action fails**.

Example

- **Scenario:** A plan begins with two eastward movements and fails.
- **Repair Strategy:** The rule suggests replacing the failed plan with a new sequence: move north, then east twice, and finally south.

5 | Execution of Multi-agent Systems in 2APL

The execution of MAS is governed by a transition system that defines how agents and environments evolve over time through a series of computational steps or transitions.

Concepts:

- **Computation Run:** Sequence (finite or infinite) of configurations representing the system's state over time.
- **Execution Defined:** Set of all possible computation runs starting from an initial configuration.

- **Interpreter's Role:** Navigates one of these computation runs, executing agents and their interactions with the environment in parallel.
- **Deliberation Process:** The execution of each agent is guided by a deliberation process, a cyclic procedure where agents continuously assess their surroundings, update their beliefs and goals, and select actions based on practical reasoning rules.

Agent Deliberation Cycle

Steps:

1. **Initial Step:** The cycle begins by applying all applicable Planning Goal Rules (PG-rules), with each rule applied once per cycle. These rules generate plans based on the agent's goals and beliefs.
2. **Plan Execution:** Only the first actions of all plans are executed in a cycle. This ensures fairness in the execution of plans, allowing each plan an opportunity to progress.
3. **Event Processing:** The cycle includes steps for processing different types of events:
   - **External Events:** Handled by applying the first applicable Procedure Call Rule (PC-rule) with an event head.
   - **Internal Events:** Indicating failed plans, which are managed by applying the first applicable Plan Repair Rule (PR-rule).
   - **Messages:** Received from other agents, processed by applying the first applicable PC-rule with a message head.

Decision to proceed with a new deliberation cycle: If no rules could be applied, no plans were executed, and no events were processed, initiating a new cycle might be redundant unless new events or messages arrive (meta-reasoning).

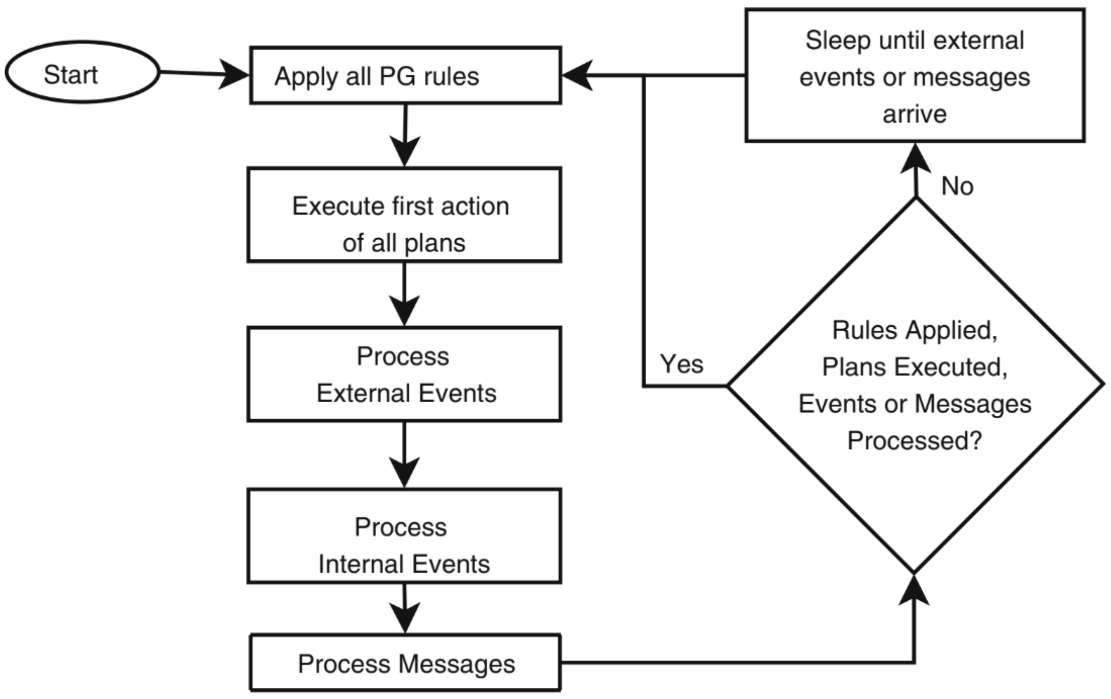

Start
Apply all PG rules
Execute first action of all plans
Process External Events
Process Internal Events
Process Messages
Rules Applied, Plans Executed, Events or Messages Processed?
Yes
No
Sleep until external events or messages arrive

Figure 1.2: Agent Deliberation Cycle

Deliberation Cycle Properties

1. **Plan Failure and Repair:** If a plan fails during execution, it will either be repaired within the same cycle or re-executed in the next cycle. This ensures resilience in agent operation, allowing for the correction or reattempt of failed actions.

2. **Adaptability to Changes:** The deliberation process is designed to adapt to changes within the agent's belief and goal bases or the environment. For example, if the execution of an action fails due to current conditions, modifications in the agent's internal state or environment might enable successful execution in subsequent cycles.

6 | Formal Semantics in Multiagent System Design

Introduction to Formal Semantics

To understand the theoretical model of the 2APL language and its operational dynamics within multiagent systems, first of all we must describe what a transition system is. A transition system serves as a foundational framework, delineating the rules that govern the evolution of configurations within the multiagent environment.

6.1 | Transition System: An Overview

A transition system provides rules to derive transitions from one configuration to another. The configuration of an agent, denoted as Ai, comprises a comprehensive set of components that collectively define the agent's state and operational context.

Formal Definition of an Agent Configuration:

$$Ai =< i, \sigma i, \gamma i, \Pi i, \theta i, \xi i >$$

Where:

- i represents the agent's identifier, providing a unique identifier for distinct agent entities within the system.
- σi denotes a set of belief expressions
- γi is a list encapsulating the agent's goal expressions
- Πi represents a set of plan entries
- σi signifies a ground substitution mechanism, binding domain variables to specific ground terms
- $\xi i =< E, I, M >$ constitutes the agent's event base, capturing the agent's interactions, notifications, and procedural triggers within the multiagent environment.

6.2 | Transition Rule for Belief Update Action (T)

The belief update action, denoted as T, is a pivotal component within the formal semantics of the 2APL language, governing the agent's belief integration and goal management processes. Below is a succinct exploration of the transition rule associated with the belief update action T.

$T(action, \theta i, \sigma i, \gamma i)$

- $action$: Actionable task with substitution θi
- θi: Ground substitution binding variables to terms

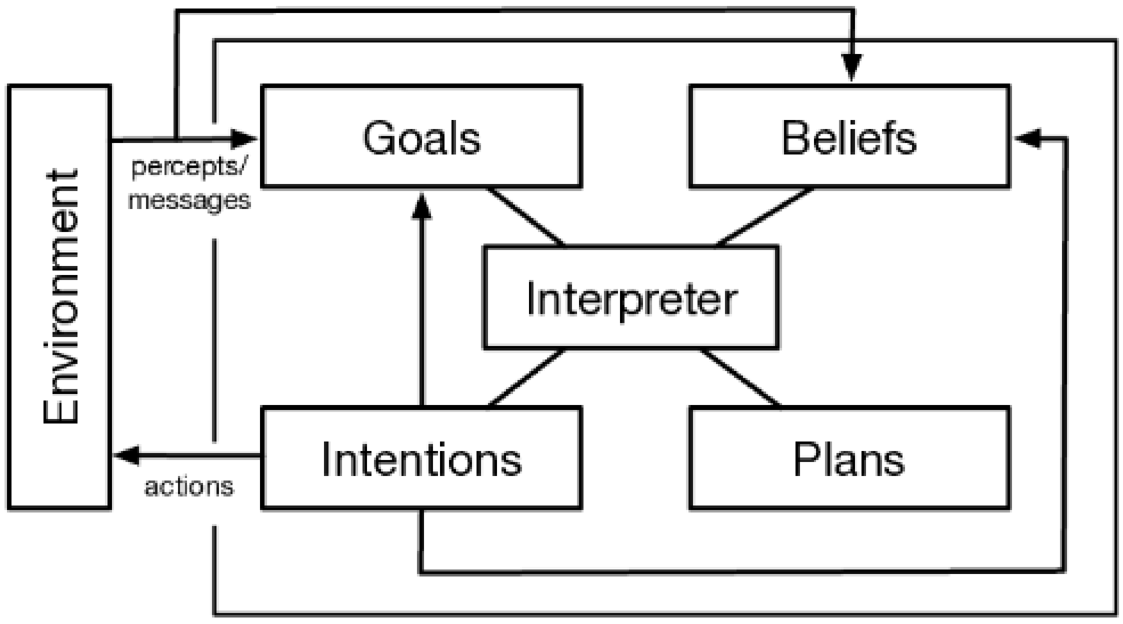


Figure 1.3: Transition System

$$\frac{T(\alpha\theta, \sigma) = \sigma'}{\langle \iota, \sigma, \gamma, \{(\alpha, id)\}, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma', \gamma', \{\}, \theta, \xi \rangle}$$

Where $\gamma' = \gamma - \{\phi \in \gamma \mid \sigma' \models \phi\}$

Figure 1.4: Formal Representation of Transition Rule T

- σi: Current belief set
- γi: Current goal base

Operational Dynamics of Transition Rule T:

1. **Precondition Verification**: T evaluates the action's precondition against σi. If satisfied, the action proceeds; otherwise, it remains pending.
2. **Belief and Goal Update**: Successful actions update σi with new belief expressions and remove newly believed goals from γi.
3. **Exception Handling**: Failed precondition verification's lead to plan failure exceptions, keeping the action in the plan.

$$\frac{T(\alpha\theta, \sigma) = \bot \ \& \ \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \langle E, I \cup \{id\}, M\rangle\rangle}$$

Figure 1.5: Operational Dynamics of *T*

Implementation Consistency: The transition rule T's mechanisms are consistent with 2APL's language functionalities, ensuring uniformity in agent interactions and belief management across multiagent systems.

2

Modules in 2APL

In 2APL modularization is considered as a mechanism to structure an individual agent's program in different modules. Each module encapsulates cognitive components such as beliefs, goals, plans and reasoning rules. A 2APL module is used to specify a specific functionality that can be used to handle specific situations or tasks. It is a data structure that specifies an agent's cognitive state.

A 2APL multi-agent program consists of a set of module specifications each with a unique name. Initially, a subset of these module specifications is identified as specifying the initial state of individual agents. The execution of a multi-agent program is then the instantiation of this subset of module specifications followed by performing a deliberation process on each module instance. In this way, an instance of a module specification forms the initial state of an individual agent.

Note that a module instance specifies the cognitive state of an agent while the agent itself is the deliberation process working on the cognitive state. When using modules in 2APL, different operations on modules can be done. These operations allow to directly and explicitly control when and how modules are used. Modules in 2APL can be used to implement agent concepts such as agent role and agent profile. In fact, a module can be used as a mechanism to specify a role that can be enacted by an agent during its execution. Modules can also be used to implement agents that can represent and reason about other agents.

7 | Operations on Modules

7.1 | Creating and Releasing Module Instances

`create(s,m)`: this action can be used to create an instance of the module specification named "s". The name assigned to the create module instance is given by the second argument "m". The creating module instance (also called the owner of the created module instance or simply the owner) can use this name to perform further operations on the created module instance (also called the owned module instance). If a module specification is instantiated more than once, the creating module instance should assign a unique name to each created instance. A module instance with name "m" can be released by its owner by means of the release(m) action. The released module instance "m" will be removed or lost.

7.2 | Executing Module Instances

A module instance "m" can be executed by its owner through the execute(m,t) action. It has two effects:

1. It suspends the execution of the owner module instance
2. It starts the execution of the owned module instance.

The execution of the owner module instance will be resumed as soon as the execution of the owned module instance is terminated: an agent that executes an owned module instance, stops deliberating on its current cognitive state and starts deliberating on a new cognitive state.

The second argument "t" is the termination of the owned module instance condition. It consists of beliefs and goal tests performed on the beliefs and goals of the owned module instance. As soon as this condition holds, the execution of the owned module instance terminates and the execution control is given back to the owner module instance. This test is performed at each deliberation cycle.

Note that the owner cannot force the owned module instance's execution to stop since its own execution has been suspended.

7.3 | Testing and Updating Module Instances

The owner can test/query whether certain beliefs and goals are entailed by the beliefs and goals of its owned module instance "m" through actions $mB(\phi)$ and $m.G(\phi)$, where ϕ is a literal. In addition, the beliefs of a module instance "m" can be updated by means of the action $m.updateBB(\phi)$, where ϕ is a literal. Finally, goals can be added and removed from the goals of a module instance "m" by means of the action $m. < adoptgoal >$ and $m. < dropgoal >$, respectively. In the figure we can see a typical life cycle of a module:

A module instance A can create a new module instance B from a specification file. The module instance A can modify B's internal state using update actions. In this case, A updates the beliefs of B with literal "bel". The module instance A can transfer the execution control to the module instance B by the execute action. The execution of B continues until the state B satisfies the stopping condition "test". The execution control is then returned back to the module instance A. When A is active again, it can query B's internal state by the testing its beliefs or goals and release (remove) it.

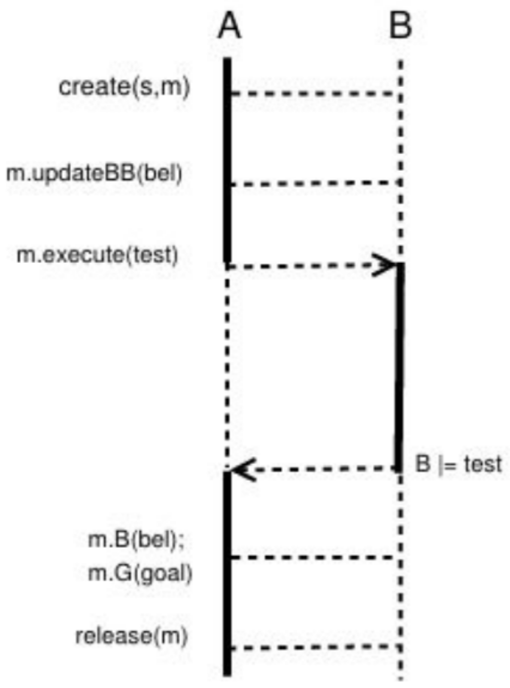


Figure 2.1: Module life cycle

8 | An Example of a Modular 2APL Multi-Agent Program

Suppose we need to build a multi-agent system in which one single manager and three workers co-operate to collect bombs in an environment called gridworld. The manager coordinates the activities of the three workers by asking them either to play the explorer role to detect the bombs in the grid-world environment or to play the carrier role to carry the detected bombs to a safe depot and store them.

```
<apaplmas>
    <environment name="blockworld" file="blockworld.jar">
        <parameter key="gridWidth" value="18"/>
        <parameter key="gridHeight" value="18"/>
    </environment>
    <agent name="m" file="manager.2apl"/>
    <agent name="w1" file="worker.2apl"/>
    <agent name="w2" file="worker.2apl"/>
    <agent name="w3" file="worker.2apl"/>
</apaplmas>
```

The manager module (i.e., manager.2apl) specifies the initial state of the manager agent with the name "m". The following is the code contained in manager.2apl :

```
1   beliefs:
2       worker(w1).
3       worker(w2).
4       worker(w3).
5
6   goals:
7       haveBomb.
8
9   pgrules:
10      haveBomb <- bomb(POS) and not assigned(carrier(POS), _) and
11      worker(W) and not assigned(_, W) |
12      { send(W, request, play(carrier, POS));
13          +assigned(carrier(POS), W);
14      }
15      haveBomb <- not bomb(_) and worker(W) and not assigned( _, W) |
16      { send(W, request, play(explorer));
17          +assigned(explorer, W);
18      }
19
20  pcrules:
21      message(A, inform, _, _, bomb(POS)) <- true |
22      { SetBomb(POS);-assigned(explorer, A);
23      }
24      message(A, inform, _, _, done(POS)) <- true |
25      { RemoveAssigned(carrier(POS), A);-bomb(POS);
26      }
27      message(A, inform, _, _, failed(POS)) <- true |
28      {-assigned(carrier(POS), A); }
```

The two PG-rules (lines 10 and 15) are used to assign specific roles to the worker agents.
The first PG-rule works in this way : if the manager believes there is a bomb at a certain position for which no agent is asked to carry it and there is a worker agent without any role assignment, then the manager sends a message to the worker agent asking the worker to carry the bomb, i.e., the manager asks the worker agent to play the carrier role. After sending the message, the manager updates its administration with the fact that there is a role assigned to the worker.

The second PG-rule asks a worker that has no role assignment to explore a bomb if the manager has no information about bombs.
The three PC-rules are used to process the incoming messages. For example, the manager could register the information about the found bombs and unregister the worker agents from their explorer role such that they can be asked for playing another role.

The worker agent, specified by the following code, is an agent that waits for requests to play either the explorer or the carrier role.

```
include:
   moving.2apl;

beliefs:
   manager(m).

plans:
   B(is(X,int(random(15))));
   B(is(Y,int(random(15))));
@blockworld(enter(X, Y,blue),_);

pgrules:

/* Wanderaround */
<-true|
{
   if B(prob(0.05))
      gotoRandomPos(15,15);
}

pcrules:
   message(A,request,_,_,play(explorer))<-manager(A)|
   {
      create(explorer,myexp);
      myexp.execute(B(bomb(POS)));
      send(A,inform,bomb(POS));
      release(myexp);
   }
```

```
29
30      message(A,request,_,_,play(carrier,POS))<-manager(A)|
31      {
32          create(carrier,mycar);
33          mycar.updateBB(bomb(POS));
34          mycar.execute(B(doneorerror));
35          if mycar.B(done)
36          {
37              send(A,inform,done(POS));
38          }
39          else
40          {
41              send(A,inform,failed(POS));
42          }
43          release(mycar);
44      }
```

When it receives a request to play the explorer role from the manager (line 22), it creates an explorer module instance and executes it (line 24-25).
When the execution of the module instance halts, the worker agent sends the position of the detected bomb to the manager (line 26), and finally releases the explorer module instance (line 27).
It is important to note that for the worker agent **the creation of an explorer module** instance and executing it is the same as **playing the explorer role**.
The second PC-rule of the worker agent (line 30) is responsible for carrying bombs by creating a carrier module instance (line 32), adding the bomb information to its beliefs (line 33), and executing it until either it has found the bombs (done condition) or an error has occurred. In other words, this second rule indicates when the worker agent should play the carrier role.

The explorer module (i.e., the implementation of the explorer role) has the goal to find bombs. In order to achieve this goal, it goes to a random location in the gridworld, performs a sense bomb action there and, if successful, adds the position of the detected bomb to its own local beliefs (line 27). Note that this belief information satisfies the stopping condition of the module instance (see line 25 of the **previous code**) since the goal foundBomb is achieved as soon as bomb(POS) is added to its beliefs (line 9).

```
1   include:
2       moving.2apl;
3
```

```
    beliefupdates:
        { at(OLDPOS) } UpdatePosition(POS) { not at(OLDPOS), at(POS) }
        { true } UpdatePosition(POS) { at(POS) }

    beliefs:
        foundBomb :- bomb(_).
        prob(P) :- is(X, rand), X < P.

    goals:
        foundBomb.

    pgrules:
    /* Wander around */
        foundBomb <- prob(0.05) |
        {
            gotoRandomPos(15,15);
        }

        foundBomb <- true |
        {
            @blockworld( senseBombs(), BOMBS );
            if B( BOMBS = default,X1,Y1 | REST ) then
            {
                +bomb(X1,Y1);
            }
        }
```

The carrier module (i.e., the implementation of the carrier role) has a goal to store a bomb (line 9). If the agent is at the same position as the bomb, then the goal can be achieved by picking up the bomb, storing it in the depot, and removing that bomb from its own local beliefs. Note that the removal of the bomb will imply that the job of carrying and storing the bomb is done, which in turn satisfies the stopping condition of the carrier module instance (see line 34).
However, if the agent is not at the bomb's position, then it will try to modify its position. If the new position is again not the same as the bomb's position, then it will adopt a goal to be at that position. This is done by the first PG-rule. The third and fourth PG-rules are to drop the carrying bomb at the safe location. The final PR-rules are for the case that the pickup and drop actions fail. In these cases, the beliefs is updated with an error message which will satisfy the stopping condition of the carrier

module instance (see line 34).

```
 1    include:
 2       moving.2apl;
 3    beliefs:
 4       trap(0,0).
 5       bombStored :- not bomb(_), not carrying_bomb.
 6       done :- bombStored.
 7
 8    goals:
 9       bombStored.
10
11    pgrules:
12       bombStored <- bomb(POS) and not carrying_bomb and not at(POS) |
13       { updatePosition();
14          if B(bomb(POS) and not at(POS))
15             adopta(at(POS));
16       }
17
18       bombStored <- bomb(POS) and not carrying_bomb and at(POS) |
19       {
20          updatePosition();
21          if B(at(POS))
22          { @blockworld( pickup(), _ );
23             +carrying_bomb;
24             -bomb(POS);
25          }
26       }
27
28       bombStored <- carrying_bomb and trap(TRAPPOS) and not at(TRAPPOS) |
29       { adopta(at(TRAPPOS)); }
30
31       bombStored <- carrying_bomb and trap(TRAPPOS) and at(TRAPPOS) |
32       { @blockworld( drop(), _ );
33          -carrying_bomb;
34       }
35
```

```
    prrules:
        @blockworld(pickup(), _); REST; <- true | { +error; }

        @blockworld(drop(), _); REST; <- true | { +error; }
```

The file moving.2apl contains codes for implementing the movement of the working agents.

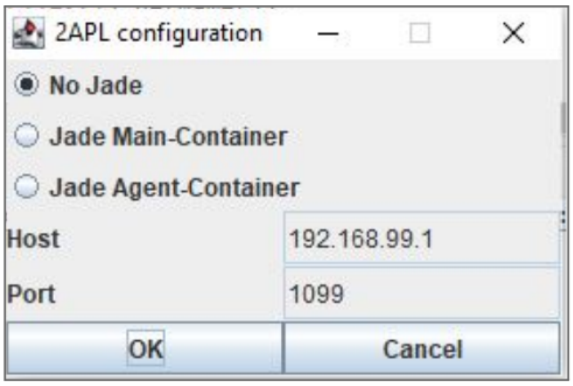
Tools

Different tools were developed to support the implementation and execution of multi-agent systems programmed in the 2APL programming language.

9 | 2APL Platform

The 2APL platform provides a graphical interface through which a user can load, run and monitor the execution of 2APL multi-agent programs using several tools, such as a state tracer, a log window and a message monitor (that we will discuss later), and allows for different execution modes, in order to create a communication among agents that run on different machines connected in a network.

By using the Hybrid 2APL version, we have to run the Java application on Eclipse; on startup, the following pop-up is shown:


2APL configuration

● No Jade
○ Jade Main-Container
○ Jade Agent-Container

Host	192.168.99.1
Port	1099

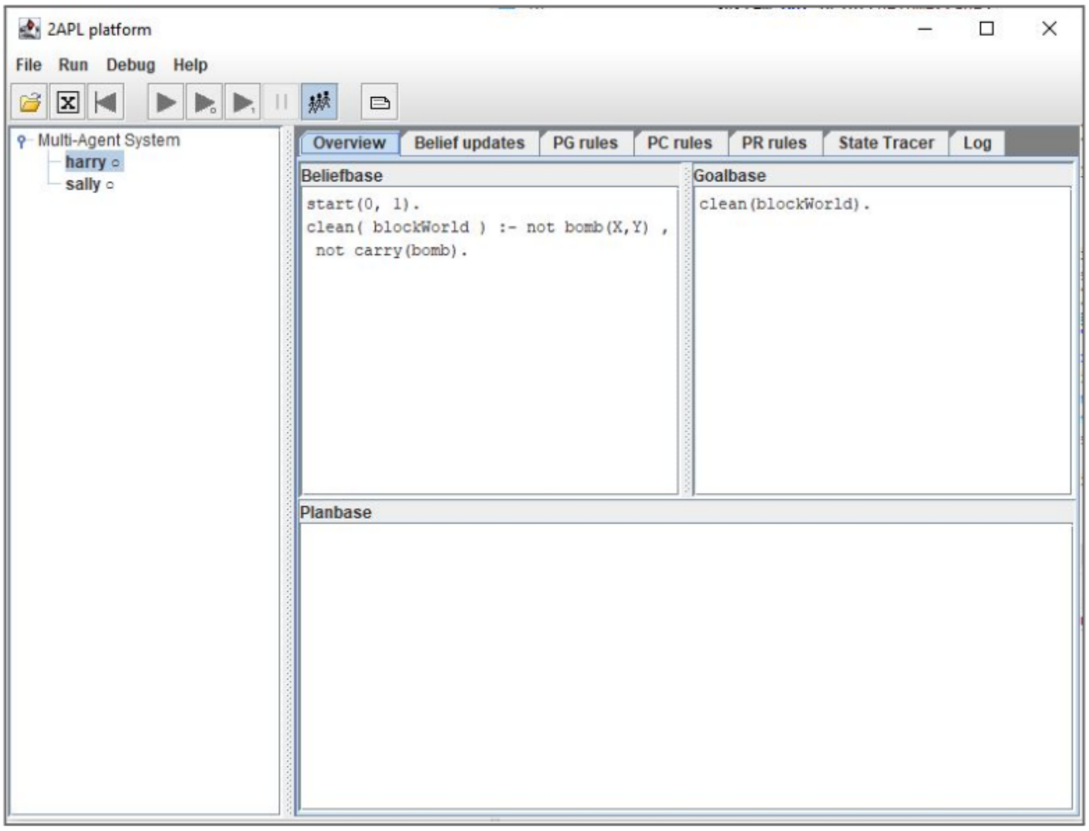
Figure 3.1: 2APL configuration window

It let us decide the configuration of the platform : the first option allows the use of the 2APL platform in a stand-alone mode without using Jade, and the last two options start 2APL on top of the Jade platform. These last two options are useful when we want to run a multi-agent program in a distributed environment, on different machines.

The platform has different Execution Monitoring Tools, to monitor the execution of individual agents. There are three monitoring tools and they can be accessed through the “Debug” toolbar button. They are activated by default and can be deactivated to improve the performance of the 2APL platform.

9.1 | Auto-update overview

When this tab is active, the beliefs, goals and plans of the current state of the selected agent are presented in the Beliefbase, Goalbase and Planbase panels, respectively, to observe the selected agent's mental state. Executing the multi-agent program will present updated informations.



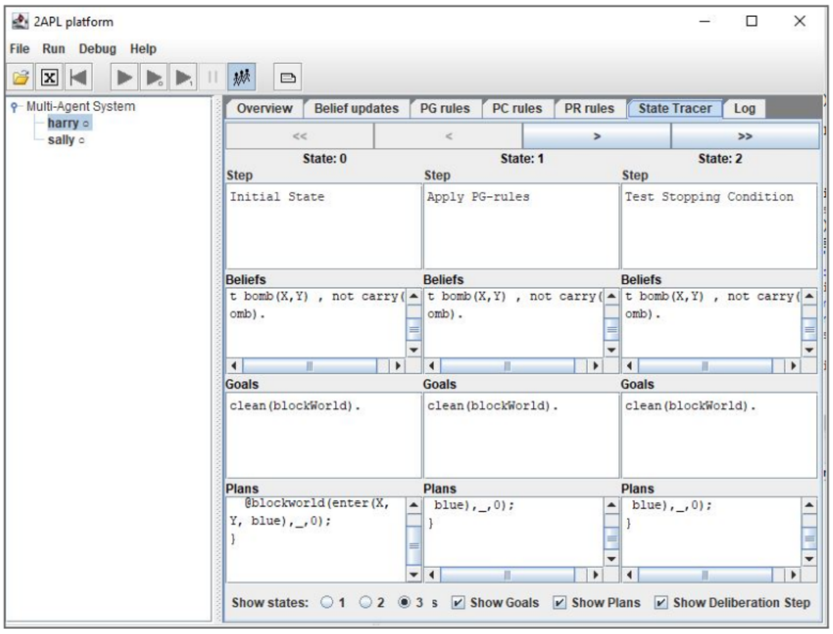
```
start(0, 1).
clean( blockWorld ) :- not bomb(X,Y) ,
 not carry(bomb).
```

```
clean(blockWorld).
```

Figure 3.2: Auto-update Overview

9.2 | The State Tracer

This tab can be seen as a temporal version of the Overview tab and stores the beliefs, goals and plans of all agents during the execution. It 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 to show on one screen and can also decide what to show.



State: 0

Step

```
Initial State
```

Beliefs

```
t bomb(X,Y) , not carry(
omb).
```

Goals

```
clean(blockWorld).
```

Plans

```
   @blockworld(enter(X,
Y, blue),_,0);
}
```

State: 1

Step

```
Apply PG-rules
```

Beliefs

```
t bomb(X,Y) , not carry(
omb).
```

Goals

```
clean(blockWorld).
```

Plans

```
 blue),_,0);
}
```

State: 2

Step

```
Test Stopping Condition
```

Beliefs

```
t bomb(X,Y) , not carry(
omb).
```

Goals

```
clean(blockWorld).
```

Plans

```
 blue),_,0);
}
```

Figure 3.3: State tracer

9.3 | The Log

This tab presents information about the deliberation steps of individual agents. It is the agent's execution log. It indicates if an agent has executed an action, processed an internal event, a message or an external event.

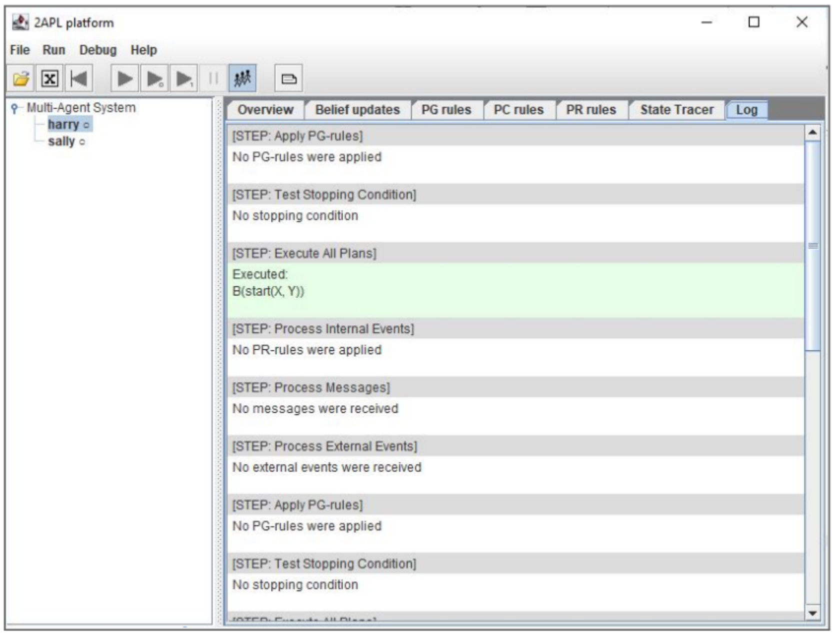
10 | 2APL Eclipse Plug-In Editor

Useful to implement easily 2APL multi-agent programs.
Using the editor, it is possible to run and monitor 2APL multi-agent programs directly from Eclipse. It is an Eclipse Xtext based plugin. It can be used also if multi-agent programs already exist; in fact, they can be developed further with this editor.

11 | 2APL Benchmark

The 2APLBenchmarker library is a generic benchmarking tool for 2APL and can be found in the package "apapal.benchmarking".

The benchmark should be called by using the benchmark command line option, followed by the .mas file that should be benchmarked. This will hand over the execution of the multi-agent system



```
[STEP: Apply PG-rules]
No PG-rules were applied

[STEP: Test Stopping Condition]
No stopping condition

[STEP: Execute All Plans]
Executed:
B(start(X, Y))

[STEP: Process Internal Events]
No PR-rules were applied

[STEP: Process Messages]
No messages were received

[STEP: Process External Events]
No external events were received

[STEP: Apply PG-rules]
No PG-rules were applied

[STEP: Test Stopping Condition]
No stopping condition
```

Figure 3.4: Log window

to the bench marker, disable both the GUI and Jade and start the benchmarking immediately.

For each operation that is being benchmarked, the benchmark will print:

- A count of the number of times that the operation has been performed
- The total execution time in milliseconds
- The average execution time for one execution in milliseconds

Example - Harry and Sally

This example can be found in the files given by the 2APL folder that can be downloaded from the site of the language.

First of all, we should introduce some elements to understand how the multi agent program is built.

- **.mas files** : the file with this extension specifies the multi-agent system by indicating which agents should be created, which **.2apl** files initialize the agents, which environments they can acess and which **.jar** files contain the environment.

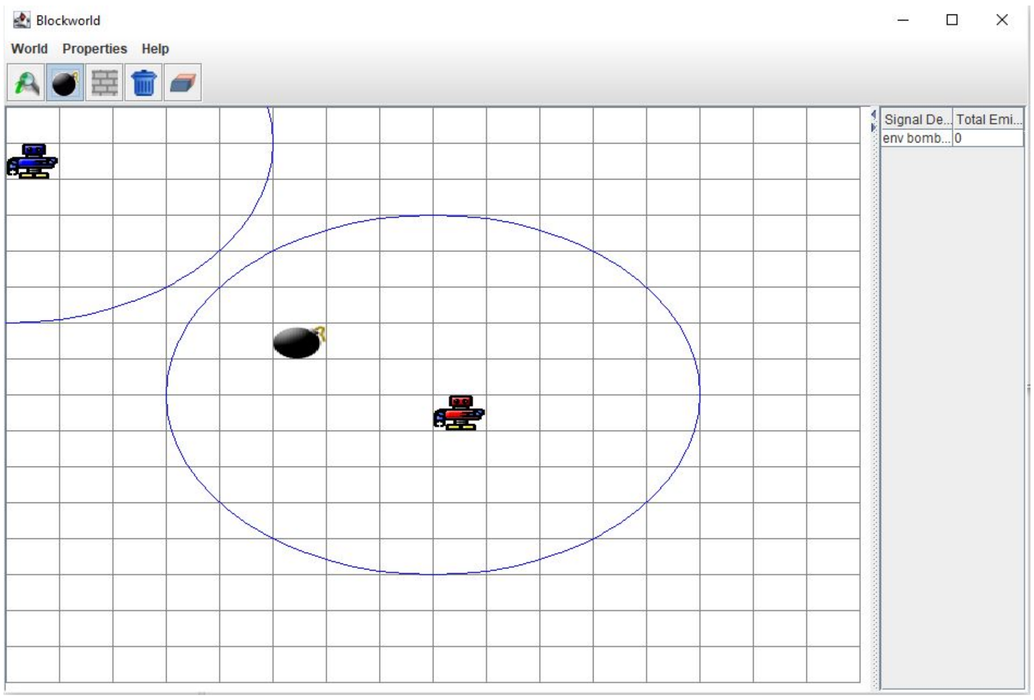
```
<apaplmas>
    <environment name="blockworld" file="blockworld.jar">
        <parameter key="gridWidth" value="18" />
        <parameter key="gridHeight" value="18" />
        <parameter key="entities" value="2" />
    </environment>

    <agent name="harry" file="harry.2apl" />
    <agent name="sally" file="sally.2apl" />
</apaplmas>
```

- **.2apl files** : these files contain the code that specifies the beliefs, goals and plans of each agent. The implementation of these elements has been explained in the first chapter of the document.

Now that we have a basic knowledge of how the code works, we can describe the example.

This example is about two agents, Harry and Sally, who are located in the so-called "blockworld environment". The "blockworld environment" is a $n \times n$ grid, which can contain agents, bombs and dustbins in which bombs can be thrown away. Sally is responsible for searching for bombs and notifying Harry when she finds one. Harry is responsible for cleaning up the blockworld by picking up the bomb and throwing it in the dustbins.

However, the code about this example is not complete since Harry has been programmed in order to just drop the bombs in the `cell (0,0)`.



Signal De...	Total Emi...
env bomb...	0

Figure 4.1: Harry (blue agent) and Sally (red agent) Example