Practical Extensions in Agent Programming Languages

Mehdi Dastani
Utrecht University
The Netherlands
mehdi@cs.uu.nl

Dirk Hobo
Utrecht University
The Netherlands
dhobo@cs.uu.nl

John-Jules Ch. Meyer
Utrecht University
The Netherlands
jj@cs.uu.nl

ABSTRACT

This paper proposes programming constructs to improve the practical application of existing BDI-based agent-oriented programming languages that have formal semantics. The proposed programming constructs include operations such as testing, adopting and dropping declarative goals, different execution modes for plans, repairing plans when their execution fail, event and exception handling mechanisms, and interfaces to existing imperative and declarative programming languages.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types, Control structures*

Keywords

Agent Programming Language, Cognitive Agents

1. INTRODUCTION

Existing BDI-based agent programming languages with formal semantics (e.g., [1, 2]) support the implementation of mental concepts such as beliefs, goals, events, and plans. Plans consist usually of basic actions composed by operations such as conditional choice, iteration, and sequence operators. The set of basic actions of these BDI-based agent programming languages includes belief update actions, belief test actions, communication actions, and interfaces to existing imperative languages. These programming languages allow the implementation of pre-compiled planning rules to indicate that certain goals (internal events) can be achieved (handled) by certain plans. Agents may select and apply planning rules to generate plans to achieve (handle) their goals (internal events).

This paper aims at improving the practical application of BDI-based agent-oriented programming languages that enjoy formal semantics. Our experience with these agent programming languages in various academic courses and research projects have shown that some of the existing programming constructs and mechanisms are indeed useful, expressive and effective in programming software agents. It also made it clear that new programming constructs and mechanisms are needed to make these type of agent programming languages even more expressive and *practical*. Moreover, in our view the practical applicability of these programming languages will be improved by providing dedicated programming constructs to implement distinct agent concepts. For example, it is not practical to implement an agent's goals by means of events, as it is the case in some BDI-based programming languages such as Jason [1].

In this paper, we propose special purpose programming constructs to improve the practical application of BDI-based agent-oriented programming languages that have formal semantics. In particular, we suggest to extend such BDI-based agent programming languages with explicit programming constructs for declarative goals, their dynamics and different types of goal test actions, events that notify agents about external environmental changes, exception handling mechanisms that notify the execution failure of plans, a recovery mechanism to repair failed plans, different execution modes for plans, and dedicated programming mechanisms for procedure calls and recursion. Our proposal is illustrated by presenting the relevant programming constructs of a BDI-based agent programming language, called 2APL (A Practical Agent Programming Language). Due to space limitation we do not present the full-fledged formal syntax and semantics of this programming language. More information about 2APL and its implemented interpreter and development platform can be found at:

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

It should be noted that some of the existing and more practical BDI-based agent programming languages such as Jadex [3] and Jack [4] provide programming constructs that may look similar to some of those proposed in this paper. However, since these agent programming languages lack a formal semantics, the exact meaning of the provided programming constructs (e.g., beliefs, goals, their dynamics, events, and event handling mechanism) is not clear. This makes it difficult to study and compare their provided programming constructs and to use (or implement) them in other BDI-based agent programming languages. The aim of these paper is to propose practical programming constructs with exact and formal semantics.

2. DECLARATIVE GOALS

An important characteristic of BDI-based agents is their proactive behavior which assumes that agents have a declarative representation of their objectives (goals) and should be able to reason about them. Moreover, agents with goals and beliefs should be able to establish a (realistic) relation between their beliefs and goals, for example, to avoid situations in which they aim at achieving states that are believed to be achieved. In 2APL an agent can have various goals each of which is a conjunction of grounded atoms. The following is an example of an agent's goals in 2APL which represent

AAMAS'07 2007 Honolulu, Hawaii USA

the agent's desire to achieve two different states: a state in which it possesses object a (at position (2,3)) and has object b on the top of object c, and a state in which room r1 is clean and object a is on the top of object b.
"possess(a,2,3) and on(b,c) , clean(r1) and on(a,b)"
Note that these two states need not be achieved simultaneously allowing the agents to have goals that are mutually inconsistent.

A BDI-based agent programming language should provide programming constructs that determines whether and which goals are pursued by the agent. In particular, a *goal test action* is a programming construct that can be used in an agent's plan to test whether a formula is derivable from the agent's goals, i.e., whether the agent has a certain goal. This action can be used in a plan to instantiate the variables of a particular pursued goal to pass it to the rest of the plan, or to block the execution of the rest of a plan if the agent does not pursue the tested goal.

For example, an agent may have a plan in which it tests if it wants to possess an object, and if so, which one, by means of a goal test action. In 2APL this action is implemented as possess(Obj,X,Y)! (the exclamation mark is used to distinguish goal from belief test action, which is implemented in 2APL by the question mark '?'). This action will be successfully executed if the agent has, for example, the goal possess(a,2,3) resulting in an instantiation of variables Obj, X and Y. Note that the goal test action can also be executed successfully if the agent has the complex goal possess(a,2,3) and on(b,c). In such a case, the agent should have the ability to reason if the tested goal possess(Obj,X,Y) is derivable from the agent's goal possess(a,2,3) and on(b,c). In addition, 2APL provides a variant of the goal test action, called *exact goal test action*. This action succeeds only if the agent has the tested goal as one of its goals. The exact goal test action is implemented by double exclamation mark '!!'. In the above example, (clean(r1) and on(a,b))!! will succeed while clean(r1)!! fails.

A pro-active agent should be able to generate plans to achieve its goals. A planning goal rule is the dedicated programming construct for this purpose. A planning goal rule in 2APL consists of three entries. 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 is a plan that may share variables with the first two entries. These shared variables will be bounded when the planning goal rule is applied, i.e., when it is tested if the goal and belief expressions of the rule are derivable from the agent's goals and beliefs, respectively. The following is an example of a planning goal rule indicating that a plan to go to room R1, departing from the current location (believed to be room R2, pos(R2)), to remove trash can be generated if the agent has goal clean(R1).
"clean(R1) <- pos(R2) | {goTo(R2,R1);RemoveTrash()}"
Note that this rule can be applied if (beside the satisfaction of the belief condition) the agent has goal clean(r1) and on(a,b) since the head of the rule is derivable from this goal.

Finally, an agent may want to modify its goals during its execution. For example, an agent may want to adopt a goal because it receives an order from its boss to clean a room, or it may want to drop a goal because it discovers that the goal is not achievable anymore. The programming constructs *adopt goal* and *drop goal* actions are used in 2APL to adopt and drop a goal to and from the agent's goals, respectively. The adopt goal action 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 goals, respectively. Note that the programmer has to ensure that the variables in ϕ are instantiated before these actions are executed since the goals should always be grounded. Finally, the drop goal action can have three different forms: dropGoal(ϕ), dropSubGoal(ϕ), and dropExactGoal(ϕ). These actions can be used to drop from an agent's goals, respectively, all goals that are a subgoal of ϕ, all goals that have ϕ as a subgoal, and exactly the goal ϕ.

3. EVENTS

As agents may operate in dynamic environments, they have to observe their environmental changes either actively by *explicit sensing operations* or passively by being notified through *(external) events*. It should be noted that in some agent programming languages events are used for various purposes. For example, in Jason [1] events are used to model an agent's goals. 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 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.

In 2APL information can be passed from an external environment to an agent through *events*. When implementing an environment in Java, the programmer should decide when and which information from the environment should be passed to one or more agents. This is done in an environment implemented in Java by the method notifyEvent(AF event, String... agents) in the *ExternalEventListener* which is an argument of the environment's constructor. The first argument of this method may be any valid atomic formula. The rest of the arguments can be filled with agent names. The event is sent to all named agents. If no agent name is given as argument, the event is sent to all agents. Such a mechanism of generating events by the environment and catching it by agents can be used to implement the agents' passive perceptual mechanism.

Events can be processed by the agents through event handling rules. The following is an example of such an event handling rule.
"event(dirty(R)) <- not dirty(R) | {UpdateRooms(R)}"
This rule generates a plan to update the agent's beliefs with the fact that the room R is dirty when it receives event dirty(R) and believes the room is not already dirty. It should be noted that sending messages also generates events that are subsequently processed by the agents to trigger their event handling rules. In practice, the generation of events for the communication actions will be performed by the 2APL interpreter and not by 2APL programmers.

4. PLANS

An agent may have a set of plans. In most agent-oriented programming languages, the executions of an agent's plans 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 at once without being interleaved with the actions of other plans. In general, a plan can be executed in various modes, for example, in *atomic mode* by executing it at once or in *interleaving mode* by allowing its execution to be interleaved with the execution of other plans. In 2APL arbitrary plan parts are allowed to be executed in either of these modes. This is realised by introducing a unary plan operator implemented by []. The application of this operator to a plan part indicates that

the plan part should be executed atomically. The following example illustrates the implementation of a 2APL plan in which the last three actions should be executed atomically. This means that the execution of this plan can only be interrupted after the execution of the first action; as soon as the agent picks up the trash, it should do nothing else but to go to room R3 to drop the trash.

```
"goTo(R2,R1);[PickUpTrash();goTo(R3);DropTrash()]"
```

The execution of plans can fail for various reasons. 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. For example, if the first action of a plan is an unspecified belief update action or an external action (to be performed in an environment) whose execution does not succeed within a given time limit, then the execution of the whole plan is considered as failed. The failed plans can be repaired by means of a plan repair mechanism. In 2APL, a programmer can specify how to repair 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 may be applied to all plans continuously. In our view, it does not make sense to modify a plan if the plan is correct and executable. 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 instantiated with a plan. A plan repair rule indicates that if the execution of an agent's plan (i.e., a plan that can be instantiated with the abstract plan expression in the head of the rule) fails and the agent has a certain belief, then the failed plan should be replaced by another plan.

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 matched with the abstract plan expression in the head of the rule, and 3) the belief query expression is derivable from the agent's beliefs. 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 instantiated 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 matched 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 a new (repaired) plan. Details on the exact matching algorithm can be found on the web page of 2APL. The following is an example of a plan repair rule. This rule indicates that if the action to go from room R1 to room R2 of a plan fails and the agent believes it is not in the departing room R1 but in room R3, then the plan can be repaired by replacing the failed action `goTo(R1,R2)` with the action `goTo(R3,R2)`.

```
"goTo(R1,R2);X <- not pos(R1) and pos(R3) | {goTo(R3,R2);X}"
```

Note the use of the plan variable `X` that indicates that any failed plan starting with `goTo(R1,R2)` can be repaired by the same plan in which the first external action is replaced by `goTo(R3,R2)`.

5. PROCEDURE CALL AND RECURSION

Encapsulation of a plan by a single action allows the reuse of the plan and recursion. In 2APL, *abstract actions* are introduced as abstraction mechanism allowing the encapsulation of a plan by a single action. An abstract action will be instantiated with a concrete plan when the action is executed. The instantiation of plans for abstract actions are specified through specific procedure rules. In fact, the general idea of an abstract action is similar to a procedure call in imperative programming languages while the procedural rules function as procedure definitions. A procedure rule consists of three entries. The head of the rule is an atom (predicate-argument expression) that can be matched with an abstract action. The belief condition of the rule indicates when the rule can be applied. The body of the rule is a plan that should replace the the abstract action that matches the head of the rule. Thus, an agent's procedure rule can be applied if the agent executes an abstract action that matches the head of the rule and the belief condition of the rule is derivable from its beliefs. The following is an example of a procedure rule which indicates how to go from room R1 to R2 when there is no direct door between these two rooms.

```
"goTo(R1,R2) <- pos(R1) and not door(R1,R2) |
                {door(R1,R3)?;move(R1,R3);goTo(R3,R2)}"
```

6. CONCLUSION AND FUTURE WORKS

In this short paper, we presented a set of programming constructs that improve the practical application of BDI-based agent-oriented programming languages. Due to the space limitation, we have explained only the intuitive semantics of the programming constructs and have omitted the presentation of their formal semantics. The presented programming constructs form a subset of programming constructs of a BDI-based agent programming language called 2APL. The complete syntax and semantics as well as the implemented interpreter and development environment of 2APL is available at the 2APL web page. It should be noted that there are many features of 2APL programming language which we could not explain in this short paper. For example, the basic actions of 2APL can have optional time-out arguments that are used to define the execution failure of those actions. Also, the communication action can have optional arguments to indicate the used language and ontology of the content of the communicated messages.

The interpreter of 2APL is integrated in a multi-agent system development environment 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 compliant 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 compliant. 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).

7. REFERENCES

[1] R. Bordini, J.F.Hübner, and R. Vieira. Jason and the golden fleece of agent-oriented programming. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.

[2] G. d. Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[3] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.

[4] M. Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.