MASD Assignment 1: A Review of 2APL

Claudio Saponaro, Pau Bosch, Alexander Temper

April 2025

Abstract

In this report, we will present 2APL ("*A Practical Agent Programming Language*"), a BDI-based agent-oriented programming language. This programming language facilitates the implementation of multi-agent systems consisting of individual agents that may share and access external environments, using an integration of declarative and imperative style programming by introducing and integrating declarative beliefs and goals with events and plans. It also provides practical programming constructs to allow the generation, repair, and (different modes of) execution of plans based on beliefs, goals, and events [1]. Throughout the document, we will discuss the language elements, operational and formal semantics, explaining the reasoning cycle the agents follow, and how they connect with the environment and other agents, and present the tools that are available to implement a multi-agent system in 2APL.

1 Introduction

We will discuss 2APL's (pronounced *double-A-P-L*) main aspects in this report, starting with the language elements in Section 2, where we introduce the various syntactic elements of the language, and explain their meaning informally. Following this, we will consider the operational semantics of the language in Section 3, which will be mainly explaining reasoning cycle of individual agents. Then, we will take a brief look at the formalization of 2APL in Section 4 and follow with two sections on things outside 2APL, namely 2APL's communication standards in Section **??** and development tools in Section 6. Additionally, we provide answers to the questions explicitly asked for in the assignment in a "*Questions & Answers*" section in Section 7.

2 Language elements

In this section, we'll explore the fundamental elements that comprise the language, examining its primary characteristics and structures.

§2.1 Beliefs In 2APL, *beliefs* are defined as the agent's internal perspective of the world, which are presumed to be accurate and are typically depicted as a conjunction of ground atoms. These beliefs are stored in the agent's belief base, which can be updated over time as the agent perceives new information from its environment and receives messages from other agents. These beliefs function as preconditions, which the agent must verify prior to formulating a new plan to achieve a specific goal.

Example. Consider this simple belief base containing three beliefs:

```
hasGold(0), trash(2,5), trash(6,8)
```

§2.2 Goals In 2APL, an agent's *goal* is defined as a desirable state that the agent seeks to achieve through a sequence of actions. Formally, goals are defined as a conjunction of ground atoms, and they act as prerequisites for the generation of plans, which are subsequently used to guide the agent's behavior.

Example. A simple example for a goal base is

```
[hasGold(5) and clean(blockworld), hasGold(10)]
```

where the agent has to collect five gold items and clean the blockward simultaneously while the second goal is to just collect ten gold items.

§2.3 Belief update actions A *belief update action* is defined as the execution of an action that results in an update to an agent's belief base. In more formal terms, an agent has the capacity to execute a belief update action if the precondition of the action is entailed by its belief base. The precondition is a formula consisting of literals composed by disjunction and conjunction operators. The following definition offers a synthetic articulation of this concept:

```
{not carry(gold)} PickUp() {carry(gold)}
```

If the agent believes that it is not carrying gold, the action of picking up the gold will trigger the belief update action which will update the belief base to carry(gold).

§2.4 Test actions A *test action*, defined as an action undertaken by an agent to ascertain the presence of a specific beliefs and objectives, is a crucial component of the agent's functionality. The verification of the belief base is achieved through the implementation of a belief query expression. It works similarly to a Prolog query, targeting the belief base and returning substitutions for all variables present in the query. As such, the queries can contain variables that can be unified against the belief or goal-base, allowing for flexible queries that use pattern matching. Formally, a belief query expression has this form: `B`(ϕ), where ϕ consists of literals formed by conjunction or disjunction operators. Similarly, a goal query expression checks a single goal within the goal base to see if it matches the query; a goal query expression has the form `G`(ϕ), where ϕ consists of atoms composed by conjunction or disjunction operators.

§2.5 Goal dynamics actions *Goal dynamics actions* are employed to modify the agent's goal base, and there are several ways to perform these updates. Goals can be added to the goal list using the adopt goal action, which manifests in two distinct forms: `adopta`(ϕ) and `adoptz`(ϕ) where ϕ is the goal to be added to the goal list; the first action inserts the goal at the beginning of the goal base, while the second action at the end. The goals can be dropped using the following actions: `dropgoal`(ϕ), `dropsubgoals`(ϕ), and `dropsupergoals`(ϕ). Recall that goals like ϕ are conjunctions of ground literals, that is, they are sets. Thus, in particular

- `dropgoal`(ϕ) removes the specific goal ϕ from the current set of goals,
- `dropsubgoals`(ϕ) removes every goal, which contains at least one of ϕ's literals, and
- `dropsupergoals`(ϕ) removes goals which contain all of ϕ's (and possibly other) literals.

Example. Assume the goal base is `[hasGold(5) and clean(blockworld), hasGold(10)]`. After calling `adopta(hasGold(15))`, the goal base would be the list

```
[adopta(hasGold(15), hasGold(5) and clean(blockworld)]
```

Now, both `deletesupergoals(clean(blockworld))` as well as `deletesubgoals(clean(blockworld), hasGold(5), hasSilver(2)` would render the goal base to

```
[adopta(hasGold(15)].
```

§2.6 Communication actions A *communication action* is defined as the trasmission of messages between agents. In 2APL, it can have three or five parameters and has either the syntax `send(Receiver, Performative, Content)` or the syntax `send(Receiver, Performative, Language, Ontology, Content)`, where `Receiver` is the name of the receiving agent, `Performative` is a speech act name, `Language` is the name of 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, and `Content` is an expression representing the content of the message.

Example. The following is an example of a communication action between agents:

```
send(Agent2, inform, english, weather_ontology, temperature(25)),
```

where the agent informs `Agent2` in English that the temperature is currently 25 degrees, as specified by some ontology concerning the weather.

§2.7 External actions Agents can interact with the environment through external actions. The consequences of these actions are determined by the environment itself and may not be known to the agents in advance. The syntax of external actions is defined as follows: `@env(ActionName, Return)`, where `env` is the name of the environment (implemented as a Java class), `ActionName` is a method call (of the Java class) that specifies the effect of the external action in the environment, and `Return` is a list of values, possibly an empty list, returned by the corresponding method.

Example. A simple example of an external action is `@blockworld(east(),L)`. Its invocation results in the execution of the '`east()`' method of the environment `blockword`, which might change the agent's x-coordinate. Whatever is returned is then accessible by the agent in the variable `L`.

§2.8 Plans A plan is defined as a sequence of actions that an agent performs in order to achieve a specific goal. The definition of actions is derived through inductive means as either a solitary action or a composition of plans, utilizing the following operators:

- the *sequence operator* (i.e.,, `p1;p2`),
- the *conditional choice operator* (i.e,,`if b then p1 else p2`),
- the *conditional iteration operator* (i.e, `while b then p1 else p2`), and
- the *non-interleaving operator* (i.e.,, `[p]`),

where `p, p1` and `p2` are plans and `b` is a test action.

The first three operators are defined in accordance with the standard methodology; the non-interleaving operator is employed to represent plans that must be executed without interleaving between actions of other plans. Initial plans are to be specified by the programmer; further plans are created through the means of a set of rules, namely *planning goal rules, procedure call rules and plan repair rules.*

Example. Consider the plan

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

The first plan is an atomic plan that ensures that immediately after the agent performs the external action `enter` in the blockworld environment, its belief base is updated with the initial position(5,5).

§2.9 Planning goal rules A *planning goal rule* (also *PG rule*) allows an agent to generate a plan based on certain goals and beliefs. This rule is defined by three components: the head, the condition, and the body of the rule. Given a certain belief, if the belief of the planning goal rules entails the belief of the agent's belief base and a particular goal, the plan is generated.

Example. A planning goal rule could be:

```
clean(R) <- pos(X1,Y1) and trash(X2,Y2) | {goTo(X1,Y1,X2,Y2); RemoveTrash()},
```

where the plan (denoted in braces) is generated by the agent if the goal is to clean a space `R`.

§2.10 Procedure call rules A *procedure call rule* (also *PC rule*) is defined as a rule that causes an agent to react to the environment or to a message from another agent that result in a change in an agent's belief. Procedure call rules are analogous to planning goal rules in that both involve three entries. However, the initial entry, designated as the "goal," is substituted by the message or event that prompts the agent's reaction.

Example. The following example illustrates the application of a PC rule:

```
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);
                                                }
```

In the scenario under consideration, Harry, the agent in question, responds to a message received from another agent, Sally. Provided that Harry's assessment does not align with the assumption that a bomb has already been placed at the specified location, the objective is appended to his goal base.

§2.11 Plan repair rules A *plan repair rule* (also *PR rule*) delineates a substitute plan to be implemented in the event of the failure of an initial plan, contingent upon the satisfaction of the rule's belief condition by the agent's belief base. Analogous to other practical reasoning rules, a plan repair rule comprises three components: two abstract plan expressions and one belief query expression. A plan repair rule stipulates that in the event an action of an agent's plan fails, given a specific belief, a new plan can be substituted for the unsuccessful plan.

Example.

```
@blockworld(east(), _); @blockworld(east(), _); X < -true | {
  @blockworld(north(), _); @blockworld(east(), _);
  @blockworld(east(), _); @blockworld(south(), _); X
}
```

In this example, `X` is utilized to denote a specific variable to denote a specific plan, that ensues subsequent to an unsucessful execution of the initial plan. The initial plan starts with the external actions `@blockworld(east(),_);@blockworld(east(),_)`.

The plan repair rule stipulates that in the event of such a plan's failure, that is the agent's inability to proceed twice in a rightward direction, the agent should instead attempt an upward movement, followed by two rightward and downward movements, and subsequently continue with the remainder of the plan, denoted by X. `X`.

§2.12 Abstract actions *Abstract actions* are analogous to functions in programming languages, that is, they allow the programmer to compose a plan from parameters given to the action. Typically, they are associated to a procedure call rule, in which the definition of those plans can be found. Executing an abstract action in a plan removes the abstract action from the plan and replaces it with an instantiation of the plan that is associated with the abstract action.

Example. Suppose an agent wants to move east in some two-dimensional world; clearly, she would like to update both her internal beliefs, as well as interact with the environment. Thus, `MoveEast(N)` could be an abstract action associated with the procedure call rule

```
MoveEast(N) <- true |
                {@blockword(east(), _);BeliefUpdateEast();if N > 1 {MoveEast(N-1)}}
```

Thus, invoking `MoveEast(2)` within a plan would move the agent east in the environment, afterwards update her beliefs about her position, and do it once again.

§2.13 Modularization 2APL allows for modularization, decomposing an agent's mental state into distinct, self-contained modules, each encapsulating its own beliefs, goals, plans, and reasoning rules [2]. These modules are treated like objects in object-oriented programming, which can be dynamically instantiated, activated, deactivated, and even replaced at runtime, giving programmers explicit control over each module's lifecycle. Each module represents a specific functionality, capability, or role, packaging together the cognitive elements required to perform that function. The agent mental state is modeled as a tree of module instances, when one module activates another, a link is created, forming a hierarchy or tree. Different module instances can be run in parallel, this allows an agent to be able to perform multiple tasks at once. The beliefs and goals of modules can be shared among them, where each module has an interface that specifies which beliefs and goals are public (exported) and which remain private. For example, a worker agent module might adopt a goal such as being at a specific location, although it may not have the means to achieve this directly, it exports the goal via its interface. A separate moving module, which also shares the same interface, imports this goal and generates plans to move the agent to the target location. Once the moving module senses that the goal has been achieved, it updates the belief base, which in turn causes the goal to be dropped automatically, thanks to the principle that prevents an agent from desiring a state it believes to be true. The framework allows modules to be created, modified, or released at runtime.

This concludes our discussion of the elements of the language. Now we will talk about how these elements interact with each other.

3 Operational semantics

A run in 2APL consists of a set of agents each stepping through a cyclical sequence of steps which define their actions called the *reasoning cycle*. Every agents deliberates that way in parallel, as such, the main part of how 2APL operates can be explained by explaining the reasoning cycle.

The execution and actions that an agent takes through its run are determined by the reasoning cycle. In 2APL it is specified as a cyclic process, where each step is a set of transition rules, which indicate the actions the agent can perform each time. The deliberation cycle can be observed in Figure 1.

It begins by applying all applicable planning goal rules. These rules are responsible for generating plans that support the agent's goals. Although an agent might have multiple goals, the planning goal rules are applied to one or a subset of goals in each cycle, ensuring that all goals receive attention over time.

Next, the agent executes the first action of every generated or existing plan. This is done to ensure that each plan is given a fair chance to make progress, avoiding the possibility of any single plan monopolizing the agent's attention.

Following the execution phase, the agent processes events from the external environment. This step can be seen as the perception phase, where incoming events or inputs are handled by applying the first applicable PC-rule, which will update the agent's internal belief base with new information about the world.

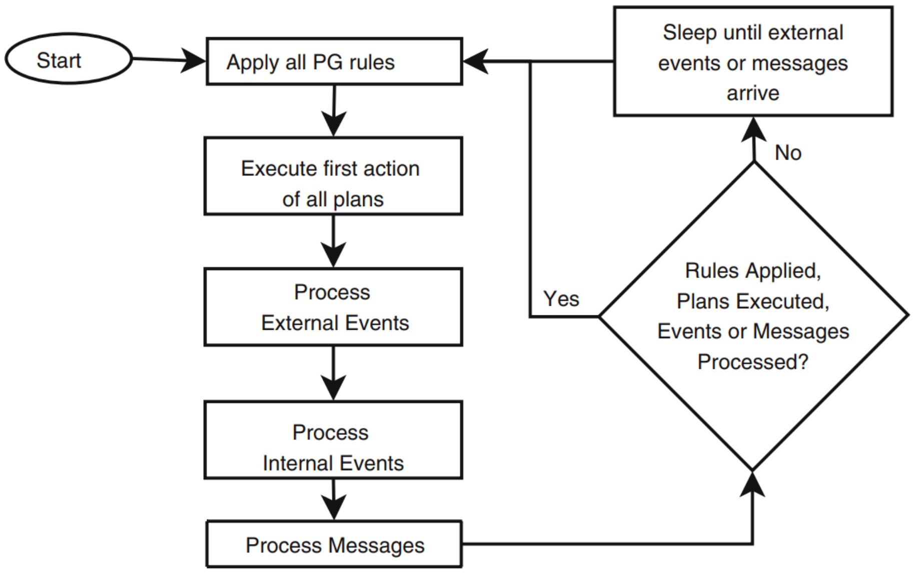

Start
Apply all PG rules
Execute first action of all plans
Process External Events
Process Internal Events
Process Messages
Sleep until external events or messages arrive
No
Yes
Rules Applied, Plans Executed, Events or Messages Processed?

Figure 1: 2APL Deliberation Cycle [1]

After handling external events, the agent turns to internal events, which are typically triggered when a plan fails. The internal events are processed by applying the first applicable plan repair rule, if a repair rule is applicable, the plan is modified to overcome the failure. Otherwise, the failed action remains and the plan may be retried in a future cycle. Then, the agent processes messages received from other agents. As with external events, these messages are processed using procedure call rules, potentially leading to creating of new plans that are then added to the agent's plan base. Finally, if the cycle completes without any rules being applied, any plans executed, or events processed, the agent concludes that there is no immediate work to do and enters a waiting state until a new external event or message arrives. This marks the end of the cycle, after which the process repeats when new inputs become available.

4 Formal semantics

Even though 2APL is operates based on its implementation, the implementation itself is informed by 2APL's formalization. The programming language is formally defined by transition rules between states in a so called *state transition system.* What this means is that a program's current state is defined to be able to transition to other states, determined by a set of rules of the form

$$\frac{\text{Premise}_1 \quad \text{Premise}_2 \quad ... \quad \text{Premise}_n}{A \to B}$$

which are to be read as *"state A can transition to state B if the premises above it are true."*. This notation is called *structural operational semantics notation.* The premises are then to be inductively justified using further transition rules, until at some point, some universally true statements would be arrived at — thus, one can reason about the execution of 2APL without having seen the actual program.

Before taking a look at an exemplary transition rule of 2APL, we first need to look at the elements of a program's state. A *(multi-agent) system state* is a tuple $S := \langle A_1, A_2, \ldots, A_n, \chi \rangle$, which contains n different *agent states* and an environment state χ. The transition rules for system states are simple, namely

$$\frac{A \to A'}{\langle A_1, \ldots, A, \ldots, A_n, \chi \rangle \to \langle A_1, \ldots, A', \ldots, A_n, \chi \rangle}$$

that is, a system state can go to another system state if one of its agents can transition, or

$$\frac{\chi \to \chi'}{\langle A_1, \ldots, A_n, \chi \rangle \to \langle A_1, \ldots, A_n, \chi' \rangle}$$

which means that a system state can go to another system state if its environment state can transition to another environment state.

The bulk of the complexity is within the *agent states*[1], which are tuples of the form $A_i := \langle i, B, G, P, E \rangle$, where

- i is the agent's identifier,
- B is the agent's set of beliefs,
- G is the agent's list of goals,

[1]Definition simplified for didactic reasons

- P are the agent's plans, and
- E is the agent's event base, containing events and messages sent to the agent.

Using this definition of agent states, we give two different examples for transition rules, firstly the *sequencing rule* and show how *belief updates* are defined formally. The sequencing rule shows how the sequencing operator '`;`' is to be interpreted. In particular, let p_1 and p_2 be two plans, then:

$$\frac{\langle i, B, G, \{p_1\}, E\rangle \rightarrow \langle i, B', G', \{\}, E'\rangle}{\langle i, B, G, \{p_1; p_2\}, E\rangle \rightarrow \langle i, B', G', \{p_2\}, E'\rangle}$$

This rule says that when an agent intends to do p_1 and then p_2, he should end up in the state in with only p_2 to do and the beliefs, goals and events that he would end up with if he only executed p_1.

Belief updates are defined as follows. Denote that calling belief update a on beliefs B results in beliefs B' as $T(B, a) = B'$. The corresponding (again simplified) rule for belief update actions naturally is

$$\frac{T(B, a) = B'}{\langle i, B, G, \{a\}, E\rangle \rightarrow \langle i, B', G, \{\}, E\rangle}$$

The power of these rules is that they can be combined well; for illustration, consider how one can reason about an agent that first updates her beliefs using belief update action a and thereafter proceeds to execute plan p:

$$\frac{\dfrac{T(B, a) = B'}{\langle i, B, G, \{a\}, E\rangle \rightarrow \langle i, B', G, \{\}, E\rangle}}{\langle i, B, G, \{a; p\}, E\rangle \rightarrow \langle i, B', G, \{p\}, E\rangle}$$

Next to the aforementioned rules, every other aspect of the language is formalized in a similar way.

As mentioned earlier, this allows for machine-independent reasoning over 2APL programs, however, this does not imply that 2APL is formally verifiable. There is no formal verification for 2APL, however, the authors of 2APL have provided ways to check safety- and liveness-properties for a 2APL-like agent programming languages in another work [3].

5 Communication and environments

§5.1 External environments An agent can perform actions in a variety of external environments, with each environment being implemented as a Java class. Specifically, any Java class that implements a specific interface can be utilized as a 2APL environment. The aforementioned interface requires that a class must have two methods implemented, namely `addAgent(String name)`, to incorporate an agent into environment, and `removeAgent(String name)` to remove an agent from the environment. Besides this, the environments can implement whatever is required for the design of the multi-agent system.

§5.2 Events and exceptions Events provide a mechanism for enabling communication and interaction between agents, predominantly through message passing. When an agent sends a message to another, then the receiving agent interprets this as an external event that can trigger a procedure call rule. These events can also be internal, generated within the agent itself, often as a consequence of goal adoption or belief updates.

On the other hand, exceptions are used to handle failures that occur during the execution of plan. When a plan encounters a failure during its execution, an exception is raised. This triggers the utilization of a plan repair rule as previously referenced.

§5.3 JADE and FIPA compliance The 2APL interpreter is built on the FIPA compliant JADE platform [4]. Therefore, the name of the receiving agent can be a local name or a full JADE name.

§5.4 Standalone mode The 2APL framework provides execution capabilities independent of the JADE infrastructure. In standalone mode, the platform requires the developer to implement essential multi-agent system components such as

- inter-agent communication mechanisms,
- agent execution control structures, and
- concurrency management between agent processes.

Although using 2APL in a standalone configuration provides extra flexibility, it introduces additional implementation complexity because engineers must address these architectural requirements independently rather than relying on JADE's built-in services.

To run 2APL in standalone execution mode, the following command syntax is used:

```
java -jar 2apl.jar [-nogui] [-nojade] [-help] [<path to MAS file>]
```

The `-nojade` parameter specifically disables the JADE configuration, enabling 2APL to operate in its standalone operational mode.

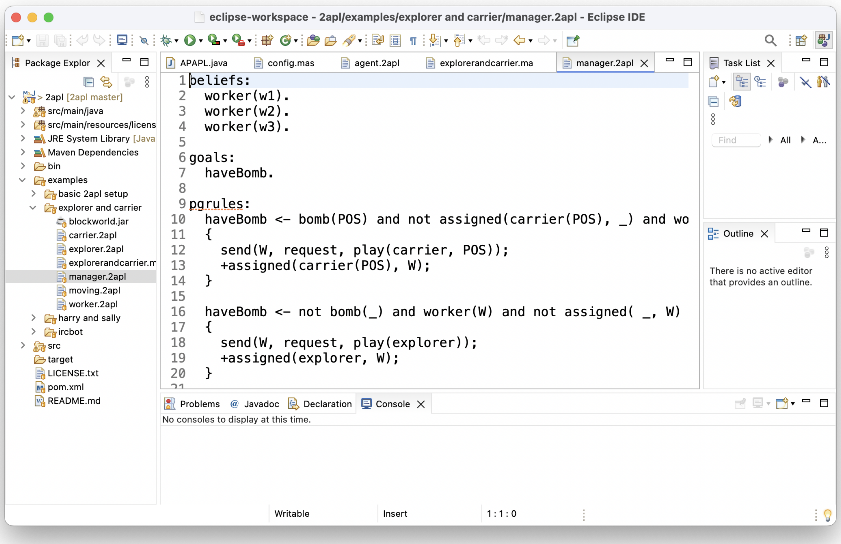
6 Tools

Finally, we are going to explain the main tools that come along with 2APL and how those can be used to improve the programmer's experience.

2APL is to be developed with the help of an Eclipse plugin, which is the main development tool available to create 2APL projects [5]. Once we download the language along with the plugin it comes with a set of development tools to support the entire development cycle of multi-agent systems. These tools include the following:

§6.1 Graphical User Interface The 2APL platform includes graphical interface that allows programmers to load and edit programs, as well as to run and debug programs. We show a snapshot of the IDE in Figure 2

§6.2 State Tracer Tool The State Tracer Tool allows developers to: inspect agent states, browse through the execution history of an agent, and view details such as its beliefs, goals, plans, and deliberation actions. It also serves to monitor internal states, helping to understand how the internal state of each agent evolves during the execution cycle. We can visualize the state tracer on Figure 3

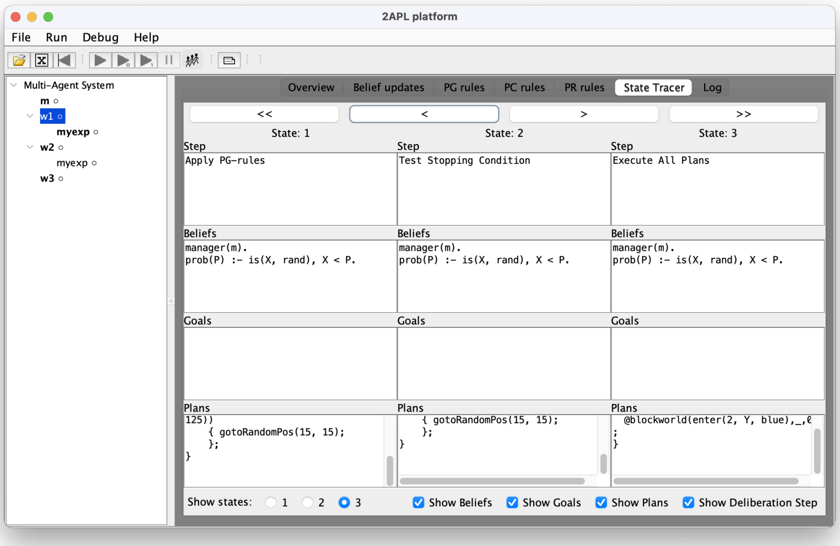


```
beliefs:
  worker(w1).
  worker(w2).
  worker(w3).

goals:
  haveBomb.

pgrules:
  haveBomb <- bomb(POS) and not assigned(carrier(POS), _) and wo
  {
    send(W, request, play(carrier, POS));
    +assigned(carrier(POS), W);
  }

  haveBomb <- not bomb(_) and worker(W) and not assigned( _, W)
  {
    send(W, request, play(explorer));
    +assigned(explorer, W);
  }
```

There is no active editor that provides an outline.

No consoles to display at this time.

Figure 2: Eclipse IDE for 2APL



State: 1

Step
```
Apply PG-rules
```
Beliefs
```
manager(m).
prob(P) :- is(X, rand), X < P.
```
Goals

Plans
```
125))
    { gotoRandomPos(15, 15);
    };
}
```

State: 2

Step
```
Test Stopping Condition
```
Beliefs
```
manager(m).
prob(P) :- is(X, rand), X < P.
```
Goals

Plans
```
    { gotoRandomPos(15, 15);
    };
}
```

State: 3

Step
```
Execute All Plans
```
Beliefs
```
manager(m).
prob(P) :- is(X, rand), X < P.
```
Goals

Plans
```
  @blockworld(enter(2, Y, blue),_,@
;
}
```

Figure 3: 2APL State Tracer Tool

§6.3 Inter-Agent Communication Support The 2APL platform is built on top of the JADE framework [4], a FIPA-compliant middleware, enabling agents to be hosted on different machines to communicate seamlessly. This support for distributed agent execution is essential for developing scalable multi-agent applications.

7 Questions & Answers

Here we answer questions that were explicitly asked for in the description of the assignment.

> Q: *How does belief revision work?*
>
> **A:** Beliefs are updated through so called *belief update actions*, which are described further in Paragraph §2.3.

> Q: *How does goal revision work?*
>
> **A:** Goals can be adopted and dropped through imperative commands which are parts of plans of the agent, see Paragraph §2.5.

> Q: *Can an agent have multiple goals?*
>
> **A:** Yes, each agent maintains a list of multiple goals.

> Q: *How does plan revision work?*
>
> **A:** Plans are revised through plan repair rules, see Paragraph §2.11

> Q: *How do agents handle critical events? Is there a way to react to sudden changes in the environment, stopping its current plan?*
>
> **A:** The deliberation cycle is designed to react to new external events immediately. Critical or emergency events—if present among external events—are processed right away by the relevant procedure call rules. This immediate processing allows the agent to generate new plans or modify its current behavior, effectively interrupting and replacing its ongoing goal intention or plan.

> Q: *Is there any Meta-Level reasoning?*
>
> **A:** 2APL has no meta-level reasoning, the deliberation process is procedural and fixed, without a separate layer that evaluates or modifies the reasoning process as a whole.

> Q: *Is there any theoretical model connected to the programming language?*
>
> **A:** Yes, 2APL is formalized as a state transition system defined by transition rules, see Section 4.

> Q: *Which are the theoretical model's elements? Are they the same as implemented in the language?*
>
> **A:** The theoretical model consists of system states, which in turn consist of agent states and environment states, connected via transition rules. The theoretical model closely aligns with the programming language's implementation. The elements are described further in Section 4.

Q: *How can the agents perceive the environment and execute actions on it?*

A: Agent's can interact with the environment using external actions, see Paragraph §2.7.

Q: *Is there any agent platform to support multi-agents?*

A: The execution of a multi-agent program is the interleaving execution of individual agent programs and the shared environ- ments. The platform provides a graphical interface through which an agent programmer can load, edit, run, and debug a 2APL multi-agent program. It also allows exploring the agent inner states generated by the execution, to see how the agent actuates. The platform is built on top of the FIPA-compliant JADE platform, which provides the infrastructure for managing multiple agents.

Q: *How can agents exchange information and knowledge among them?*

A: Agents exchange information using communication actions, see Paragraph §2.6

Q: *Do messages follow FIPA specifications?*

A: Since 2APL is implemented on the JADE platform [4], which is FIPA compliant, the communication actions follow FIPA standards, so it also uses performatives and defined message structure.

Q: *Which component manages the message passing between agents?*

A: The message passing between agents is managed by the underlying multi-agent system infrastructure, so the 2APL system uses communication actions for sending messages and the JADE platform manages their broadcast and reception.

Q: *Can ontologies be used in messages?*

A: The communication action has a parameter for *ontology* to provide a semantic context for the message content, enabling agents to interpret the symbols consistently. The parameter is optional, and generally it is not used if all the agents are programmed by the same user, since they would already know the semantic context of the messages based on their uses. For more on that, see Paragraph §2.6.

Q: *Are there any available development tools?*

A: Yes, there are debugging tools, graphical interfaces and IDE plugins available to develop in 2APL, see Section 6.

8 Bibliography

References

[1] Mehdi Dastani. "2APL: A Practical Agent Programming Language". In: *Autonomous Agents and Multi-Agent Systems* 16.3 (June 2008), pp. 214–248. ISSN: 1573-7454. DOI: `10.1007/s10458-008-9036-y`. (Visited on 03/18/2025).

[2] Mehdi Dastani Michal Cap. "Belief/Goal Sharing BDI Modules (Extended Abstract)". In: *Autonomous Agents and Multi-Agent Systems* (May 2011), pp. 1201–1202. (Visited on 03/18/2025).

[3] N. Alechina et al. "A logic of agent programs". In: *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 1*. AAAI'07. Vancouver, British Columbia, Canada: AAAI Press, 2007, pp. 795–800. ISBN: 9781577353232.

[4] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "Developing Multi-agent Systems with JADE". In: *Intelligent Agents VII Agent Theories Architectures and Languages*. Ed. by Cristiano Castelfranchi and Yves Lespérance. Berlin, Heidelberg: Springer, 2001, pp. 89–103. ISBN: 978-3-540-44631-6. DOI: `10.1007/3-540-44631-1_7`.

[5] *2APL installation guide*. URL: `https://www2.projects.science.uu.nl/Net2APL/prototype.html`.