BARCELONA SCHOOL OF INFORMATICS

MASD

MULTI-AGENT SYSTEMS DESIGN

Multi-Agent System Design: Analysis of the 2APL language

Authors:
Alex Aushev
Salvador Medina
Carles Miralles
Alejandro Suárez

Contents

1. Introduction

2APL is an agent-oriented programming language that facilitates the implementation of multi-agent systems (MAS) based on the BDI architecture. It is heavily inspired on Prolog and comes with declarative constructs and a complete logical inference system. However, it also incorporates the imperative paradigm to specify plans in a much more natural way. It is built on top of JADE which means, among other things, that it needs a JVM instance to be executed. The interaction with the real world is done by means of Java classes and the platform comes with many tools to facilitate development cycle.

The present document introduces the characteristics of the language. First we describe the different language elements through its syntactic constructs. After this we study the formal and operational semantics of the language. Another important aspect discussed in this document is the deliberation cycle. Although through the document we show small unconnected examples along our explanation, we have also devoted a section to put into context the discussed concepts by means of an illustrative cohesive example. Finally, we end with a comparison of 2APL against other MAS frameworks.

2. Language elements and syntax

This section is devoted to introduce the basic programming constructs in 2APL. These can be summarized as: beliefs and goals base, the plans to achieve the goals and the agents' actions.

2.1. Beliefs and goals

In BDI software models the **beliefs** represent the informational state of the agent, which can also include inference rules, allowing forward chaining to lead to infer new beliefs. Agent beliefs may not necessarily be true and in fact may change in the future.

```
Beliefs:
  pos(1,1).
  hasGold(0).
  trash(2,3).
  clean(blockworld) :- not trash(_,_).
```

Listing 1. Beliefs syntax in 2APL

On the other hand a **goal** is a desire that has been adopted for active pursuit by the agent. The agent must be consistent in this pursuit. For instance: one should not maintain simultaneous goals to go to a party and to stay at home even though they could both be desirable.

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

Listing 2. Goals syntax in 2APL

2.2. Basic actions

Actions specify the capabilities of agents, that is, actions that an agent can perform to achieve its goals. There are five kind of actions:

- **belief update action**: when executed, it updates the belief base of an agent. This type of action is specified in terms of pre-conditions (predicates that must hold for the action to be available) and post-conditions (changes to the belief base of the agent).

```
  {trash(X,Y}) and pos(X,Y)}
    RemoveTrash()
  {not trash(X,Y)}

```

Listing 3. Belief update action

- **test action:** A test action performed by an agent checks whether the agent has certain beliefs or goals. The syntax of a belief test action is B(ϕ), while the syntax of a goal test action is G(ϕ). A test action can be used in a plan to (1) instantiate variables in the subsequent actions of the plan (if the test succeeds), or (2) block the execution of the plan

(if the test fails). For example if an agent believes p(a) and has the goal q(b) then B(p(X)) G(q(X)) fails because is not possible unify two atoms with one variable but B(p(X)) G(q(Y) or r(X)) succeeds with {X/a, Y/b}.

- **goal dymanics actions:** The adopt goal and drop goal actions are used to add and remove a goal to and from an agent's goal base, respectively. We use adopta(ϕ), adoptz(ϕ) to add the goal ϕ at the beginning/end; and dropgoal(ϕ), dropsubgoals(ϕ) and dropsupergoals(ϕ) to remove, respectively, the goal ϕ, all the goals that are logical subgoal of ϕ and all goals that have ϕ as a logical subgoal.

- **communication action:** A communication action passes a message to another agent. We use send(Receiver, Performative, Language, Ontology, Content) or send(Receiver, Performative, Content) if the language and Ontology is assumed by the involved agents. It should be noted that 2APL interpreter is built on the FIPA compliant JADE platform. For this reason, the name of the receiving agent can be a local name or a full JADE name. A full JADE name has the form localname@host:port/JADE.

- **external action:** is used to change executed an action on an external environment. Its is denoted with the following syntax: @env(ActionName,Return), where env is the name of the agents 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.

- **abstract actions:** are used in the same way that procedures in imperative programming, that is, to reuse functionalities in a single action (see PC-Rules). More specifically, abstract actions expand to potentially bigger and more complex plans.

2.3. Plans

Plans are those constructions that allow achieve the goals of an agent by means of executing one or several of the aforementioned actions. They are written in an imperative fashion using something as classical as sequence of actions and structured programming (loops and conditionals). Plans can be distinguished according to how they are generated:

- **Initial plan:** which specifies the actions to be executed at the beginning of an agent's life.

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

Listing 4. Initial plan

- **Planning goal rules (PG-Rules):** the plans generated by these rules are goals oriented. The rules have the form form $[< goalquery >] \leftarrow < belquery > | < plan >$ and the plan is executed if the goal is in the goal stack and the belief query (optional) is fulfilled.

- **Plan repair rules (PR-Rules):** They generate plans to be executed as a consequence of any failed action and have the form $< plan > \leftarrow < belquery > | < plan >$. An action fails if:

  - The precondition of a belief update action is not entailed by the belief base.
  - If there is not applicable procedure rule for an abstract action.
  - If an external actions throws an ExternalActionFailedException.
  - The agent does not have access to an external environment.
  - An external action is not defined for an environment.
  - A test expression is not entailed by the belief and goal bases.
  - A goal being adopted that is already in the belief base or the goal is not ground.
  - An abstract plan if one action of the plan fails.

  The following example indicates a backup plan for any plan that starts with $@bw(est(),)$, $@bw(est(),)$ and fails for any reason. Notice the use of the variable X in the condition to indicate whatever other predicate and in the plan to add X to the backup plan:

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

Listing 5. PR-Rules

- **Procedure call rules (PC-Rules):** They allow the generation of plans as response to: messages sent by other agents, events generated by the environment. They also handle abstract actions. Check the following example:

```
PC-rules:
  message(A,inform,La,On,goldAt(X2,Y2))
  <- not carry(gold) |
   { getAndStoreGold(X2,Y2) }
  event(gold(X2,Y2),blockworld) <-
  <- not carry(gold) |
    { getAndStoreGold(X2,Y2) }
  getAndStoreGold(X,Y) <- pos(X1,Y1) |
   [ goTo(X1,Y1,X,Y);@bw(pickup(),_); PickUp();
     goTo(X,Y,s3,s3);@bw(drop(),_);StoreGold()
   ]
```

Listing 6. PC-Rules

2.4. Interfacing with an external environment

In 2APL agents can interact with external environments trough actions and events. These environments have to be implemented as a Java class extending the 2APL's `Environment` class and implementing the methods that the agent can invoke.

```
class Env1 extends apapl.Environment {
  ...
  public Term move(String agent, String direction) throws ExternalActionFailedException {
    if (direction.equals("north") {moveNorth();}
    else if (direction.equals("east") {moveEast();}
    else if (direction.equals("south") {moveSouth();}
    else if (direction.equals("west") {moveWest();}
    else throw new ExternalActionFailedException("Unknown direction");
    return getPositionTerm();
  }
  ...
    APLFunction event("eventName", arg1, arg2, ...)
    notifyAgents(event, agName);
  ...
}
```

Listing 7. Enviroment implementation

To call the external action in 2APL we have to include them in the plans as follows: `@env1(move("north"),L).`

On the other hand, it is also possible for the environment to trigger events. The agents can react in response to these events, modifying their beliefs or firing plans. As it has been previously said in this same section, the mechanism for dealing with external events is the PC rules. More specifically, we implement a rule that fires at the arrival of a certain event (provided that the rule's conditions are met). From the `Environment`'s side, the events are triggered via the `notifyAgents` method. From the agents' side, the events are processed as follows:

```
event(eventName(Arg1, Arg2, ...), env1) <- query | {
    action1;
    action2;
    ...
}
```

Listing 8. PC rule for handling an external event

3. Semantics

2APL is built over a strong theoretical model. Formal semantics rules also work as operational semantics rules in the specification of 2APL. Multiple formalisms are defined to clearly specify the rules that guide the execution of a 2APL. It is

also worth pointing out that the elements used for defining the language's semantics, like propositions, actions, events, plans; have a direct representation in its implementation.

2APL is a very powerful BDI-based programming language. It allows the programmer to define complex behaviors, by updating the beliefs, goals and plans of the different agents:

- Belief revision can be carried out by means of the belief update action.
- Goal is also possible with the adopt goal and drop goal actions.
- Plans can be revised by means of the plan repair rules, when any action of the plan fails to complete.
- In addition to normal events such as messages and failed actions, special events can be triggered by the environment at any point; that may lead to a sudden reaction by the agents.

Semantics in 2APL are defined in terms of a transition system. Transition rules define how one state, or configuration of the system, can be transformed into another in an execution step.

The execution of a 2APL MAS program modifies its initial configuration by means of transitions that are derivable from the transition rules. If multiple rules are applicable, the interpreter will select which transition rule to apply in a deterministic order.

3.1. Multi-Agent System Configuration

A Multi-Agent System configuration at a given step of the execution is defined as a set of atoms containing the configuration of every specific agent and the state of the external environments as well as the agents' access relations to them.

Definition Let A_ι be the configuration of agent ι and let χ be a set of external shared environments each of which is a consistent set of atoms. the configuration of a 2APL multi-agents system is defined as $\langle A_1, \ldots, A_n, \chi \rangle$.

The configuration of an individual agent consists of its identifier, beliefs, goals, plans, specifications or belief update actions, practical reasoning rules, substitutions result from queries to the belief an goal bases, and the received events. Note that practical reasoning rules and belief update actions are not included in this definition because the are not altered in any step of the execution.

Definition The configuration of an individual 2APL agent is defined as $A_\iota = \langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle$ where ι is the agent's identifier, σ is a set of belief expressions $\langle belief \rangle$ representing the agent's identifier, γ is a list of goal expressions $\langle goal \rangle$ representing the agents goal base, Π is a set of plan entries $(\langle plan \rangle, \langle pgrules \rangle, \langle ident \rangle)$ representing the agents plan base, θ is a ground substitution that binds domain variables to ground terms, and ξ the agent's event base.

The agent's event base is defined as $\xi = \langle E, I, M \rangle$; where E is a set of events from the external environments, I is a set of plans identifiers denoting failed plans and M is the set of messages sent to the agent.

Each plan entry is a tuple (π, r, p), where π is the executing plan, r is the instantiation of the PG-rule through which π is generated, and p is the plan identifier.

Considering the previous definitions, in the initial configuration of a 2APL MAS is defined as the tuple $\langle A_1, \ldots, A_n, \chi \rangle$, where the configuration of each agent is $A_\iota = \langle \iota, \sigma, \gamma, \Pi, \emptyset, \langle \emptyset, \emptyset, \rangle \rangle$. This is coherent with the fact that the agents could not receive any event and no ground substitution has been made.

3.2. Transition rules

Transitions define how a configuration can be reached from a previous configuration in an execution step. To formally define these transitions we use fractions in which the numerator defines the condition and the denominator defines how the configuration is transformed. We use $\vDash$ as a first-order entailment relation (as in the *Prolog* engine) and $\gamma \vDash_g \kappa$ to indicate that there exists a goal expression in γ which entails the goal κ and assume that $\gamma \vDash_g \texttt{true}$ for any goal base γ.

Additionally, we use $\gamma - \{\gamma_1, ..., \gamma_m\}$ to indicate that the elements from the subset $\{\gamma_1, ..., \gamma_m\}$ are removed from γ. Furthermore, we use $G(r)$, $B(r)$ and $P(r)$ to indicate the head κ, the belief condition β, and the plan π of a rule $r = (\kappa \leftarrow \beta | \pi)$.

Many of the transition rules defined in this section have the condition $\gamma \vDash_g G(r)$. It means that the transition can only be applied if the goal that is assigned to it is entailed by the agent's goal base. This condition ensures that a plan is not executed if its purpose (i.e., the goal for which the plan is generated) is not desirable anymore.

3.3. Transition rules for basic actions

Basic actions can be executed at individual agent level. In general, when the execution of a basic action fails, an exception is added to the ϵ_I component of the event base. This exception can later be used to repair the corresponding plan by means of plan repair rules. It should be noted that the application conditions of the transition rules that capture the success and the failure of a basic action exclude each other.

3.3.1 Skip action

The *Skip action* removes the plan entry $(\texttt{skip}, r, id)$ from the agent's plan base. The Skip action is formally defined as follows:

$$\frac{\gamma \vDash_g G(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{skip}, r, id)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma, \{\}, \theta, \xi\rangle} \tag{1}$$

3.3.2 Belief update action

A belief update action modifies the belief base when it is executed. After the execution of the action, its post-condition is entailed by the belief base. The modification of the belief base is realized by adding positive literals of the post-condition to the belief base and removing the atoms of negative literals of the post-condition from the belief base.

Observation Beliefs and goals are interrelated. If an agent believes something, the agent will not have the goal to achieve that fact. This implies that modifying the belief base may also change the goal base.

If the belief update action succeeds, the transition is defined as follows:

$$\frac{T(\alpha\theta, \sigma) = \sigma' \& \gamma \vDash_g G(r)}{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma', \gamma', \{\}, \theta, \xi\rangle} \tag{2}$$

Where T is a function that takes a belief update action and a belief base, and returns the modified belief base if the pre-condition of the action is entailed by the agent's belief base and $\gamma' = \gamma - \{\gamma_i | \sigma' \vDash \gamma_i\}$.

If the belief update action fails $[T(\alpha\theta, \sigma) = \bot]$, an exception is added to ξ. That means that if $\xi = \langle E, I, M\rangle$, then $\xi' = \langle E, I \cup \{id\}, M\rangle$.

Property The belief update actions preserve the consistency of the belief base. Moreover, goals will be removed from the goal base as soon as they are believed to be achieved. All other goals of an agent remain unachieved in the goal base.

3.3.3 Test action

Test actions check if the belief and goal queries within a $\langle test\rangle$ expression are entailed by the agent's belief and goal bases. It uses the already binded variables of the substitution θ and can produce a new substitution for the unbound variables τ, which is added to θ.

A test action $G(\psi_1)\&G(\psi_2)$ succeeds if the agent has either one single goal that entrails both ψ_1 and ψ_2 or two different goals that entrails one each. The test action $G(\psi_1\&\psi_2)$ only succeeds if there is a single goal that entrails both ψ_1 and ψ_2.

A test action can fail if any of its involved query expressions are not entailed by the belief or goal bases. In that case, the test action remains in the agent's plan base and an exception is generated to indicate its failure.

A test action φ can be executed successfully if φ is entailed by the agent's belief and goal bases.

$$\frac{(\sigma, \gamma) \vDash_t \varphi\theta\tau \& \gamma \vDash_g G(r)}{\langle \iota, \sigma, \gamma, \{(\varphi, r, id)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma, \{\}, \theta \cup \tau, \xi\rangle} \tag{3}$$

Similarly to the previous case, if the test action fails $[\neg\exists\tau : (\sigma, \gamma) \vDash_t \varphi\theta\tau]$ and $\xi = \langle E, I, M\rangle$, then $\xi' = \langle E, I \cup \{id\}, M\rangle$.

3.3.4 Adopt goal actions

Goals can be adopted and added to the agent's goal base by means of basic actions $\texttt{adopta}(\phi)$ and $\texttt{adoptz}(\phi)$. The first action adds the goal ϕ to the beginning of the goal base and the second action adds the goal ϕ to the end of the goal base.

Again, if the agent believes that the goal is already achieved or if the goal to be adopted is not ground $[\sigma \vDash \phi\theta \vee \neg\texttt{ground}(\phi\theta)]$, then the action fails and an exception is added $[\xi' = \langle E, I \cup \{id\}, M\rangle]$.

The adopt goal actions are defined as follows:

$$\frac{\sigma \nvDash \phi\theta \& \texttt{ground}(\phi\theta) \& \gamma \vDash_g G(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{adoptX}(\phi), r, id)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma', \{\}, \theta, \xi\rangle} \tag{4}$$

Where $\gamma' = \phi\theta + \gamma$ if *adoptX* is *adopta* and $\gamma' = \gamma + \phi\theta$ if *adoptX* is *adoptz*.

3.3.5 Drop goal actions

Goals can be dropped and removed from the goal base by means of $\texttt{dropgoal}(\phi)$, $\texttt{dropsubgoals}(\phi)$, and $\texttt{dropsupergoals}(\phi)$ actions. The first action removes from the goal base the goal ϕ, the second removes all goals that are subgoals of ϕ, and the third action removes all goals that have ϕ as a subgoal. These actions always succeed, even if there is no goal to be removed from the goal base. The drop goal actions are defined as follows:

$$\frac{\gamma \vDash_g G(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{dropX}(\phi), r, id)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma', \{\}, \theta, \xi\rangle} \tag{5}$$

Being $\gamma' = \gamma - \{f \| f \equiv \phi\theta \& ground(\phi\theta)\}$ if $dropX(\phi)$ is $dropgoal(\phi)$, $\gamma' = \gamma - \{f \| \phi\theta \vDash f \& ground(\phi\theta)\}$ if $dropX(\phi)$ is $dropsubgoals(\phi)$ and $\gamma' = \gamma - \{f \| f \vDash \phi\theta \& ground(\phi\theta)\}$ if $dropX(\phi)$ is $dropsupergoals(\phi)$.

Property If a goal is successfully adopted by an adopt goal action, then it is unachieved, i.e., it is not entailed by the belief base. The goal base remains ground after adopting and dropping goals. Dropping a goal, which is not ground, does not affect the goal base.

3.3.6 Abstract action

The execution of an abstract action replaces the action with the plan it represents. The relation between an abstract action and the plan it represents is specified by means of a procedure call rule (PC-rule).

Let α be an abstract action, $\varphi \leftarrow \beta|\pi$ be a variant of a PC-rule of agent and Unify be a function that returns the most general unifier of α and φ if they are unifiable, otherwise it returns $\perp$. A successful execution of an abstract action will replace it with a plan.

$$\frac{Unify(\alpha\theta, \phi) = \tau_1 \& \sigma \vDash \beta\tau_1\tau_2 \& \gamma \vDash_g G(r)}{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma', \{(\pi\tau_1\tau_2, r, id)\}, \theta, \xi\rangle} \tag{6}$$

If there is no PC-rule applicable $[\forall r' \in PC : (Unify(\alpha\theta, G(r')) = \perp \vee \sigma \nvDash B(r')) \& \gamma \vDash_g G(r)]$, then the execution of the abstract action is considered as failed and $[\xi' = \langle E, I \cup \{id\}, M\rangle]$.

3.3.7 Communication actions

An agent can send a message to another agent by means of the $send(j, p, l, o, \phi)$ action. The execution of the send action broadcasts a message which will be added to the event base of the receiving agent.

The transition of the sender (top) and the receivers (bottom) is are defined below:

$$\frac{\varphi = \langle \iota, j, p, l, o, \phi\rangle \& \gamma \vDash_g G(r))}{\langle \iota, \sigma, \gamma, \{(\texttt{send}(j, p, l, o, \phi), r, id)\}, \theta, \xi\rangle \underrightarrow{\varphi\theta!} \langle \iota, \sigma, \gamma, \{\}, \theta, \xi\rangle} \tag{7}$$

$$\frac{\varphi = \langle \iota, j, p, l, o, \phi\rangle \& \psi = message(j, p, l, o, \phi)}{\langle \iota, \sigma, \gamma, \pi, \theta, \langle E, I, M\rangle\rangle \underrightarrow{\varphi?} \langle \iota, \sigma, \gamma, \pi, \theta, \langle E, I \cup \{id\}, M\rangle\rangle} \tag{8}$$

Where $\varphi\theta = \langle \iota, j\theta, p\theta, l\theta, o\theta, \phi\theta\rangle$ and $\varphi\theta!$ means that the transition proceeds by broadcasting a message event and $\varphi?$ means that the transition cannot proceed unless φ is received.

3.3.8 External actions

By means of external actions, an individual agent can affect the external environment, shared with other agents. These actions return a value back to the agent who performed it with the result, or $\perp$ if the action failed. In the following definition, we assume that no other external action can be executed between the execution of the external action and the reception of its return value. The transition for external actions broadcast an event to the MAS and waits for the return value in one execution step. The formal definition is shown below:

$$\frac{t \neq \perp \& \gamma \vDash_g G(r)}{\langle \iota,\sigma,\gamma,\{(@env(\alpha(t_1,\ldots,t_n),V),r,id)\},\theta,\xi\rangle \xrightarrow{env(\alpha(t_1\theta,\ldots,t_n\theta),t)} \langle \iota,\sigma,\gamma,\{\},\theta\cup\{V/t\},\xi\rangle} \tag{9}$$

Where t is the return value of the external action and $env(\alpha(t_1\theta,\ldots,t_n\theta),t)$ is an event that indicates that agent ι performs action $\alpha(t_1\theta,\ldots,t_n\theta),t$ in environment env.

If the return value of the external function is an exception $[t = \perp]$, the execution of the external action is considered as failed. Consequently, the error event is added to the agent's event base $[\xi' = \langle E, I \cup \{id\}, M\rangle]$ and the action is not removed from the plan base.

3.4. Transition rules for plans

In 2APL, plans consist of basic actions composed by sequence, conditional choice, conditional iteration and not-interleaving operators. The transition rules describing plan executions are described in the next sections.

3.4.1 Sequence plan

In a sequence plan $\alpha;\pi$, the plan π is executed after basic action α has been executed. These kinds of plans are formally defined as follows:

$$\frac{\langle \iota,\sigma,\gamma,\{(\alpha,r,id)\},\theta,\xi\rangle \rightarrow \langle \iota',\sigma',\gamma,\{\},\theta',\xi'\rangle}{\langle \iota,\sigma,\gamma,\{(\alpha:\pi,r,id)\},\theta,\xi\rangle \rightarrow \langle \iota',\sigma',\gamma,\{(\pi,r,id)\},\theta',\xi'\rangle} \tag{10}$$

3.4.2 Conditional plan

In a conditional plan `if` φ `then` π_1 `else` π_2, the condition φ is evaluated with respect to the agent's belief and goals base and one of the plans π_1 or π_2 is executed, depending on whether or not the agent's belief and goal base entails φ. If the else part is not explicitly defined π_2 is considered a `skip` plan.

$$\frac{(\sigma,\gamma) \vDash_t \varphi\theta\tau \& \gamma \vDash_g G(r)}{\langle \iota,\sigma,\gamma,\{(\texttt{if}\varphi\texttt{then}\pi_1\texttt{else}\pi_2,r,id)\},\theta,\xi\rangle \rightarrow \langle \iota,\sigma,\gamma,\{(\pi_1\tau,r,id)\},\theta,\xi\rangle} \tag{11a}$$

$$\frac{\neg\exists\tau : (\sigma,\gamma) \vDash_t \varphi\theta\tau \& \gamma \vDash_g G(r)}{\langle \iota,\sigma,\gamma,\{(\texttt{if}\varphi\texttt{then}\pi_1\texttt{else}\pi_2,r,id)\},\theta,\xi\rangle \rightarrow \langle \iota',\sigma',\gamma,\{(\pi_2\tau,r,id)\},\theta,\xi\rangle} \tag{11b}$$

3.4.3 While plan

In a while plan `while` φ `do` π, plan π is executed again while φ is entailed by the agent's belief and goal bases. It is formally defined as follows:

$$\frac{(\sigma,\gamma) \vDash_t \varphi\theta\tau \& \gamma \vDash_g G(r)}{\langle \iota,\sigma,\gamma,\{(\texttt{while}\varphi\texttt{do}\pi,r,id)\},\theta,\xi\rangle \rightarrow \langle \iota',\sigma',\gamma,\{(\pi\tau;\texttt{while}\varphi\texttt{do}\pi,r,id)\},\theta,\xi\rangle} \tag{12a}$$

$$\frac{\neg\exists\tau : (\sigma,\gamma) \vDash_t \varphi\theta\tau \& \gamma \vDash_g G(r)}{\langle \iota,\sigma,\gamma,\{(\texttt{while}\varphi\texttt{do}\pi,r,id)\},\theta,\xi\rangle \rightarrow \langle \iota',\sigma',\gamma,\{\},\theta,\xi\rangle} \tag{12b}$$

3.4.4 Atomic plan

In atomic plans, the actions included in the plan are executed in a linear non-interleaved. The execution of an atomic plan is the non-interleaved execution of the maximum number of actions of the plan. An atomic plan can be defined like $[\alpha_1; \ldots; \alpha_n]$, where α_i is an action. The following transition rule defines the transition of one initial configuration to a final configuration, executing each intermediate transition due to each action α_i:

$$\frac{(\forall_i{:}1{\leq}i{\leq}m{\rightarrow}transition(A_i,Ai{+}1))\&\forall A{:}\neg transition(A_{m+1},A)}{\langle\iota,\sigma_1,\gamma_1,\{([\alpha_1;\ldots;\alpha_n],r,id)\},\theta_1,\xi_1\rangle{\rightarrow}\langle\iota,\sigma_{m+1},\gamma_{m+1},\Pi,\theta_{m+1},\xi_{m+1}\rangle} \tag{13}$$

Property The event base of the last configuration A_{m+1} resulted from performing atomic plan $[\alpha_1; \ldots; \alpha_n]$ contains an exception if action α_m of the atomic plan was executed and failed. The failed action α_m and all its subsequent actions will remain in the plan base.

3.4.5 Multiple concurrent plans

The following transition formally shows the fact that an agent executes one of the constituent actions of one of its plans at each execution in a concurrent and interleaved manner:

$$\frac{\langle\iota,\sigma,\gamma,\rho,\theta,\xi\rangle \rightarrow \langle\iota,\sigma',\gamma',\rho',\theta',\xi'\rangle}{\langle\iota,\sigma,\gamma,\Pi,\theta,\xi\rangle \rightarrow \langle\iota,\sigma',\gamma',\Pi',\theta',\xi'\rangle} \tag{14}$$

Where ρ is an action so that $\rho = \{\pi, r, id\} \subset \Pi$ and $\Pi' = (\Pi/\rho) \cup \rho'$.

3.5. Practical reasoning rules

The following sections present the transition rules that captures the application of the three types of practical reasoning rules: planing goal rules, procedure call rules and plan repair rules.

3.5.1 Planing goal rules

Plan generation in 2AP is done by applying PG-rules. PG-rules might indicate that a plan should be generated to achieve a certain goal or generated when the agent has a particular belief. In general, a PG-rule $[\kappa \leftarrow \beta|\pi]$ can be applied if κ is entailed by one of the agent's goals, β is entailed by the agent's belief base and there is no plan in the plan base generated by applying the very same PG-rule to achieve the same goal. Formally, we can define the planing goal rules as:

$$\frac{\gamma{\vDash_g}\kappa'\tau'\&\sigma{\vDash}\beta'\tau_1\tau_2\&{\neg}\exists\pi*{\in}P{:}(\pi*,(\kappa'\tau_1{\leftarrow}\beta|\pi),id'){\in}\Pi}{\langle\iota,\sigma,\gamma,\Pi,\theta,\xi\rangle{\rightarrow}\langle\iota,\sigma,\gamma,\Pi{\cup}\{(\pi'\tau_1\tau_2,(\kappa'\tau_1{\leftarrow}\beta|\pi),id)\},\theta,\xi\rangle} \tag{15}$$

3.5.2 Procedure call rules

The execution of abstract actions is based on the application of PC-rules, that generate plans to be replaced for these actions. Let $event(\phi, env) \in E$ be an event broadcasted by environment env,$message(s, p, l, o, e) \in M$ be a message broadcasted by agent s, and $Unify$ be a function that returns the most general unifier of two atomic formulas, or $\perp$ if unification is not possible. Let $\varphi \leftarrow \beta|\pi$ be a variant of a PC-rule of agent ι and $\psi = \langle E, I, M\rangle$ be the event base of agent ι. The transition rule for applying PC-rules to the events from E or M is defined as follows:

$$\frac{\psi \in E \cup M \& Unify(\psi, \varphi) = \tau_1 \& \sigma \vDash \beta\tau_1\tau_2}{\langle\iota, \sigma, \gamma, \Pi, \theta, \xi\rangle \rightarrow \langle\iota, \sigma, \gamma, \Pi', \theta, \xi'\rangle} \tag{16}$$

Note that if there is no PC-rule the head of which is unifiable with an event or message ψ from the agent's E or M $[\psi \in E \cup M \& \forall r \in PC : Unify(\psi, G(r)) = \perp]$, then the event or message will be removed from E and M respectively.

Property An event ψ for which there exists a rule r such that $Unify(\psi, G(r)) = \perp$ remains in the event base until it is processed.

3.5.3 Plan repair rules

Plan repair rules can be applied to replace a plan if an exception is received indicating that the execution of the first action of the plan π is failed. A plan repair rule $\pi_1 \leftarrow \beta\pi_2$ can be applied to repair π if the abstract plan expression π_1 matches π and β is entailed by the agent's belief base. The result of the match will be used to instantiate the abstract plan expression π_2 and to generate a new plan that replaces the failed plan:

$$\frac{PlanUnify(\pi, \pi_1) = (\tau_d, \tau_p, \pi*)\&\sigma \vDash \beta\tau_d\tau\&idinI}{\langle \iota, \sigma, \gamma, \{(\pi, r, id)\}, \theta, \langle E, I, M\rangle\rangle \rightarrow \langle \iota, \sigma, \gamma, \{(\pi_2\tau_d\tau\tau_p; \pi*, r, id)\}, \theta, \langle E, I/\{id\}, M\rangle\rangle} \quad (17)$$

4. Execution of the Multi-Agent System

4.1. Deliberation cycle

Figure 1 shows a graphical representation of the Deliberation cycle (DC) in 2APL. The interpreter of 2APL executes individual agents and the environment in parallel in an interleaving mode. Each cycle starts by applying all applicable PG-rules, each rule only one time [1]. So if there are additional goals that require applying a certain PG-rule that has been already applied in the current cycle, then those goals need to wait for the next cycle to use this PG-rule again. The next step in the DC is executing the first actions of all plans. In order to allow all plans to get a chance to be executed, only the first actions are applied.

After this, in the next three steps, the DC processes all events received from the external environment, all internal events indicating the failure of plans, and all messages received from other agents, respectively. In order to process an event from external environment it is necessary to apply the first applicable PC-rule (with the head event) to it [1]. For internal event processing similarly the first applicable PC-rule is being used but on a failed plan this time. Finally, for message processing the first applicable PC-rule with the head message is used. All these applied PC-rules add plans to the corresponding agents plan base.[1]. In the end of the DC it simply checks if there no rule could be applied, no plan could be executed, and no event could be processed. If no, then there's no need to run the DC again until some external events or messages arrive. If yes, then the next cycle begins.

In summary, 2APL provides a variety of update and revision actions on its belief, goals, events, and plans. Agents react to external, internal events and messages. All reactions to sudden changes in the environment need to be processed in the corresponding steps of the DC. There's no Meta-Level reasoning over the DC, however the language does implicit reasoning over the logical statements that takes its roots in first-order logic and a formal logic. In [1] authors of the paper additionally highlight two properties of the DC:

Property 1 If the execution of a plan fails, then the plan will either be repaired in the same deliberation cycle or get re-executed in the next deliberation cycle.

Property 2 If the first action of a failed plan is a belief update, test, adopt goal, abstract or external action and there is no plan repair rule to repair it, then the failed plan may be successfully executed in the next deliberation cycle.

.

5. Working with 2APL

5.1. Agent platform

2APL has its own agent platform which is implemented in Java as a standalone program. This platform includes an editor with a several monitoring tools. To run the platform one needs to install Java Runtime Environment (JRE) 6 or Java Developers Kit (JDK) 6 (or later versions) and download platform files from the official website. It supports all general operating systems: Windows, Mac OS X, Linux and Unix (Solaris).

There are two ways of running the 2APL agent platform. One is by directly opening a 2apl.jar file and another one is by using the platform as plug-in for Eclipse IDE. Additional instructions on running the platform can be found in the 2APL user guide, which goes with the platform. The 2APL platform is built on top of JADE and can be run on different machines that are connected through the net. To do so the platform uses a JADE agent-container. Figures 2 and 3 show two possible versions of interfaces of the 2APL agent platform.

In order to create a multi-agent system the platform requires three types of files. The first one is MAS specification file. This is a text file with the extension .mas that contains an XML specification of the multi-agent system. The name and path

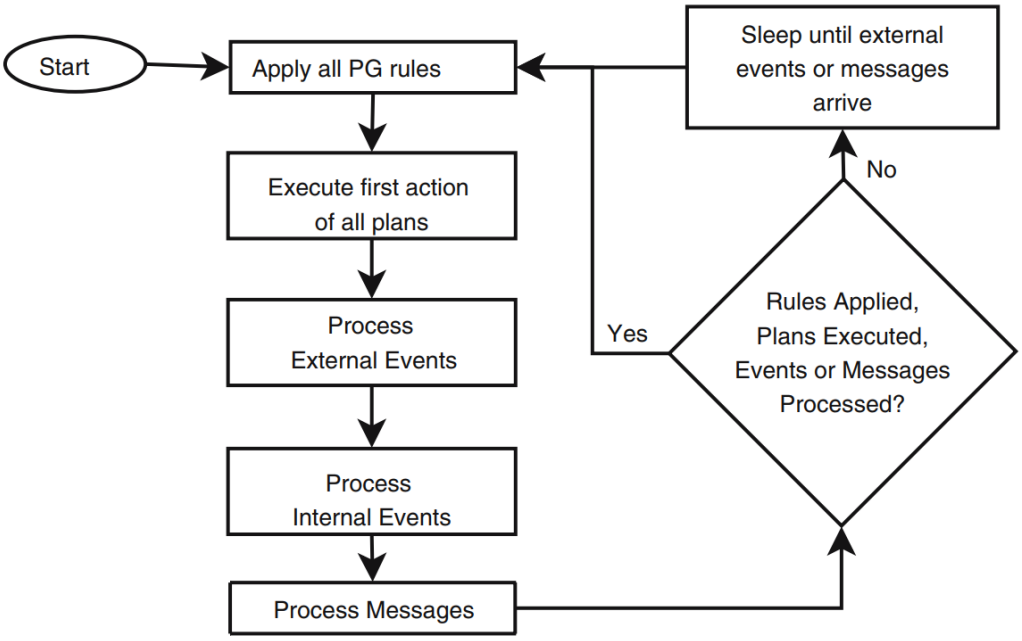

Figure 1. Deliberation cycle of 2APL

to the source of all agents and environments that start in the application are denoted here [1]. The second one is a 2APL agent file. This is a text file with the extension .2apl and contains the 2APL code of the agent [1]. This file should be provided for every agent in the system. And the last one is an environment file. This is a runnable JAR with the extension .jar that provides an interface with the environment. To create a 2APL multi-agent system it is necessary to provide all three types of files to the agent platform.

5.2. Messages

In 2APL agents communicate with each other through messages. A 2APL agent is assumed to be able to receive a message that is sent to it at any time. The received message is added to the M component of the event base of the agent and later can be used in PC-rules. In order to ensure that the sent message is received, 2APL presents a transition rule at multi-agent level which indicates that the sending and receiving agents exchange the message synchronously [1].

Sending messages is done by means of the *send* action. There are two version of this action. The full version requires all five arguments to be passed to the action. A 2APL message needs *Receiver* (to whom pass the message), *Performative* (inform, request, etc.), *Content* of the message, *Language* (that is used to express the content) and *Ontology* (that gives a meaning to the symbols in the content). In the short version the last two arguments can be left out since agents often assume them. 2APL interpreter is built on the FIPA compliant JADE platform and messages follow FIPA specifications [1].

5.3. Tools

2APL platform has additional tools that were designed to help developers monitor the work of a multi-agent system. These monitoring tools give additional information about the execution of each agent. There are three such tools and they are accessible from the built-in editor.

The first one is Auto-update Overview tool. The purpose of this tool is to provide information about the current state of all beliefs, goals and plans of the agent. This information updates automatically. In order to use this tool it is necessary to open the corresponding tab and choose the required agent. Figure 2 shows the interface of this tool.

The second one is State Tracer. It provides information on how all beliefs, goals and plans changed with the respect to a state of the agent. With this tool a developer can track the changes in agent's internal state. It stores the beliefs, goals, and plans of all agents during execution. This tool allows a developer to execute a multi-agent program for a while, pause the execution, and browse through the execution of each agent. It can be accessed similarly to the previous tool. An interface of the tool is shown in Figure 4.

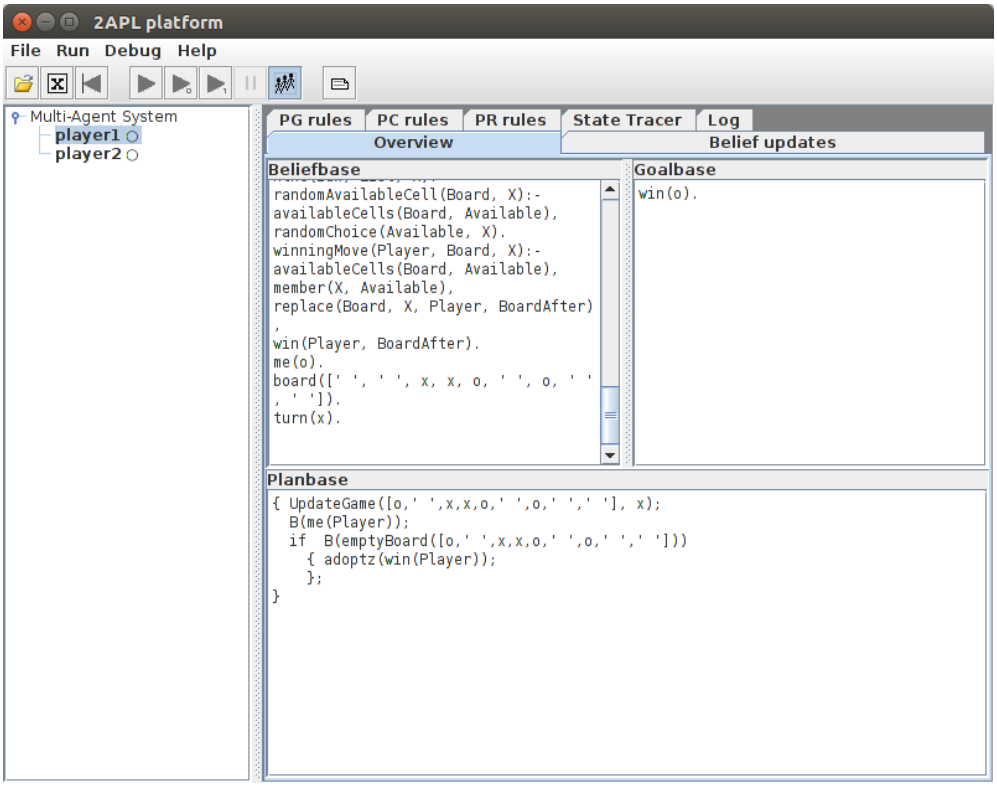

Figure 2. 2APL platform interface in the built-in editor, auto-update overview tool tab

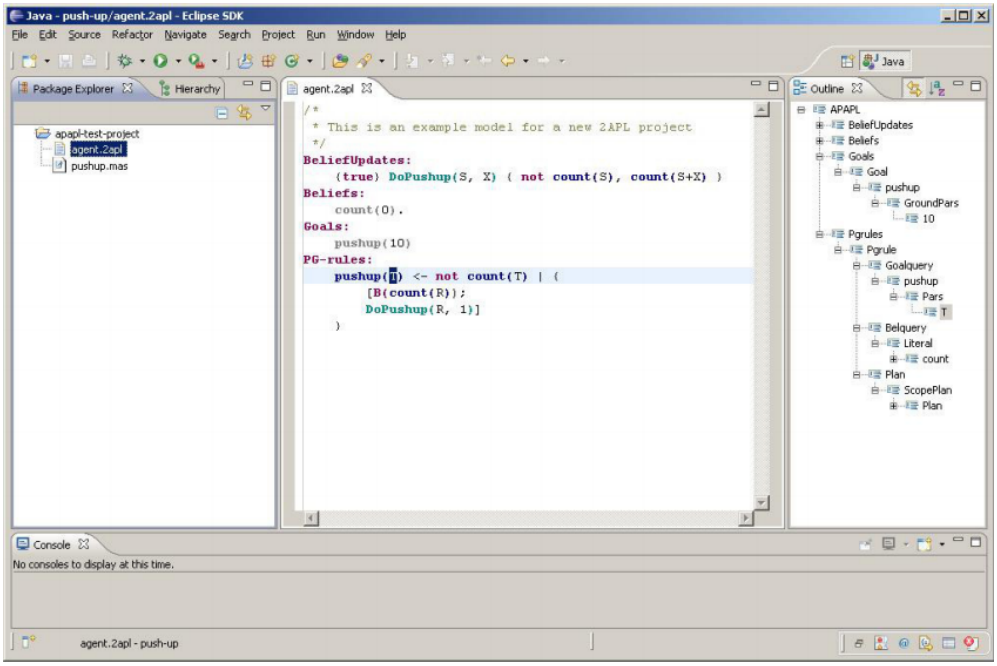

Figure 3. 2APL platform interface in Eclipse

Finally, a Log tool presents information about the deliberation steps of individual agents. A developer can browse through this window to see which deliberation steps have been preformed. This tool is focused on the deliberation cycle and all changes in it. It can be accessed form its tab and it is shown in Figure 5.

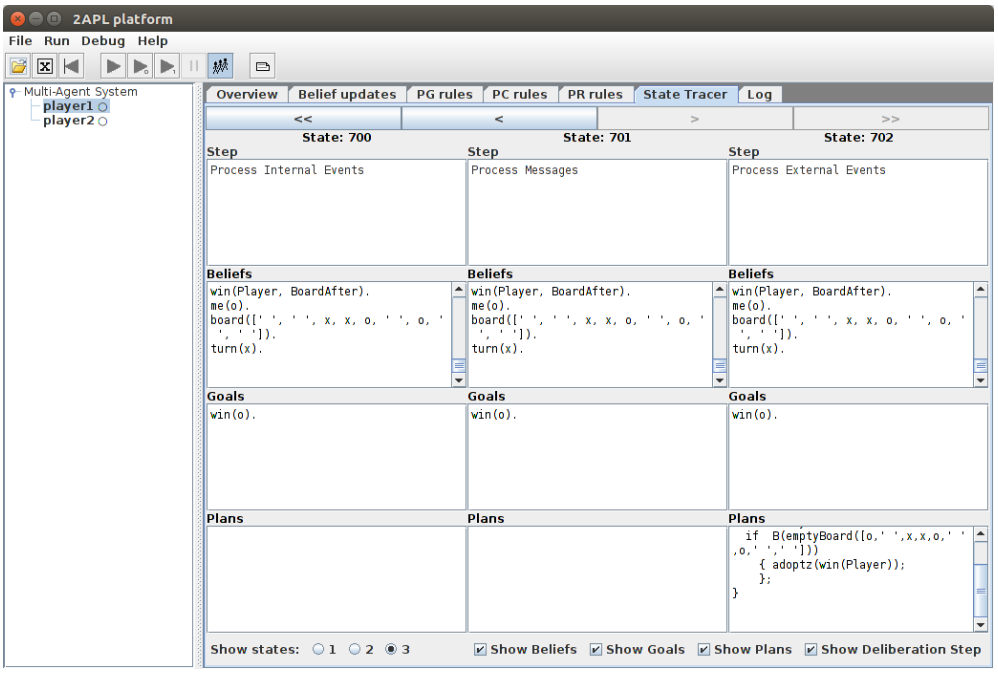

Figure 4. 2APL platform State Tracer tool interface

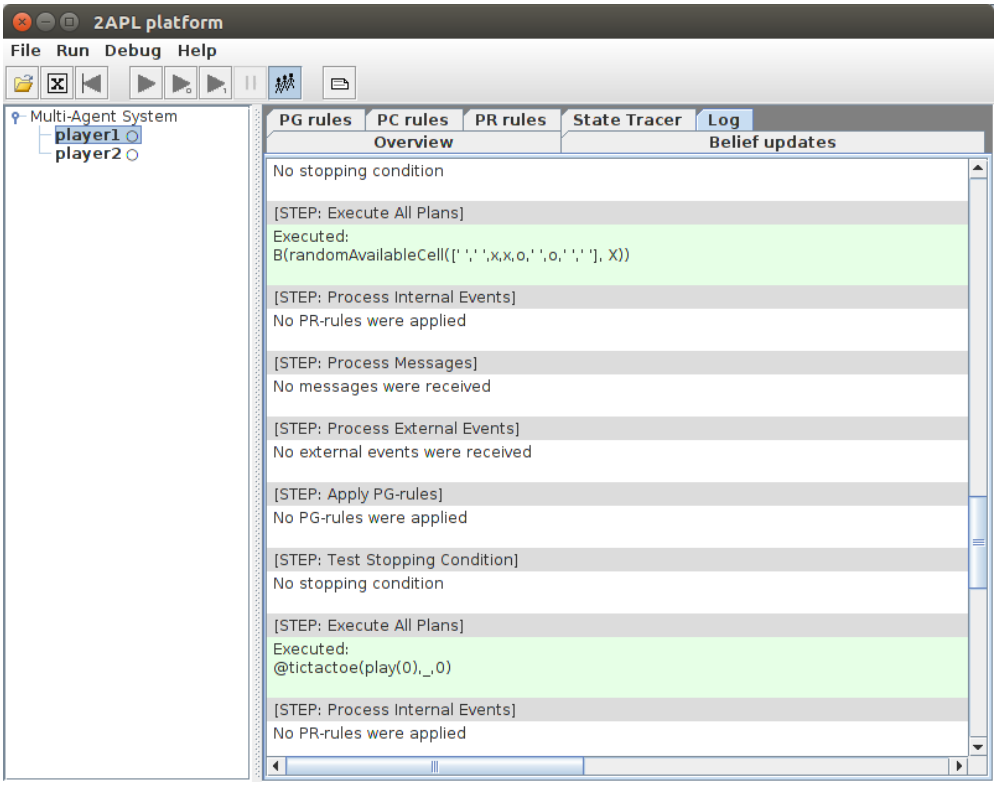

Figure 5. 2APL platform Log interface

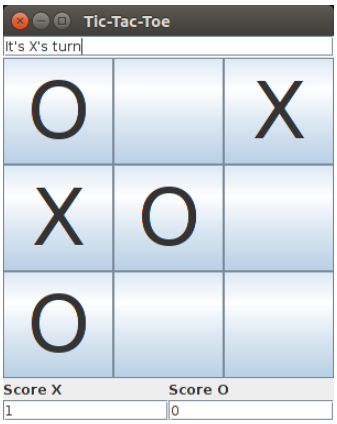

Figure 6. GUI of our *Tic-Tac-Toe* demonstration

6. Example

For the sake of illustration we have designed and implemented a small demonstration of 2APL. Namely we have worked in a small *Tic-Tac-Toe* game in which two agents (from now on *the players*) play against each other. Our application is shown in figure 6. The purpose of this section is to put into context some of the concepts previously described, giving relevant examples of applications where they are useful. For full insight about our implementation we refer the reader to the repository where the code is hosted[1].

6.1. Description of the MAS

Our MAS consists of two identical *players* with the same initial beliefs, PG rules, PC rules and PR rules (detailed below). We have designed and implemented the *player* as well as the environment. The `.mas` file of our configuration is shown in the following snippet.

```
<apaplmas>
    <environment name="tictactoe" file="Environment2APL.jar"/>

    <agent name="player1" file="player.2apl"/>
    <agent name="player2" file="player.2apl"/>
</apaplmas>
```

Listing 9. Configuration of the MAS

The snippet is quite self-explanatory. In essence we can see how an environment is defined under the name of `tictactoe` and, after that, the inclusion of two agents of the same type (`player1` and `player2`).

6.2. Initial beliefs of the player agent

The game's board (the 3×3 grid of the Tic-Tac-Toe) is represented with a list of 9 elements: each position of the list represents the contents of a board cell from left to right and from top to bottom. On the other hand, moves are represented simply as an index: the next position of the list to be filled by the player. Bearing this representation in mind we have elaborated a set of Prolog predicates for inferring facts such as whether there exists a winning move for one of the players or to gather together the empty cells (available moves) in a list. Because of this, the initial amount of predicates and rules in the belief database of the agents is large compared to the examples by the 2APL development group and we will not cover them all in this document. Nonetheless let us analyze some illustrative examples to get an idea of what can we find in the agent's beliefs:

```
beliefs:
    equal(X, X, X, X).

    ...

    win(Player, [A1, A2, A3, B1, B2, B3, C1, C2, C3]):- equal(A1, A2, A3, Player);
```

[1] `https://bitbucket.org/cmirallesp/mai-masd`

```
                                                    equal(B1, B2, B3, Player);
                                                    equal(C1, C2, C3, Player);
                                                    equal(A1, B1, C1, Player);
                                                    equal(A2, B2, C2, Player);
                                                    equal(A3, B3, C3, Player);
                                                    equal(A1, B2, C3, Player);
                                                    equal(A3, B2, C1, Player).
    win(Player):-
        board(Board),
        win(Player, Board).

    ...
```

Listing 10. Beliefs of the player agent

Let us observe for instance the `win/2` predicate. It has as arguments a player and a board, and it is true whenever the board represents a winning position for the given player (i.e. three aligned cells filled by the player). Notice that we do this check thanks to the `equal` predicate and the inner unification mechanism of the Prolog engine. On the other hand, we have the `win/1` predicates, which uses to the 2-argument version, grounding the board with the current (most updated) one.

6.3. Reasoning rules

Initially we programmed the Minimax algorithm as a method in the environment, accessible by the agents. The method provides the best possible move (a random one if there are several optimal moves) for the current player. Needless to say, if both Tic-Tac-Toc players follow a perfect strategy, neither of them will win. Therefore we have came up with an alternative strategy that tries to perform the first action available from the following list:

1. If the board is empty (first turn) fill (randomly) one of the corners or the center.
2. If there exists a winning move, execute it.
3. If there exists a winning move for the opponent, block it.
4. Fill randomly one of the available cells.

Next we describe some of the rules we have implemented to follow such strategy, using the tools provided to us by 2APL. These are, namely, the PG rules, the PC rules and the PR rules. Also, by means of these rules we will see in context several of the different types of action that there exist in 2APL.

Each time a player makes a move, the environment triggers an event called `newTurn/2`. The event has as arguments the current configuration of the board (already as a list) and the player that gets to move next. This event is processed with a PC rule as follows:

```
pcrules:
    event(newTurn(Board, Turn), tictactoe) <- true |
    {
        UpdateGame(Board, Turn);
        B(me(Player));
        if B(emptyBoard(Board))
        {
            adoptz(win(Player));
        }
    }
```

Listing 11. PC rules

The effect of applying the plan imposed by this rule is retracting from the belief database the previous outdated board and add the current one. At the same time, we update the player turn. This two effects are accomplished via the `UpdateGame` **belief update** action, that takes the following form:

```
beliefUpdates:
    { board(OldBoard), turn(OldTurn) } % Preconditions
    UpdateGame(NewBoard, NewTurn) % Action
    { not board(OldBoard), not turn(OldTurn), board(NewBoard), turn(NewTurn) } % Postconditions

    ...
```

Listing 12. `UpdateGame` belief update action

The rule also shows an example of a **test action**, `B(me(Player))`, that is employed to ground the Player variable to the current player (either `'x'` or `'o'`). There is an example of a **dynamical goal action**, `adoptz(win(Player))`. This action is performed each time the board is empty (presumably new game), since the `win` goal may have been accomplished previously (goals are not added twice if they have been previously added to the goal stack).

Next we cover an example PG rule (plan for achieving the goals):

```
pgrules:
    win(Player) <- turn(Player) |
    {
        B(board(Board));
        B(winningMove(Player, Board, X));
        @tictactoe(play(X), _);
        PassTurn();
    }

    ...
```

Listing 13. PG rules

Contrarily to the previous rule, here the query is not unconditionally true. Instead, it is required that it is the player's turn to move. The idea of the plan is to perform the next move. More specifically, the plan tries to bound the next move to one that guarantees the victory of the player. Of course, this is not always possible. We explain below how we take this into consideration with the PR rules. Let us notice that we have introduced an external action in this example, namely `play` (which, needless to say, is implemented in the environment). The purpose of this action is to act upon the GUI's board. `PassTurn` is a belief update action that transfers the turn to the opponent so the rule does not fire again until the opponent makes its move.

Finally we review some of the PR rules:

```
prrules:
  ...

  B(winningMove(Opponent, Board, X)); REST; <- me(Player) and opponent(Player, Opponent) |
    {
        if B(emptyBoard(Board)) then
        {
            % Choose the first movement from the corners and the center.
            B(randomChoice([0, 2, 4, 6, 8], X));
        }
        else
        {
            B(randomAvailableCell(Board, X));
        }
        REST;
    }

    B(winningMove(Player, Board, X)); REST; <- me(Player) |
    {
        B(opponent(Player, Opponent));
        B(winningMove(Opponent, Board, X));
        REST;
    }
```

Listing 14. PR rules

While no more types of action are introduced here, it is very interesting to observe how we make use of the language's PR mechanism to complete the implementation of the previously proposed strategy. The second rule fires when a `winningMove(Player, Board, X)` action with `Player` grounded to the same variable that satisfies `me(Player)` (i.e. not the opponent). In such a case, it tries to block a winning movement of the adversary. Similarly, the first rule fires when the opponent does not have a potential immediate victory, and in such case it generates a plan to either choose randomly a corner or the center (if the board is empty) or simply a random cell from all the available ones.

6.4. Environment

In order to implement a GUI that shows the moves of the agents and the current state, we have implemented an environment following the 2APL conventions. Our environment implements the following actions:

- `Term acquirePlayerId(String agName)`: since both players share the same definition, we cannot hard-code the me/1 predicate with the appropriate symbol. Instead, the symbol (player id, either `'o'` or `'x'`) is assigned dynamically by the environment when the agents execute this action. Each agent executes this action just once at the beginning.

- `Term play(String agName, APLNum idx)`: fills the *idx*th cell of the board, if possible. This method throws an ExternalActionFailedException only if the move is illegal (i.e. index out of bounds or already filled cell). Since the agents select their next move among the empty cells, this never happens. The method does not return any useful value (`null`).

On the other hand, the environment fires the `newTurn` event after each move, alerting both players. The event is fired after a fixed delay (e.g. 2s) so we have time to appreciate the evolution of the game.

7. Comparison with other alternatives

In this section we compare five different agent-oriented programming languages to 2APL: 3APL, Jack, Jadex, Jason and KGP. This section is heavily based on the comparison presented in [1]. Instead of explaining specifics of each language we mainly focus on differences between each language and 2APL. All the languages except Jack and Jadex use explicit formal semantics. Since Jack and Jadex share same differences compared to 2APL, they are allocated in the same section.

7.1. 2APL vs 3APL

2APL is a modified version of its ancestor 3APL. So naturally, those two languages have a lot in common. The main difference between them is that 3APL is a language for single agents while 2APL focuses on the whole multi-agent system. Moreover, 3APL has a different representation of an agent state. It consists of beliefs and plans, where plans consist of belief update, test, and abstract actions composed by the sequence and choice operator [1].

2APL complemented 3APL's plan revision rules with a set of external environments, the access relation between agents and environments, events, goals, and a variety of action types such as external actions, goal related actions, and communication action [1]. Also two additional rules are introduced in 2APL to implement reactive and proactive behavior of an agent. Additionally, in 2APL only failed plans can be revised in contrast to 3APL, where any plan can be change at any time. Finally, in 3APL at each cycle plans are executed and revised while in 2APL at each cycle goals, (internal and external) events, and messages are processed as well [2].

7.2. 2APL vs Jack and Jadex

The main difference between Jack and Jadex vs 2APL is that the first two have different semantic of the BDI concepts. Agents in Jack and Jadex are not capable of generating plans that contribute to the main goal without necessarily achieving it. In other words, unlike in 2APL they can't form plans that achieve the goal only partially. The reason for this is that agents in Jack and Jadex can't reason about their beliefs and goals due to the lack of logical semantics [1].

Another major difference between the languages is in the consistency of an agent's state. In Jack and Jadex it is left to an agent programmer to make sure that state updates preserve the state consistency. Finally, in 2APL beliefs and goals are used to generate a plan. They are used in a logical formula while in Jack and Jadex beliefs and goals are not related. Instead they use events and goals to generate plans [1].

7.3. 2APL vs Jason

In contrast to 2APL, Jason doesn't support direct declarative goals implementation, instead goals can be indirectly simulated with a pattern of plans. The beliefs of a Jason agent is a set of literals with strong negation, while in 2APL an agents beliefs is a set of Horn clauses. Additionally, one may annotate extra information to the beliefs and plans that can be used in belief queries and plan selection process. In 2APL the same information can be encoded in beliefs and plans themselves. Finally, in 2APL there are two additional rule types for creating plans to achieve declarative goals and repair plans when their executions fail. In Jason, plan failure can be modeled differently. So-called deletion events (and plans reactions to them) serve this purpose [1].

7.4. 2APL vs KGP

While KGP model aims at presenting an agent model with a variety of capabilities such as planning and goal decision, 2APL aims at providing a more expressive representation for the agents internal state [1]. The KGP model is based on

propositional language unlike 2APL, which use the first-order representation for beliefs, goals and plans components. KGP agents plan is a partially ordered set of primitive actions, while 2APL plans consists of actions that are composed by a variety of complex composition operators such as conditional choice, iteration, and non-interleaving operators. 2APL provides also a larger variety of actions such as belief and goal update actions, test actions, and external actions [1].

References

[1] M. Dastani. 2apl: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248, 2008.

[2] M. Dastani and B. R. Steunebrink. Operational semantics for bdi modules in multi-agent programming. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 83–101. Springer, 2009.