A Practical Agent Programming Language

Mehdi Dastani and John-Jules Ch. Meyer

Utrecht University
The Netherlands
{mehdi,jj}@cs.uu.nl

Abstract. This paper discusses the need for an effective and practical BDI-based agent-oriented programming language with formal semantics. It proposes an alternative by presenting the syntax and semantics of a programming language, called 2APL (A Practical Agent Programming Language). This programming language facilitates the implementation of multi-agent systems consisting of individual cognitive agents. 2APL distinguishes itself from other BDI-based agent-oriented programming languages by having formal semantics while realising an effective integration of declarative and imperative style programming. This is done by introducing a set of practical programming constructs, including both declarative goals and events (which are used interchangeably in other programming languages), and specifying their operational semantics.

1 Introduction

Existing BDI-based agent-oriented programming languages such as Jason[2], Jack[8], Jadex[7], and 3APL[3,5] provide programming constructs and mechanisms that allow direct implementation of software agents in terms of BDI concepts. These programming languages differ from each other as they facilitate the implementation of different but overlapping sets of agent concepts. For example, they all share programming constructs to support the implementation of an agent's beliefs and plans. However, 3APL differs from other programming languages as it supports the implementation of declarative goals as well as plan revision mechanism, but lacks programming constructs to support the implementation of events and event handling mechanism. Although, a comparison between these BDI-based programming languages is outside the scope of this paper, we would like to emphasize that some of the concepts that are shared by these languages have different semantics or even are not comparable at all. For example, the beliefs and goals in 3APL (and the beliefs in Jason) are propositions having declarative semantics while the beliefs in Jack and Jadex can be represented by conventional data structures lacking a formal semantics. The situation gets even worse because different languages use different concepts with the same name or they use the same concept with different names. For example, although both Jason and 3APL use the concept of goal, a goal in Jason is an event that triggers a plan while a goal in 3APL is a proposition that can be reasoned with [3,4]. The declarative nature of goals in 3APL allows the implementation of generic planning rules that assign plans to *subgoals*, i.e., goals that are derivable from the goal base.

Our experience with using these agent-oriented programming languages in various academic courses and research projects have shown that some of the existing programming constructs, tools, and mechanisms are indeed useful, expressive and effective in programming software agents. It also made it clear that new programming constructs, mechanisms and tools are needed to make the BDI-based programming languages more expressive and *practical*. We have also learned that the use and a deep understanding of these programming languages are significantly improved by the availability of their formal semantics. In our opinion, a challenge of a practical BDI-based agent-oriented programming language is 1) to realise an effective integration of declarative and imperative style programming, and 2) to have formal semantics. The declarative style programming should facilitate the implementation of the mental state of agents allowing agents to reason about their mental states and update them accordingly. The imperative style programming should facilitate the implementation of processes, the flow of control, and mechanisms such as procedure call, recursion, and interface to existing imperative programming languages.

In this paper, we present a BDI-based agent-oriented programming language called 2APL (A *Practical* Agent Programming Language). The main motivation to design and develop 2APL is an *effective integration* of programming constructs that support the implementation of declarative concepts such as belief and goals with imperative style programming such as events and plans. Like most BDI-based programming languages, different types of actions such as belief and goal update actions, test actions, external actions, and communication actions are distinguished. These actions are composed by conditional choice operator, iteration operator, and sequence operator. The composed actions constitute the plans of the agents. Like the existing agent programming languages, 2APL provides rules to indicate that a certain goal can be achieved by a certain pre-compiled plan. Agents may select and apply such rules to generate plans to achieve their goals.

As agents may operate in dynamic environments, they have to observe their environmental changes. In 2APL environmental changes will be notified to the agents by means of *events*. It should be noted that in some agent programming languages events are used for various purposes. For example, a goal in Jason is an event, while in Jadex events are generated to notify an agent's internal changes as well. In our view, although both goals and events cause an agent to execute actions, there are fundamental differences between them. For example, an agent's goal denotes a desirable state for which the agent performs actions to achieve it, while an event carries information about (environmental) changes which may cause an agent to react and execute certain actions. After the execution of actions, an agent's goal may be dropped if the state denoted by it is achieved, while an event can be dropped just before (or immediately after) executing the actions that are triggered by it. Moreover, because of the declarative nature of goals, an agent can reason about its goals while an event only carries information which is not necessarily the subject of reasoning.

Some characterizing 2APL features are related to the constructs designed with respect to an agent's plans. The first construct is a part of an exception handling mechanism allowing a programmer to specify how an agent should repair its plans when their executions fail. This construct has the form of a rule which indicates how a plan should be repaired. It is similar to the plan revision rules introduced in 3APL, but it differs from it as 2APL rules can only be applied to repair *failed* plans. In contrast, the plan revision rules of 3APL is applied to all plans continuously. In our view, it does not make sense to modify a plan if the plan is correct and executable. The second 2APL programming construct with respect to plans is related to the so-called atomic plans. In most agent-oriented programming languages, an agent can have various plans whose executions can be interleaved. The arbitrary interleaving of plans may be problematic in some cases such that a programmer may want to indicate that a certain part of a plan should be executed atomically at once without being interleaved with the actions of other plans. Similar construct is also introduced in Jadex[7].

In the following sections, we first present the complete syntax of 2APL and discuss the intuitive meaning of its ingredients. Then, because of space limitation, the formal semantics of only some characterizing programming constructs of 2APL is presented. We conclude the paper by discussing the implementation of 2APL interpreter and its corresponding platform.

2 2APL: Syntax

This section presents the complete syntax of 2APL, which is specified using the EBNF notation. In this specification, illustrated in Figure 1, we use $\langle atom \rangle$ to denote a Prolog like atomic formula starting with lowercase letter, $\langle Atom \rangle$ to denote a Prolog like atomic formula starting with a capital latter (parentheses are required for such formula with zero argument), $\langle ident \rangle$ to denote a string and $\langle Var \rangle$ to denote a string starting with a capital letter. We use $\langle ground\_atom \rangle$ to denote a ground atomic formula. An individual 2APL agent may be composed of various ingredients that specify different aspects of the agency. A 2APL agent can be programmed by implementing the initial state of those ingredients. In the following, we discuss each ingredients and give examples to illustrate them.

2.1 Beliefs and Goals

A 2APL agent may have beliefs and goals which change during the agent's execution. The *beliefs* of the agents are implemented by the belief base, which contains information the agent believes about its surrounding world including other agents. The implementation of the initial belief base starts with the keyword 'Beliefs:' followed by one or more belief expressions of the form $\langle belief \rangle$.

```
Beliefs:
  pos(1,1).
  hasGold(0).
  trash(2,5).
  trash(6,8).
  clean(blockWorld) :- not trash(_,_).
```

Note that a $\langle belief \rangle$ expression is a Prolog fact or rule such that the belief base of a 2APL agent becomes a Prolog program. All facts are assumed to be ground. The example above illustrates the implementation of the initial belief base of a 2APL agent. This

```
⟨2APL_Prog⟩        ::= (”Include:” ⟨ident⟩
                    |    ”BeliefUpdates:” ⟨BelUpSpec⟩
                    |    ”Beliefs:” ⟨belief⟩
                    |    ”Goals:” ⟨goals⟩
                    |    ”Plans:” ⟨plans⟩
                    |    ”PG-rules:” ⟨pgrules⟩
                    |    ”PC-rules:” ⟨pcrules⟩
                    |    ”PR-rules:” ⟨prrules⟩)*
⟨BelUpSpec⟩        ::= ( ”{”⟨belquery⟩ ”}” ⟨beliefupdate⟩ ”{”⟨literals⟩”}” )+
⟨belief⟩           ::= ( ⟨ground_atom⟩ ”.” | ⟨atom⟩ ”: −” ⟨literals⟩”.” )+
⟨goals⟩            ::= ⟨goal⟩ (”,” ⟨goal⟩)*
⟨goal⟩             ::= ⟨ground_atom⟩ (”and” ⟨ground_atom⟩)*
⟨baction⟩          ::= ”skip” | ⟨beliefupdate⟩ | ⟨sendaction⟩ | ⟨externalaction⟩
                    |    ⟨abstractaction⟩ | ⟨test⟩ | ⟨adoptgoal⟩ | ⟨dropgoal⟩
⟨plans⟩            ::= ⟨plan⟩ (”,” ⟨plan⟩)*
⟨plan⟩             ::= ⟨baction⟩ | ⟨sequenceplan⟩ | ⟨ifplan⟩ | ⟨whileplan⟩ | ⟨atomicplan⟩
⟨beliefupdate⟩     ::= ⟨Atom⟩
⟨sendaction⟩       ::= ”send(” ⟨iv⟩ ”,” ⟨iv⟩ ”,” ⟨atom⟩ ”)”
                    |    ”send(” ⟨iv⟩ ”,” ⟨iv⟩ ”,” ⟨iv⟩ ”,” ⟨iv⟩ ”,” ⟨atom⟩ ”)”
⟨externalaction⟩   ::= ”@” ⟨iv⟩”(” ⟨atom⟩ ”,” ⟨Var⟩ ”)”
⟨abstractaction⟩   ::= ⟨atom⟩
⟨test⟩             ::= ”B(” ⟨belquery⟩ ”)” | ”G(” ⟨goalquery⟩ ”)” | ⟨test⟩ ”&” ⟨test⟩
⟨adoptgoal⟩        ::= ”adopta(” ⟨goalvar⟩ ”)” | ”adoptz(” ⟨goalvar⟩ ”)”
⟨dropgoal⟩         ::= ”dropGoal(” ⟨goalvar⟩ ”)” | ”dropSubgoal(” ⟨goalvar⟩ ”)”
                    |    ”dropExactgoal(” ⟨goalvar⟩ ”)”
⟨sequenceplan⟩     ::= ⟨plan⟩ ”;” ⟨plan⟩
⟨ifplan⟩           ::= ”if” ⟨test⟩ ”then” ⟨scopeplan⟩ (”else” ⟨scopeplan⟩)?
⟨whileplan⟩        ::= ”while” ⟨test⟩ ”do” ⟨scopeplan⟩
⟨atomicplan⟩       ::= ”[” ⟨plan⟩ ”]”
⟨scopeplan⟩        ::= ”{” ⟨plan⟩ ”}”
⟨pgrules⟩          ::= ⟨pgrule⟩+
⟨pgrule⟩           ::= ⟨goalquery⟩? ”< −” ⟨belquery⟩ ”|” ⟨plan⟩
⟨pcrules⟩          ::= ⟨pcrule⟩+
⟨pcrule⟩           ::= ⟨atom⟩ ”< −” ⟨belquery⟩ ”|” ⟨plan⟩
⟨prrules⟩          ::= ⟨prrule⟩+
⟨prrule⟩           ::= ⟨planvar⟩ ”< −” ⟨belquery⟩ ”|” ⟨planvar⟩
⟨goalvar⟩          ::= ⟨atom⟩(”and”⟨atom⟩)*
⟨planvar⟩          ::= ⟨plan⟩ | ⟨Var⟩ | ”if” ⟨test⟩ ”then” ⟨scopeplanvar⟩ (”else” ⟨scopeplanvar⟩)?
                    |    ”while” ⟨test⟩ ”do” ⟨scopeplanvar⟩ | ⟨planvar⟩ ”;” ⟨planvar⟩
⟨scopeplanvar⟩     ::= ”{” ⟨planvar⟩ ”}”
⟨literals⟩         ::= ⟨literal⟩ (”,” ⟨literal⟩)*
⟨literal⟩          ::= ⟨atom⟩ | ”not” ⟨atom⟩
⟨ground_literal⟩   ::= ⟨ground_atom⟩ | ”not” ⟨ground_atom⟩
⟨belquery⟩         ::= ”true” | ⟨belquery⟩ ”and” ⟨belquery⟩ | ⟨belquery⟩ ”or” ⟨belquery⟩
                    | ”(” ⟨belquery⟩ ”)” | ⟨literal⟩
⟨goalquery⟩        ::= ”true” | ⟨goalquery⟩ ”and” ⟨goalquery⟩ | ⟨goalquery⟩ ”or” ⟨goalquery⟩
                    | ”(” ⟨goalquery⟩ ”)” | ⟨atom⟩
⟨iv⟩               ::= ⟨ident⟩ | ⟨Var⟩
```

Fig. 1. The EBNF syntax of 2APL

belief base represents the information of an agent about its `blockworld` environment. In particular, the agent believes that its position in this environment is `(1,1)`, it has no gold item in possession, there are trash at positions `(2,5)` and `(6,8)`, and that the `blockworld` environment is clean if there are no trash anymore.

The *goals* of a 2APL agent are implemented by its goal base, which is a list of formulas each of which denotes a situation the agent wants to realize (not necessary all at once). The implementation of the initial goal base starts with the keyword '`Goals:`' followed by a list of goal expressions of the form ⟨*goal*⟩. Each goal expression is a conjunction of ground atoms. Note that a ground atom is treated as a Prolog fact. The following example is the implementation of the initial goal base of a 2APL agent. This goal base indicates that the agent wants to achieve a desirable situation in which it has five gold items and the `blockworld` is clean. Note that this single conjunctive goal is different than having two separate goals '`hasGold(5)`' and '`clean(blockworld)`'. In the latter case, the agent wants to achieve two desirable situations independently of each other, i.e., one in which the agent has a clean `blockworld`, not necessarily with a gold item, and one in which it has five gold items and perhaps a `blockworld` which is not clean. Note that different goals in the goal base are separated by a comma.

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

The beliefs and goals of agents are related to each other. In fact, if an agent believes a certain fact, then the agent does not pursue that fact as a goal. This means that if an agent modifies its belief base, then its goal base may be modified as well.

2.2 Basic Actions

Basic actions specify the capabilities that an agent can perform to achieve its desirable situation. The basic actions will constitute an agent's plan, as we will see in the next subsection. In 2APL, six types of basic actions are distinguished: actions to update the belief base, communication actions, external actions to be performed in an agent's environment, abstract actions, actions to test the belief and goal bases, and actions to manage the dynamics of goals.

Belief Update Action. A *belief update action* updates the belief base of an agent when executed. A belief update action ⟨*beliefupdate*⟩ is an expression of the form ⟨*Atom*⟩ (i.e., a first-order atom in which the predicate starts with a capital letter). Such an action is specified in terms of pre- and post-conditions. An agent can execute a belief update action if the pre-condition of the action is derivable from its belief base. The pre-condition is a formula consisting of literals composed by disjunction and conjunction operators. The execution of a belief update action modifies the belief base in such a way that after the execution the post-condition of the action is derivable from the belief base. The post-condition of a belief update action is a list of literals. The update of the belief base by such an action removes the atom of the negative literals from the belief base and adds the positive literals to the belief base. The specification of the belief update actions starts with the keyword '`BeliefUpdates:`' followed by the specifications of a set of belief update actions ⟨*BelUpSpec*⟩.

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

Above is an example of the specification of the belief update actions. In this example, the specification of the `PickUp()` indicates that this belief update action can be performed if the agent does not already carry gold items and that after performing this action the agent will carry one gold item. The agent is assumed to be able to carry only one gold item. Note that the agent cannot perform two `PickUp()` action consecutively. Note also the use of variables in the specification of `ChgPos(X1,Y1)`. It requires that an agent can change its current position to `(X1,Y1)` if its current position is `(X,Y)`. After the execution of this belief update action, the agent believes that its position is `(X1,Y1)` and not `(X,Y)`. Note also that variables in the post-conditions are bounded since otherwise the facts in the belief base will not be ground.

Communication Action. A *communication action* passes a message to another agent. A communication action ⟨*sendaction*⟩ can have either three or five parameters. In the first case, the communication action is the expression `send(Receiver, Performative, Language, Ontology, Content)` where `Receiver` is a name referring to the receiving agent, `Performative` is a speech act name (e.g. inform, request, etc.), `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. It is often the case that agents assume a certain language and ontology such that it is not necessary to pass them as parameters of their communication actions. The second version of the communication action is therefore the expression `send(Receiver,Performative,Content)`. 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` where `localname` is the name as used by 2APL, `host` is the name of the host running the agent's container and `port` is the port number where the agent's container, should listen to (see [1] for more information on JADE standards).

External Action. An *external action* is supposed to change the external environment in which agents operate. The effects of external actions are assumed to be determined by the environment and might not be known to the agents beforehand. An agent thus decides to perform an external action and the external environment determines the effect of the action. The agent can know the effects of an external action by performing a sense action (also defined as an external action), by means of events generated by the environment, or by means of a return parameter. An external action ⟨*externalaction*⟩ is an expression of the form `@Env(ActionName,Return)`. The parameter `Env` is the name of the agent's environment, implemented as a Java class. The parameter `ActionName` is a method call (of the Java class) that specifies the effect of the external action in the environment. The environment is assumed to have a state represented by the instance variables of the class. The execution of an action in an environment is then a read/write operation on the state of the environment. The parameter `Return` is a list of values, possibly an empty list, returned by the corresponding method. An example of the implementation of an external action is `@blockworld(east(),L)` (go one step east in the blockworld environment). The effect of this action is that the position of the agent in the blockworld environment is shifted one slot to the right. The list `L` is expected as the return value.

Abstract Action. The general idea of an abstract action is similar to a procedure call in imperative programming languages. The procedures should be defined in 2APL by means of the co-called PC-rules, which stands for procedure call rules (see subsection 2.4 for a description of PC-rules). As we will see in subsection 2.4, a PC-rule can be used to associate a plan to an abstract action. The execution of 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 to the abstract action by a PC-rule. Like a procedure call in imperative programming languages, an abstract action ⟨*abstractaction*⟩ is an expression of the form ⟨*atom*⟩ (i.e. a first order expression in which the predicate starts with a lowercase letter). An abstract action can be used to pass parameters from one plan to another plan. In particular, the execution of an abstract action passes parameters from the plan in which it occurs to another plan that is associated to it by a PC-rule.

Test Actions. A *test action* is to check whether the agent has certain beliefs and goals. A test action is an expression of the form ⟨*test*⟩ consisting of belief and goal query expressions. A belief query expression has the form `B(`ϕ`)`, where ϕ consists of *literals* composed by conjunction and disjunction operators. A goal query expression has the form `G(`ϕ`)`, where ϕ consists of *atoms* composed by conjunction and disjunction operators.

A belief query expression, which constitutes a test action, is basically a (Prolog) query to the belief base and generates a substitution for the variables that are used in the belief query expression. A goal query expression, which also constitutes a test action, is a query to an individual goal in the goal base, i.e., it is to check if there is a goal in the goal base that satisfies the query. Such a query may also generate a substitution for the involved variables.

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). The instantiation of variables in a test action is determined through belief and goal queries performed from left to the right. For example, let an agent believes `p(a)` and has the goal `q(b)`, then the test action `B(p(X)) & G(q(X))` fails, while the test action `B(p(X)) & G(q(Y) or r(X))` succeeds with {`X/a , Y/b`} as the resulting substitution.

Goal Dynamics Actions. The *adopt goal* and *drop goal* actions are used to adopt and drop a goal to and from an agent's goal base, respectively. The adopt goal action ⟨*adoptgoal*⟩ can have two different forms: `adopta(`ϕ`)` and `adoptz(`ϕ`)`. These two actions can be used to add the goal ϕ (a conjunction of atoms) to the begin and to the end of an agent's goal base, respectively. Note that the programmer has to ensure that the variables in ϕ are instantiated before these actions are executed since the goal base should contain only ground formula. Finally, the drop goal action $\langle dropgoal \rangle$ can have three different forms: `dropGoal(`ϕ`)`, `dropSubGoal(`ϕ`)`, and `dropExactGoal(`ϕ`)`. These actions can be used to drop from an agent's goal base, respectively, all goals that are a logical subgoal of ϕ, all goals that have ϕ as a logical subgoal, and exactly the goal ϕ, respectively. Similar actions are proposed in [6].

2.3 Plans

In order to reach its goals, a 2APL agent adopts *plans*. A plan consists of basic actions composed by some process composition operators. In particular, basic actions can be composed by means of the sequence operator, conditional choice operators, conditional iteration operator, and an unary operator to identify atomic plans.

The sequence operator `;` is a binary operator that takes two plans and generates one $\langle sequenceplan \rangle$ plan. The sequence operator indicates that the first plan should be performed before the second plan. The conditional choice operator generates $\langle ifplan \rangle$ plans of the form `if` ϕ `then` π_1 `else` π_2, where π_1 and π_1 are arbitrary plans. The condition part of this expression (i.e., ϕ) is a test that should be evaluated with respect to an agent's belief and goal bases. Such a plan can be interpreted as to perform the if-part of the plan (i.e., π_1) when the test ϕ succeeds, otherwise perform the else-part of the plan (i.e., π_2). The conditional iteration operator generates $\langle whileplan \rangle$ plans of the form `while` ϕ `do` π, where π is an arbitrary plan. The condition ϕ is also a test that should be evaluated with respect to an agent's belief and goal bases. The iteration expression is then interpreted as to perform the plan π as long as the test ϕ succeeds.

The last unary operator generates $\langle atomicplan \rangle$ plans, which are expressions of the form `[`π`]`, where π is an arbitrary plan. This plan is interpreted as an atomic plan π, which should be executed at once ensuring that the execution of π is not interleaved with the actions of other plans. Note that an agent can have different plans at the same time and that plans cannot be composed by an explicit parallel operator. As there is no explicit parallel composition operator, the nested application of the unary operator has no effect, i.e., the executions of plans `[`π_1`;`π_2`]` and `[`π`;[`π_2`]]` result identical behaviors.

The plans of a 2APL agent are implemented by its plan base. The implementation of the initial plan base starts with the keyword '`Plans:`' followed by a list of plans. The following example illustrates the 2APL implementation of the initial plan base of an agent. The first plan is an atomic plan ensuring that the agent updates its belief base with its initial position `(5,5)` immediately after performing the external action `enter` in the `blockworld` environment. The second plan is a single action by which the agent requests the administrator to register him.

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

2.4 Reasoning Rules

The 2APL programming language provides constructs to implement practical reasoning rules that can be used to implement the generation of plans. In particular, three types of practical reasoning rules are proposed: planning goal rules, procedure call rules, and plan repair rules. In the following subsections, we explain these three types of rules.

Planning Goal Rules (PG-rules). A planning goal rule can be used to implement an agent that generates a plan if it has certain goals and beliefs. The specification of a planning goal rule ⟨*pgrule*⟩ consists of three entries: the head of the rule, the condition of the rule, and the body of the rule. The head and the condition of a planning goal rule are query expressions used to check if the agent has a certain goal and belief, respectively. The body of the rule consists of a plan in which variables may occur. These variables should be bound by the goal and belief expressions. A planning goal rule of an agent can be applied when the goal and belief expressions (in the head and the condition of the rule) are derivable from the agent's goal and belief bases, respectively. The application of a planning goal rule generates a substitution for variables that occur in the head and condition of the rule as they are queried from the goal and belief bases. The resulted substitution will be applied to the generated plan to instantiate it. A planning goal rule is of the form: ⟨*goalquery*⟩? "< −" ⟨*belquery*⟩ "|" ⟨*plan*⟩.

Note that the head of the rule is optional which means that the agent can generate a plan only based on its belief condition. The following is an example of a planning goal rule indicating that a plan to go to a position `(X2,Y2)`, departing from a position `(X1,Y1)`, to remove trash can be generated if the agent has the goal `clean( blockworld)` and it believes its current position is `pos(X1,Y1)` and there is trash at position `(X2,Y2)`.

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

The action `goTo(X1,Y1,X2,Y2)` in the above PG-rule is an abstract action (see subsection 2.4 for how to execute an abstract action). Note that this rule can be applied if (beside the satisfaction of the belief condition) the agent has a conjunctive goal `hadGold(5) and clean(blockworld)` since the head of the rule is derivable from this goal.

Procedure Call Rules (PC-rules). The procedure call rules is introduced for various reasons and purposes. Besides their use as procedure definition (used for executing abstract actions), they can also be used to respond to messages and handle external events. In fact, a procedure call rule can be used to generate plans as a response to the reception of messages send by other agents, events generated by the external environment, and the execution of abstract actions. Like planning goal rules, the specification of procedure call rules consist of three entries. The only difference is that the head of the procedure call rules is an atom ⟨*atom*⟩, rather than a goal query expression ⟨*goalquery*⟩. The head of a PC-rule can be a message, an event, or an abstract action. A message and an event are represented by atoms with the special predicates `message`/3 (`message`/5) and `event`/2, respectively. An abstract action is represented by any predicate name starting with a lowercase letter. Note that like planning goal rules, a procedure call rule has a belief condition indicating when a message (or event or abstract action) should generate a plan. Thus, a procedure call rule can be applied if the agent has received a message (or an event or executes an abstract action) and the belief query of the rule is derivable from its belief base. The resulted substitution for variables are applied in order to instantiate the generated plan. A procedure call rule ⟨*pcrule*⟩ is of the form: ⟨*atom*⟩ "< −" ⟨*belquery*⟩ "|" ⟨*plan*⟩. The following are examples of procedure call rules.

```
PC-rules:
  message(A,inform,La,On,goldAt(X2,Y2)) <-  not carry(gold) |
         {[  pos(X1,Y1)?; goTo(X1,Y1,X2,Y2);
             @blockworld(pickup(),_); PickUp();
             goTo(X2,Y2,3,3);
             @blockworld(drop(),_); AddGold()]
         }

  event(gold(X2,Y2),blockworld) <-  not carry(gold) |
        {[  pos(X1,Y1)?;goTo(X1,Y1,X2,Y2);
            @blockworld(pichup(),_);PickUp();
            goTo(X2,Y2,3,3);
            @blockworld(drop(),_);AddGold()]
        }

  goTo(X1,Y1,X2,Y2) <- X1 < X2 |
        {[  @blockworld(east(),_);ChgPos(X1+1,Y1);
          goTo(X1+1,Y1,X2,Y2)]
        }
```

The first rule indicates that if an agent `A` informs that there is some gold at position `(X2,Y2)` and the agent believes it does not carry any gold, then the agent has to go from its current position to the gold position, pick up the gold, go to the depot position (i.e. position `(3,3)`), and drop the gold in the depot. The `PickUp()` and `AddGold()` are belief update actions to administrate the facts that the agent is carrying gold and has certain amount of gold, respectively. The second rule indicates that if the environment `blockworld` notifies the agent that there is some gold at position `(X2,Y2)` and the agent believes it does not carry gold, then the abovementioned sequence of actions should take place. The generation of plans without a belief condition enables a programmer to implement reactive agent behavior, i.e., plans are generated if the agent is notified about an environmental change. Finally, the last rule indicates that the abstract action `goTo` should be performed as a certain sequence of actions. Note that all plans are implemented as atomic plans. The reason is that in these plans external actions and belief update actions are executed consecutively such that an unfortunate interleaving of actions can have undesirable effects. Note the use of recursion in this PC-rule.

Plan Repair Rules (PR-rules). Like other practical reasoning rules, a plan repair rule consists of three entries: two abstract plan expressions and one belief query expression. We have used the term abstract plan expression since such plan expressions include variables that can be substituted with a plan. A plan repair rule indicates that if the execution of an agent's plan (i.e., any plan that can be unified with the abstract plan expression) fails and the agent has a certain belief, then the failed plan should be replaced by another plan. A plan repair rule ⟨*prrule*⟩ has the following form: ⟨*planvar*⟩ "< −" ⟨*belquery*⟩ "|" ⟨*planvar*⟩.

A plan repair rule of an agent can thus be applied if 1) the execution of one of its plan fails, 2) the failed plan can be unified with the abstract plan expression in the head of the rule, and 3) the belief query expression is derivable from the agent's belief base. The satisfaction of these three conditions results in a substitution for the variables that occur in the abstract plan expression in the body of the rule. Note that some of these variables will be substituted with a part of the failed plan through the match between the abstract plan expression in the head of the rule and the failed plan. For example, if π, π_1, π_2 are plans and X is a plan variable, then the abstract plan $\pi_1; X; \pi_2$ can be unified with the failed plan $\pi_1; \pi; \pi_2$ resulting the substitution $X = \pi$. The resulted substitutions will be applied to the second abstract plan expression to generate the new (repaired) plan.

The following is an example of a plan repair rule. This rule indicates that if the execution of a plan that starts with `@blockworld(east(),_);@blockworld(east(),_)` fails, then the plan should be replaced by a plan in which the agent first goes one step to north, then makes two steps to east, and goes one step back to south. This repair can be done without a specific belief condition.

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

Note the use of the variable `X` that indicates that any failed plan starting with external actions `@blockworld(east(),_);@blockworld(east(),_)` can be repaired by the same plan in which the external actions are replaced by four external actions.

The question is when the execution of a plan fails. We consider the execution of a plan as failed if the execution of its first action fails. When the execution of an action fails depends on the type of action. The execution of a belief update action fails if the pre-condition of the action is not derivable from the belief base or if the action is not specified, an abstract action if there is no applicable procedure call rule, an external action if the corresponding environment throws an ExternalActionFailedException or if the agent has no access to that environment or if the action is not defined in that environment, a test action if the test expression is not derivable from the belief and goal bases, a goal adopt action if the goal is already derivable from the belief base, and an atomic plan if one of its actions fails. The execution of all other actions are always successful. When the execution of an action fails, then the execution of the whole plan is stopped. The failed action will not be removed from the failed plan such that it can be repaired.

2.5 External Environment

An agent can perform actions in different external environments that are implemented by a programmer as Java classes. Any Java class that implements the *environment interface* can be used as a 2APL environment. The environment interface contains two methods, *addAgent*(*String name*) and *removeAgent*(*String name*) to add/remove an agent to/from the environment, respectively . The constructor of the environment must require exactly one parameter of the type *ExternalEventListener*. This object listens to external events.

The execution of action `@Env(f(`$a_1, \ldots, a_n$`),R)` calls a method with name `f` in environment `Env` with arguments $a_1, \ldots, a_n$. The first argument a_1 is assumed to be the identifier of the agent that executes the action. The environment needs to have this identifier, for example, to pass information back to the agent by means of events. The second parameter `R` of an external action is meant to pass information back to the *plan* in which the external action was executed. Note that the execution of a plan is blocked until the method `f` is ready and the return value is accessible to the rest of the plan.

Methods may throw an exception (`ExternalActionFailedException`). If they throw an exception, the corresponding external action is considered as failed. The following is an example of a method that can be called as external action.

```
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();
}
```

2.6 Events and Exception

Information between agents and environments can be passed through *events* and *exceptions*. The main use of events is to pass information from environments to agents. When implementing a 2APL environment in Java, the programmer should decide when and which information from the environment should be passed to agents. This can be done by calling the method `notifyEvent(AF event, String... agents)` in the `ExternalEventListener`, which is an argument of the environments constructor. The first argument of this method may be any valid atomic formula. The rest of the arguments may be filled with strings that represents local names of agents. The events can be catched by agents whose name is included in the argument list to trigger one of their procedure call rules. If the programmer does not specify any agents in the argument list, all agents can catch the event. Such a mechanism of generating events by the environment and catching it by agents can be used to implement the agents' perceptual mechanism.

The exceptions in 2APL are used to apply plan repair rules. In fact, a plan repair rule is triggered when a plan execution fails. Exceptions are used to notify that the execution of a plan was not successful. The exception contains the identifier of the failed plan such that it can be determined which plan needs to be repaired. 2APL does not provide programming constructs to implement the generation and throwing of exceptions. In fact, exceptions are semantical entities that cannot be used by 2APL programmers.

3 2APL: Semantics

In the previous section, we described 2APL programming constructs and their intuitive meanings. In this section, we present the formal semantics of 2APL in terms of a transition system. A transition system is a set of derivation rules for deriving transitions. A transition is a transformation of one configuration into another and it corresponds to a single computation step. Because of the space limitation, we only present the configuration of 2APL agents, external actions, and characterizing 2APL constructs such as goal related constructs, atomic plan construct, and plan repair rules.

The configuration of an individual agent consists of its identifier, beliefs, goals, plans, specifications of belief update actions, reasoning rules, the substitutions resulted from queries to the belief and goal bases, and the received events. Since reasoning rules and the specification of belief update actions do not change during an agent's execution, we will not include them in the agent's configuration to keep the presentation as simple as possible. It should be noted that additional information is assigned to an agent's plan. In particular, an identifier is assigned to each plan which can be used to notify that the execution of the plan is failed. This is needed to identify and repair the plans the execution of which have failed. Moreover, the instantiation of the PG-rule through which a plan is generated is assigned to the plan. This information is used to avoid selecting a PG-rule to generate a plan if there is still a plan in the plan base that is generated by the same PG-rule and for the same goal.

Definition 1. *The configuration of a 2APL agent is defied as $A_\iota = \langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle$ where ι is a string representing the agent's identifier, σ is a set of belief expressions $\langle belief \rangle$ representing the agent's belief base, γ is a list of goal expressions $\langle goal \rangle$ representing the agent's goal base, Π is a set of plan entries consisting of $\langle plan \rangle$, enriched with additional information, representing the agent's plan base, θ is a ground substitution that binds domain variables to ground terms, and ξ is the agent's event base. 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. The agent's event base ξ is a tuple $\langle E, I, M \rangle$ where*

- *E is a set of events received from external environments. An event has the form $event(A, S)$, where A is a ground atom passed by the environment S.*
- *I is a set of identifiers denoting failed plans. An identifier represents an exceptions thrown because of a plan execution failure.*
- *M is the set of messages sent to the agent. Each message is of the form $message(s, p, l, o, \phi)$, where s is the sender identifier, p is a performative, l is the communication language, o is the communication ontology, and ϕ is a ground atom representing the message content.*

In the rest of this paper, we use $\models$ as a first-order entailment relation (we use Prolog engine for the implementation of this relation).

The configuration of a multi-agents system is defined in terms of the configuration of individual agents in the multi-agent system and their shared external environments.

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

The idea of a test action is to check if the belief and goal queries within a $\langle test \rangle$ expression are entailed by the agent's belief and goal bases. Moreover, as some of the variables that occur in the belief and goal queries may already be bound by the substitution θ, we apply the substitution to the $\langle test \rangle$ expression before testing it against the belief and goal bases. After applying θ, the test expression can still contain unbound variables to bind next occurrences of the variable in the plan in which it occurs. Therefore, the test action results a substitution τ which will be added to θ.

Definition 3. *Let φ and φ' be $\langle test \rangle$ expressions, ϕ be a $\langle belquery \rangle$ expression, ψ be a $\langle goalquery \rangle$ expression, and $\models_t$ be the entailment relation that evaluates test expressions with respect to an agent's belief and goal bases (σ, γ).*

- $(\sigma,\gamma) \models_t B(\phi)\tau \;\Leftrightarrow\; \sigma \models \phi\tau$
- $(\sigma,\gamma) \models_t G(\psi)\tau \;\Leftrightarrow\; \exists \gamma_i \in \gamma : \gamma_i \models \psi\tau$
- $(\sigma,\gamma) \models_t (\varphi \,\&\, \varphi')\tau_1\tau_2 \;\Leftrightarrow\; (\sigma,\gamma) \models_t \varphi\tau_1$ *and* $(\sigma,\gamma) \models_t \varphi'\tau_1\tau_2$

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

$$\frac{(\sigma,\gamma) \models_t \varphi\theta\tau}{\langle \iota, \sigma, \gamma, \{(\varphi, r, \_)\}, \theta, \_ \rangle \longrightarrow \langle \iota, \sigma, \gamma, \{\}, \theta \cup \{\tau\}, \_ \rangle}$$

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

$$\frac{\neg\exists\tau : (\sigma,\gamma) \models_t \varphi\theta\tau}{\langle \iota, \sigma, \gamma, \{(\varphi, r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\varphi, r, id)\}, \theta, \langle E, I \cup \{id\}, M\rangle\rangle}$$

The execution of an external action $@Env(\alpha(t_i, \ldots, t_n), V)$ has two different effects. The shared environments is changed and the variable V might be assigned to a term. To define the effect of an external action on the agent's state we define a function that returns a tuple containing the new state of the environments and the assignment for V. Let $F_\alpha^{Env}(t_1, \ldots, t_n, \chi)$ be the function that executes external action α with arguments $t_1, \ldots, t_n$ in the environment $Env \in \chi$ and returns a tuple (τ, χ'), where τ contains one substitution for the output variable V and χ' is the updated set of environments (a change in Env may change other environments in χ). A successful execution of an external action updates the agent's substitution θ and the set of shared environments χ. Note that because the environment is shared among agents, the transition for an external action of an individual agent is defined at the multi-agent level.

$$\frac{F_\alpha^{Env}(t_1\theta, \ldots, t_n\theta, \chi) = (t, \chi') \;\&\; t \neq \bot}{\langle A_1, \ldots, A_i, \ldots, A_n, \chi\rangle \longrightarrow \langle A_1, \ldots, A_i', \ldots, A_n, \chi'\rangle}$$

where
$A_i = \langle i, \sigma_i, \gamma_i, \{(@\mathtt{Env}(\alpha(\mathtt{t}_1, \ldots, \mathtt{t}_n), \mathtt{V}), \mathtt{r}, \mathtt{id})\}, \theta, \xi\rangle$ &
$A_i' = \langle i, \sigma_i, \gamma_i, \{\}, \theta \cup \{[\mathtt{V}/t]\}, \xi\rangle$

However, if the execution of an external action fails, then the environment Env generates an exception such that F_α^{Env} returns $(\bot, \chi')$. The failed action remains then in the plan base, the environments χ may be updated, and the event base ξ is updated to capture the failure exception.

$$\frac{F_{\alpha}^{Env}(t_1\theta,\ldots,t_n\theta,\chi) = (\bot,\chi')}{\langle A_1,\ldots,A_i,\ldots,A_n,\chi\rangle \longrightarrow \langle A_1,\ldots,A_i',\ldots,A_n,\chi'\rangle}$$

where
$A_i = \langle i,\sigma_i,\gamma_i,\{(\texttt{@Env}(\alpha(\texttt{t}_1,\ldots,\texttt{t}_\texttt{n}),\texttt{V}),\texttt{r},\texttt{id})\},\theta,\langle E,I,M\rangle\rangle$ &
$A_i' = \langle i,\sigma_i,\gamma_i,\{(\texttt{@Env}(\alpha(\texttt{t}_1,\ldots,\texttt{t}_\texttt{n}),\texttt{V}),\texttt{r},\texttt{id})\},\theta,\langle E,I\cup\{id\},M\rangle$

In order to achieve an agent's goal, plans should be generated and executed. The generation of plans is through application of planning goals rules. Applying PG-rules update only the plan base. Let $r = \kappa \leftarrow \beta \mid \pi$ be a PG-rule of the agent, P be the set of all possible plans, I be the set of all plan identifiers, $\gamma = [\gamma_1,\ldots,\gamma_i,\ldots,\gamma_n]$ be the agent's goal base, and $\kappa' \leftarrow \beta' \mid \pi'$ be a variant of r, i.e., all variables occurring in r are assumed to be fresh variables.

$$\frac{\gamma_i \models \kappa'\tau_1 \ \&\ \sigma \models \beta'\tau_1\tau_2 \ \&\ \neg\exists\pi^* \in P : (\pi^*,(\kappa'\tau_1 \leftarrow \beta \mid \pi),\_) \in \Pi}{\langle\iota,\sigma,\gamma,\Pi,\theta,\xi\rangle \longrightarrow \langle\iota,\sigma,\gamma,\Pi\cup\{(\pi'\tau_1\tau_2\ ,\ \kappa'\tau_1 \leftarrow \beta \mid \pi\ ,\ id)\},\theta,\xi\rangle}$$

where *id* is a fresh plan identifier. Note that it is checked that there is not already a plan in Π which is generated by the same planning rule for the same goal. Note also that κ can be `true`. In such a case, the applied PG-rule can be re-applied if the plan generated by it is completely executed and removed from the plan base.

Goals can be adopted and dropped from the goal base by means of specific adopt and drop goal actions. There are two actions to add goal `g` to the goal base: `adopta(g)` and `adoptz(g)`.

$$\frac{\sigma \not\models \texttt{g}\theta \ \&\ \texttt{ground}(\texttt{g}\theta)}{\langle\iota,\sigma,\gamma,\{(\texttt{adoptX}(\texttt{g}),r,id)\},\theta,\xi\rangle \longrightarrow \langle\iota,\sigma,\gamma',\{\},\theta,\xi\rangle}$$

where $\texttt{ground}(\texttt{g}\theta)$ means that $\texttt{g}\theta$ is a ground formula, $\gamma' = \texttt{g}\theta\cdot\gamma$ if `adoptX` is `adopta` (i.e., the goal $\texttt{g}\theta$ is added to the begin of the list γ of goals) and $\gamma' = \gamma\cdot\texttt{g}\theta$ if `adoptX` is `adoptz` (i.e., the goal $g\theta$ is added to the end of γ).

The `dropGoal(g)` action drops all goals that are logical subgoals of `g` from the goal base.

$$\frac{\gamma' = \gamma \setminus \{f \mid \texttt{g}\theta \models f\}}{\langle\iota,\sigma,\gamma,\{(\texttt{dropGoal}(\texttt{g}),r,id)\},\theta,\xi\rangle \longrightarrow \langle\iota,\sigma,\gamma',\{\},\theta,\xi\rangle}$$

The transitions for `dropSubGoal(g)` and `dropExactGoal(g)` are similar except that $\gamma' = \gamma \setminus \{f \mid f \models \texttt{g}\theta\}$ and $\gamma' = \gamma \setminus \{f \mid f \equiv \texttt{g}\theta\}$, respectively. See section 2.2 for their intuitive meanings.

The execution of an atomic plan is the non-interleaved execution of the maximum number of actions of the plan. Let $[\alpha_1;\ldots;\alpha_n]$ be an atomic plan. We need to define a transition rule that allows the derivation of a transition from a configuration $A_1 = \langle\iota,\sigma_1,\gamma_1,\{([\alpha_1;\ldots;\alpha_n],r,id)\},\theta_1,\xi_1\rangle$ to a configuration$A_m = \langle\iota,\sigma_m,\gamma_m,\Pi,\theta_m,\xi_m\rangle$ such that either $\Pi = \{([\alpha_k;\ldots;\alpha_n],r,id)\}$ and α_k is the first action whose execution fails, or all actions of the atomic plan are successfully executed, i.e., $\Pi = \{\}$. Let $A_i = \langle\iota,\sigma_i,\gamma_i,\{[(\pi,r,id)]\},\theta_i,\xi_i\rangle$ and $A_{i+1} = \langle\iota,\sigma_{i+1},\gamma_{i+1},\{[(\pi',r,id)]\},\theta_{i+1},\xi_{i+1}\rangle$. In order to specify the transition rule for atomic plans, we define $transition(A_i,A_{i+1})$ to indicate that the following one-step transition is derivable (the execution of one step of plan π results plan π')[1]:

$A_i = \langle \iota, \sigma_i, \gamma_i, \{(\alpha;\pi, r, id)\}, \theta_i, \xi_i\rangle \longrightarrow \langle \iota, \sigma_{i+1}, \gamma_{i+1}, \{(\pi', r, id)\}, \theta_{i+1}, \xi_{i+1}\rangle = A_{i+1}$
The following transition rule specifies the execution of atomic plan $[\alpha_1;\ldots;\alpha_n]$.

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

where $A_1 = \langle \iota, \sigma_1, \gamma_1, \{([\alpha_1;\ldots;\alpha_n], r, id)\}, \theta_1, \xi_1\rangle$ and $A_{m+1} = \langle \iota, \sigma_{m+1}, \gamma_{m+1}, \Pi, \theta_{m+1}, \xi_{m+1}\rangle$. Note that the condition $\forall A : \neg transition(A_{m+1}, A)$ can hold for two reasons: either there is no action to execute or the execution of one of the involved action has failed. In the first case the resulting plan base Π contains an empty plan $([\epsilon], r, id)$ and in the second case a non-empty plan $([\pi_{m+1}], r, id)$.

Finally, the execution of the application of a plan repair rule is based on the received exceptions that identify failed plans. Let $\xi = \langle E, I, M\rangle$ be the event base of a 2APL agent and $\pi_1 \leftarrow \beta \mid \pi_2$ be a variant of a PR-rule. Suppose the execution of a plan $(\pi, r, id) \in \Pi$ fails such that $id \in I$. Then, the plan repair rule can be applied if the failed plan π matches the abstract plan expression π_1 in the head of the rule, and moreover, its belief condition is derivable from the belief base. The result is a substitution that will be applied to the abstract plan expression in the body of the rule to generate a new plan and to add it to the plan base. We assume a unification operator $U(\pi, \pi_1)$ that implements a prefix matching strategy for matching plan π with abstract plan expression π_1. Roughly speaking, a prefix matching strategy means that the abstract plan expression is matched with the prefix of the failed plan. The unification operator returns a tuple (τ_T, τ_P, π^*) where τ_T is a term substitution, τ_P is a plan substitution and π^* is the postfix of π that did not play a role in the match with π_1 (e.g., $U(\alpha(a);\alpha(b);\alpha(c)\ ,\ X;\alpha(Y)) = ([Y/b], [X/\alpha(a)], \alpha(c))$). Note the all substitutions are applied to the abstract plan expression from the body of the rule to generate a new plan.

$$\frac{U(\pi, \pi_1) = (\tau_T, \tau_P, \pi^*)\ \&\ \sigma \models \beta\tau_T\tau_2\ \&\ id \in I}{\langle \iota, \sigma, \gamma, \{(\pi, r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\pi_2\tau_T\tau_2\tau_P;\pi^*, r, id)\}, \theta, \langle E, I \setminus \{id\}, M\rangle\rangle}$$

If no plan repair rule can be applied to the failed plan, then the exception is deleted from the event base and the failed plan remains in the plan base.

$$\frac{\forall(\pi_1 \leftarrow \beta \mid \pi_2) \in PR : (U(\pi, \pi_1) = \bot \text{ or } \sigma \not\models \beta)\ \&\ id \in I\ \&\ (\pi, r, id) \in \Pi}{\langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I \setminus \{id\}, M\rangle\rangle}$$

4 Conclusion and Future Works

In this paper, we presented a BDI-based agent-oriented programming language that provides practical constructs for the implementation of cognitive agents. The complete syntax and the intuitive interpretation of the involved programming constructs are discussed. Unfortunately, because of the space limitation we could only present the transition semantics of some characterising programming constructs.

[1] Note that the execution of an abstract action in a plan can extend the plan.

We have implemented this semantics in the form of an interpreter that can execute 2APL programs (i.e., initial configuration of 2APL agents). The execution of agents is based on a deliberation cycle. Each cycle determines which transition in which order should take place. The 2APL interpreter starts with applying planning goal rules to generate plans for the agent's goals, selects and executes plans, checks for exceptions and repairs failed plans by applying plan repair rules, and finally checks for received messages and events to apply the procedural call rules. This interpreter is integrated in a 2APL platform that allows an agent programmer to load, edit, run, and debug a set of 2APL agents. This platform is built on top of the JADE platform in order to exploit all tools and facilities that are developed for the JADE platform. These include tools such as the Sniffer, Introspector, and RMA (Remote Agent management). We use also the JADE communication layer to implement the communication between agents. Note that the JADE platform aims to be complaint with the FIPA standards. Since the communication between 2APL agents are through the JADE platform, the 2APL interpreter inherits the objective of the JADE platform of being FIPA complaint.

We are working on various extensions of both 2APL language (e.g., adding constructs to implement organisations and coordination artifacts at the multi-agent level) as well as tools to be integrated in the 2APL platform (e.g., visual programming and debugging facilities). The current implementation of the 2APL platform together with some examples and documentation can be downloaded from the following 2APL web site.

`http : //www.cs.uu.nl/2apl/`

References

1. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE - a java agent development framework. In: Multi-Agent Programming: Languages, Platforms and Applications, Kluwer, Dordrecht (2005)
2. Bordini, R., Hübner, J.F., Vieira, R.: Jason and the golden fleece of agent-oriented programming. In: Multi-Agent Programming: Languages, Platforms and Applications, Kluwer, Dordrecht (2005)
3. Dastani, M., van Riemsdijk, M., Meyer, J.-J.C.: Programming multi-agent systems in 3apl. In: Multi-Agent Programming: Languages, Platforms and Applications, Kluwer, Dordrecht (2005)
4. Dastani, M., van Riemsdijk, M.B., Meyer, J.-J.C.: Goal types in agent programming. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006) (2006)
5. Hindriks, K.V., Boer, F.S.D., Hoek, W.V.D., Meyer, J.-J.C.: Agent programming in 3apl. In: Autonomous Agents and Multi-Agent Systems, vol. 2(4), pp. 357–401 (1999)
6. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J.C.: Agent Programming with Declarative Goals. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS (LNAI), vol. 1986, pp. 228–243. Springer, Heidelberg (2001)
7. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Multi-Agent Programming: Languages, Platforms and Applications, Kluwer, Dordrecht (2005)
8. Winikoff, M.: JACK^{TM} intelligent agents: An industrial strength platform. In: Multi-Agent Programming: Languages, Platforms and Applications, Kluwer, Dordrecht (2005)