Chapter II
Modular Rule-Based Programming in 2APL

Mehdi Dastani
Intelligent Systems Group, Utrecht University, The Netherlands

ABSTRACT

This chapter presents a modular version of a rule-based programming language called 2APL (A Practical Agent Programming Language). This programming language is designed to support the implementation of multi-agent systems where individual agents are specified in terms of cognitive concepts such as beliefs, goals, events, actions, plans, and three types of reasoning rules. The reasoning rules facilitate an effective integration of these concepts and allow generation, repair, and execution of plans based on beliefs, goals, and events. The modules can be used to implement different agent concepts such as roles and agent profiles, or to adopt common programming techniques such as encapsulation and information hiding. The syntax and the informal semantics of the programming constructs of modular 2APL are presented and discussed. A simple example is provided to illustrate how various ingredients of the presented programming language can be used.

INTRODUCTION

Agent-oriented software engineering paradigm is a modern approach for the development of distributed intelligent systems. In this paradigm, software systems (called multi-agent systems) consist of a number of interacting (software) agents, each of which is capable of sensing its environment (including other agents) and deciding to act in order to achieve its design objectives. Examples of such systems are e-commerce applications, auctions, electronic institutions, power management systems or transportation systems.

Multi-agent systems are specified, designed, and implemented in terms of high-level concepts and abstractions such as roles, communication,
beliefs, goals, plans, actions, and events. Different development methodologies (Bergenti et al., 2004), specification languages (e.g., BDI\(_{CTL}\) (Rao et al., 1991, Cohen et al., 1990) and KARO (Meyer et al., 1999), and programming languages (Bordini et al., 2007, Winikoff et al., 2005, Pokahr et al., 2005, Hindriks et al. 1999, Kakas et al. 2004, Giacomo et al. 2000) have been proposed to facilitate the development of agent-based systems.

While most agent-oriented development methodologies specify and design system architectures in terms of agent concepts and abstractions, the proposed agent-oriented programming languages and development tools aim at providing programming constructs to facilitate direct and effective implementation of these concepts and abstractions. Moreover, existing agent-oriented programming languages aim at supporting programming techniques such as modularity, reuse, encapsulation and information hiding. The availability and combination of agent-oriented programming constructs and programming techniques characterize and differentiate these programming languages and determine their usefulness and applicability.

Existing agent-oriented programming languages differ as they provide programming constructs for specific, sometimes overlapping, sets of agent concepts and abstractions. They also differ as they are based on different logics and use different technologies. Some of them are rule-based capturing the interaction of agent concepts by means of specific rules, while others are extension of Java programming language. Some capture specific rationality principles that underlie agent concepts in their semantics, while such principles are assumed to be implemented by agent programmers in other programming languages. Finally, they differ in programming techniques that are introduced to support the implementation of multi-agent systems. See (Bordini et al., 2005) for a comparison between some of these agent programming languages.

In this chapter, a modular rule-based agent-oriented programming language is presented that 1) separates multi-agent from single-agent concerns, 2) provides and integrates programming constructs that are expressive enough to implement a variety of agent concepts and abstractions used in the existing agent-oriented methodologies, 3) provides different types of rules to capture the interaction of agent concepts such as beliefs, goals and plans, 4) introduces a specific notion of modules and provides a set of module related operations that allows an agent programmer to determine how and when modules are used, and 5) realizes an effective integration of declarative and imperative programming styles. It is important to emphasize that multi-agent systems can be implemented in any existing programming language. However, we aim at designing an agent-oriented programming language that provides dedicated and expressive programming constructs and techniques to facilitate practical and effective implementation of agent related concepts and abstractions.

The structure of this chapter is as follows. In the next section, we provide a brief discussion on the exiting BDI-based agent-oriented programming languages (BDI stands for Beliefs, Desires, and Intentions). These programming languages are motivated by the BDI logics (Rao, 1996, Rao et al., 1991, Cohen et al., 1990) that are designed to specify agent behavior. The BDI-based programming languages provide dedicated programming constructs to implement individual agents in terms of (cognitive) concepts such as beliefs, desires and intentions. As we will see later in this chapter, these (cognitive) concepts can be used to implement individual agents that can decide to act in order to achieve their objectives. Then, a general description of a BDI-based agent-oriented programming language called 2APL (A Practical Agent Programming Language) is presented and some of its characterizing features are discussed. Subsequently, the complete syntax of 2APL is given and the intuitive meaning of its ingredients.
Modular Rule-Based Programming in 2APL

is explained by means of some examples. It is then explained how the programming language is interpreted and executed. An example of a complete 2APL multi-agent program is provided and its execution behavior is explained. Finally, the chapter is concluded and some future directions to extend the programming language are discussed.

BACKGROUND

Existing BDI-based agent-oriented programming languages such as Jason (Bordini et al., 2007), Jack (Winikoff, 2005), Jadex (Pokahr et al., 2005), 3APL (Hindriks et al., 1999), KGP (Kakas et al., 2004), Minerva (Leite et al., 2001), GOAL (Hindriks et al., 2001), (Concurrent) MetateM (Fisher et al., 1994) and the family of Golog languages (Giacomo et al., 2000, Sardina et al., 2004) are designed to implement individual agents in terms of BDI (Belief, Desire, Intention) concepts. These programming languages provide programming constructs to implement individual agents in terms of beliefs, goals, events, and plans. Based on this model of agency, the actions and plans of an implemented agent are generated and executed in order to achieve its goals or to react to the events and messages that it receives from the environment or other agents, respectively. In this section, we give a brief overview of some of these programming languages that have influenced the development of 2APL because they are rule-based or because they provide a notion of modularity in terms of BDI concepts. Some similarity and differences between these languages and 2APL will be enumerated. A detailed comparison between some of these programming languages can be found in (Bordini et al., 2005). It is important to note that these languages are not meant to be used to implement large-scale industrial applications. Their main purpose is to show the type of programming constructs and their semantics that can facilitate the implementation of multi-agent systems. In our view, a full-fledge industrial strength multi-agent programming language is expected to be developed based on these proposals. It is therefore not sensible to compare these programming languages with respect to issues such as performance and execution time.

2APL extends and modifies the original version of 3APL (An Abstract Agent Programming Language) (Hindriks et al., 1999) in many different ways. In this version of 3APL, an agent consists of beliefs and plans, where plans can consist of belief update, test, and abstract actions composed by the sequence and choice operator. The original version of 3APL provides only rules that can be applied to revise an agent’s plan. The execution of a 3APL agent program is a continuous execution and repair of the agent’s plans. 2APL includes a modification of these programming constructs and adds new constructs to implement, among other things, events, goals, and a variety of action types such as external actions, goal related actions, and communication action. Furthermore, 2APL provides two additional rule types to implement an agent’s reactive and proactive behavior. While 3APL rules can be applied to revise any arbitrary plan at run-time, the corresponding modified 2APL rules are applied only to revise plans when their executions are failed. In our view, it does not make sense to modify a plan if the plan is executable. The addition of declarative goals and events are the reasons for the introduction of the two new types of rules. In particular, the first type of rules, which can be used to implement proactive behavior, can be applied to achieve an agent’s (declarative) goals and the second type of rules, which can be used to implement reactive behavior, can be applied to react to the received events (including messages from other agents). The original version of 3APL is not modular and does not provide any mechanism to encapsulate cognitive concepts such as beliefs, goals, events, and plans. However, in (van Riemsdijk et al., 2006) a notion of goal-oriented modularity is introduced that can be applied to agent-oriented
programming languages such as 3APL. In this proposal, a module is associated with a specific goal (or a set of related goals) indicating which and how rules should be applied to achieve that goal (or the set of related goals).

It is worth mentioning that the GOAL programming language is recently extended with a notion of modularity as well (Hindriks, 2008). A GOAL agent can be implemented in terms of beliefs, goals, and a rule-based programming construct to implement the selection of actions based on goals and beliefs. In this extension of GOAL, a module encapsulates beliefs, goals, and action selection rules and can be activated when certain beliefs and goals hold. The activation of modules, which is considered as the ‘focus of execution’, is used to disambiguate the application and execution of actions. In contrast to this module-based approach, 2APL provides a variety of programming constructs that enable an agent programmer to control and determine the activation and use of modules. These programming constructs can be used to implement, among other things, the ‘focus of execution’ and goal processing as discussed in (Hindriks, 2008, van Riemsdijk et al., 2006).

Jason is introduced as an interpreter of an extension of AgentSpeak (Rao, 1996), which is originally proposed by Rao (Rao, 1996). An individual agent in Jason is characterized by its beliefs, plans and the events that are either received from the environment or generated internally. Like 2APL, Jason is a rule-based agent-oriented programming language. A rule in Jason is called a plan and consists of a triggering event, a belief condition, and a sequence of actions. The execution of individual agents in Jason is controlled by means of a fixed (not programmable) cycle of operations encoded in its operational semantics. In each cycle, a rule is selected and applied when its triggering event is received and its corresponding belief condition holds. The application of the selected rule will generate an intention (instantiated plan), which is subsequently added to the intention base. An intention is then selected from the intention base and executed (through which new internal events can be generated). A plan in Jason is similar to one specific type of rules in 2APL, called procedure call rules. Such a rule is applied by a 2APL agent to generate a plan when it receives a relevant event and has a specific belief. 2APL has two additional rule types that generate plans to achieve declarative goals and repair failed plans. The introduction of the goal related rules is due to the fact that Jason has no explicit programming construct to implement declarative goals, though goals can in Jason be indirectly simulated by means of a pattern of plans. It should also be noted that plan failure can be modeled in Jason by means of the so-called deletion events. Jason is not a modular programming language as it does not provide programming constructs to implement modules and operate on them.

In contrast to Jason and 3APL, Jack and Jadex provide a specific and advanced notion of modularity. Jack and Jadex extend Java with programming constructs to implement BDI concepts such as beliefs, goals, plans, and events. In both Jack and Jadex syntactic constructs are added to Java to allow programmers to declare beliefsets, to post events, and to select and execute plans. Jadex uses XML notation to define and declare the BDI ingredients of an agent. Some of these ingredients such as beliefs and plans are implemented in Java. The execution of agent programs in both languages are motivated by the classical sense-reason-act cycle, i.e., processing events, selecting relevant and applicable plans, and execute them. Jack and Jadex support modularity by introducing the notion of capability (Busetta et al., 2000, Braubach et al., 2005). A capability in JACK encapsulates beliefs, events, and plans. A capability can import another capability. Using the import and export mechanism for capabilities, the beliefs of a capability can be used by its super-capability and sub-capability, respectively. A similar mechanism exists for events through which events that are posted from one capability can be handled by plans of its sub- and super-
Modular Rule-Based Programming in 2APL

capabilities. The Jadex capability encapsulates beliefs, goals, plans, and events. It differs from the capability in JACK by the fact that a general import/export mechanism is introduced for all kinds of elements of a capability. In both approaches, the encapsulated components are used during an agent’s execution when an event should be created or when a plan should be generated to handle an event.

2APL: A PRACTICAL AGENT PROGRAMMING LANGUAGE

One of the features of 2APL is the separation between multi-agent and individual agent concerns. Therefore, 2APL provides two distinguished sets of programming constructs to implement multi-agent and individual agent programs.

For multi-agent programs, it provides programming constructs to 1) declare modules and environments, and 2) create agents that constitute multi-agent systems. In fact, individual agents are created as instances of the declared modules. The agents will interact with each other and perform actions in the environments when they are executed. It is important to understand the distinction between a module and an agent. In our approach a module specifies a state (e.g., the initial state of an agent), while an agent is a deliberative process that continuously perceives its environments, update its state (initially specified by a module), and reason about its state to decide which action to perform. The initial state of an agent is thus based on a module instance. Besides using modules to create individual agents, a module can also be used at run-time by individual agents for various reasons such as playing a role, processing an agent profile, or for encapsulation purpose (see below for more details).

As agents (and modules) can perform actions in the environments, it may be desirable to control the access relation between them and the environments. For this reason, in a multi-agent program this access relation is specified by associating environments to modules (and thus to agents). In 2APL, environments are assumed to be implemented as Java objects. Each environment (Java object) has a state and allows the execution of a set of actions to change its state. The actions that can be executed in an environment are modeled as methods of the Java object that implements the environment. The performance of an action by an agent in an environment is then a method call of the environment object. It should be noted that these Java objects can also function as interfaces to the physical environments or other software. Moreover, the separation between multi-agent and individual agent concerns can be used to design and add a set of programming constructs at the multi-agent level to implement a wide range of social and organizational concepts in a principled and modular manner. These programming constructs can be used to coordinate and regulate the behavior of individual agents (see Dastani et al., 2008, Tinnemeier et al., 2008).

For individual agent programs (modules), 2APL provides programming constructs to implement beliefs, goals, actions, plans, events, and three different types of rules that can be applied to generate plans. In fact, a module can be considered as the specification of a (cognitive) state (e.g., the initial state of an individual agent). The beliefs and goals are implemented in a declarative way, while plans and (interfaces to) environments are implemented in an imperative programming style. The declarative programming part supports the implementation of reasoning and update mechanisms that are needed to allow individual agents to reason about and update their (cognitive) states. The imperative programming part facilitates the implementation of plans, flow of control, and mechanisms such as procedure call, recursion, and interfacing with existing imperative programming languages.

An instance of a module can be created, processed, and removed at run-time. As noticed, the modules are declared in the multi-agent program.
There are several operations that can be performed on a declared module. One of these operations is to create a module instance. One module can be created more than once resulting in different instances of a module. A unique name is assigned to each created module instance. In 2APL, an agent can create several instances of one and the same module and two agents can create two instances of one and the same module. Moreover, one and the same module instance can be used by two different agents, though not at the same time. In order to allow a module instance to be used by more than one agent, a special type of module, called singleton module, is introduced. A module instance can also be executed because it may include plans that can be executed, and rules that can be applied to generate plans. Besides executing a module instance, the internals of a module instance can be accessed and updated. For example, an agent can test and update the beliefs and goals of a module instance. In order to control the access to the internals of a module instance, two types of modules are introduced: public and private. An instance of a private module can only be executed and does not allow other module instances to access its internals, while the internals of a public module instance can also be accessed.

In summary, a multi-agent system can be implemented in 2APL by means of one multi-agent program (declaration of modules and environments, and creation of agents), a set of module programs (each specifying a state in terms of concepts such as beliefs, goals, plans, and rules), and a set of environments (Java classes). The execution of a multi-agent program starts with the creation of individual agents (modules instances) and environments (Java objects), followed by the parallel execution of individual agents. Note that executing individual agents may result in performing actions in the environments (i.e., method calls of Java objects). Figure 1 illustrates the general architecture of such a multi-agent system. The individual agents $A_1,..., A_n$ can interact with each other or perform external actions in the environments $Env-1,..., Env-k$.

Figure 2 shows the general architecture of individual agents. As illustrated, an individual agent has a state composed from its beliefs, goals, plans, events, action specifications, rules, and a deliberation cycle. The deliberation cycle, which models the interpreter/execution of individual agent programs, specifies a process through which the agent senses its environments, updates its states accordingly, and decides to act in order to achieve its goals or to react to its received events. This deliberation cycle is explained in details later in this chapter.

**Multi-Agent Programming**

In 2APL, multi-agent concepts are implemented by means of a language that allows the declaration of modules, the initialization and creation of individual agents in terms of modules, and the access relation between modules and environments. Note that the access relation between modules and environments is inherited by the agents such that the access relation determines which agent can access which environments. The syntax of this specification language is presented in Figure 3 using the EBNF notation. In the following, we use $<ident>$ to denote a string and $<int>$ to denote an integer.

A 2APL multi-agent program starts with the declaration of a non-empty list of modules (i.e., $<module>$). A module is declared by means of a file name, with .2apl exten-
Modular Rule-Based Programming in 2APL

Figure 2. The general architecture of 2APL individual agents

The following is a simple example of a 2APL multi-agent program. This program implements a multi-agent system in which one single manager and two workers cooperate to collect some gold items in the blockworld environment. The manager coordinates the activities of the two workers by asking them either to explore the blockworld environment to detect the gold items or to carry the detected gold items and store them. In this example, the manager module (i.e., manager.2apl) specifies the initial state of the manager agent with the name m (the implementation of the manager module is explained later on). The manager module, and thus the manager agent m, can access the database environment. Note that only one manager agent will be initialized and created. Moreover, the worker module (worker.2apl) specifies the initial state of two worker agents. As there will be two worker agents, the name w will be indexed with numbers 1 and 2, i.e., there will be two worker agents with names w1 and w2. Finally, two additional modules are declared to implement the explorer and carrier.
functionalities. As we will see later on, these functionalities will be used by the worker agents. Note that both functionalities have access to the blockworld environment.

Modules: manager.2apl @database
worker.2apl
explorer.2apl @blockworld
carrier.2apl @blockworld

Agents: m : manager
w : worker 2

Individual Agent Programming

A 2APL module is composed of various ingredients that specify different aspects of agency. Such a module can be used to specify the initial state of an agent. The state of some of these ingredients will change at run-time while the state of other ingredients remains the same during the system execution. A module is implemented by means of a specification language. The EBNF syntax of this specification language is illustrated in Figure 4. In this specification, we use <atom> to denote a Prolog-like atomic formula starting with lowercase letter, <Atom> to denote a Prolog-like atomic formula starting with a capital latter, <ground_atom> to denote a ground atom and <Var> to denote a string starting with a capital letter. The implementation of a 2APL module is stored in a file with “.2apl” extension. In the following, we will discuss each ingredient and give examples to illustrate them.

Beliefs and Goals

An agent may have initial beliefs and goals that change during the agent’s execution. In 2APL, the initial beliefs of an agent are implemented by the belief base, which includes information the agent believes about itself and 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. Note that a <belief> 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 following example illustrates the implementation of the initial belief base of a 2APL agent. This belief base represents the information of an agent about its environment. In particular, the agent believes that the names of worker agents are w1 and w2, and that it believes to have found gold items if it believes it has some goal item.

Beliefs:
worker(w1).
Worker(w2).
foundGold() :- gold(_).

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. The following example is the implementation of the initial goal base of a 2APL agent. This goal base includes two goals. The first goal indicates that the agent wants to achieve a situation in which it has found gold items and the blockworld is fully explored. Note that this single conjunctive goal is different than having two separate goals foundGold() and explored(blockworld). In the latter case, the agent wants to achieve two situations independently of each other, i.e., one in which the agent has found gold items, not necessarily explored the blockworld, and one in which it has explored the blockworld and perhaps not found any gold item. The second goal of the agent in the example below indicates that the agent desires a state in which the gold items are stored. Note that different goals in the goal base are separated by a comma.
Goals:
foundGold() and explored(blockworkd),
goldStored()

The beliefs and goals of an agent are governed by a rationality principle. According to this principle, 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 in order to accommodate this rational principle. An agent’s beliefs and goals change during the agent’s execution. Note in the above example of the belief and goal bases that the goal foundGold() is considered as being achieved as soon as the agent believes gold( _ ).

Basic Actions

Basic actions specify capabilities of agents, i.e., actions that an agent can perform to achieve its desirable situation (goals). Basic actions constitute an agent’s plan, as we will see in the next subsection. Six types of basic actions are distinguished in 2APL: actions to update the belief base, actions to test the belief and goal bases, actions to manage the dynamics of goals, abstract actions, communication actions and external actions to be performed in an agent’s environment (including sense actions).

Belief Update Action

A belief update action updates the belief base of an agent when executed. This action type can be used to store information received from other agents (through messages) or environments (through sense actions and events), or to temporarily store data or the results of some computations. 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 entailed by 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 the post-condition of the action is entailed by the belief base after the execution of the action. 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> (see Box 1).

In this example, the specification of the Ready(A) indicates that this belief update action can be performed if agent A (A is a variable that can be instantiated with an agent name) carries a gold item and that after performing this action the agent A will not carry the gold item. Note that the variables in the pre- and post-conditions are bounded by the argument of the action. Note also a possible use of lists in the pre- and post-condition of the Remove(X) action. The specification of belief update actions do not change during agent execution.

Box 1.

<table>
<thead>
<tr>
<th>BeliefUpdates:</th>
<th>Ready(A)</th>
<th>{not carryGold(A)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{carryGold(A)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{not carryGold(A)}</td>
<td>Busy(A)</td>
<td>{carryGold(A)}</td>
</tr>
<tr>
<td>{true}</td>
<td>GoldInf(X)</td>
<td>{gold(X)}</td>
</tr>
<tr>
<td>{gold([X</td>
<td>R])}</td>
<td>Remove(X)</td>
</tr>
</tbody>
</table>
Modular Rule-Based Programming in 2APL

Figure 4. The EBNF syntax of 2APL single agent/module programs

```
<Agent_Prog>  =  {"private" | "public"} ["singleton"]
| "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> | not <atom>} ;

<goals>       =  <goal> "(" <goal> ")" ;

<goal>        =  <ground_atom> ["and" <ground_atom>] ;

<baction>     =  "skip" | <beliefupdate> | <sendaction> | <externalaction>
| ["abstractation"] | ["test"] | ["adoptgoal"] | ["dropgoal"]
| ["createaction"] | ["releaseaction"] | ["return"] | ["moduleaction"]
;

<createaction> =  "create(" <ident> ")" ;

<releaseaction> =  "release(" <ident> ")" ;

<moduleaction> =  <ident> "." <baction> ;

<maction>     =  
| "execute(" <test> ")" | "executeasync(" [test] ")" 
| "stop" | <test> | ["adoptgoal"] | ["dropgoal"]
| ["createaction"] | ["releaseaction"] | ["return"] | ["moduleaction"]
;

<updBB>       =  "updateBB(" ["<literals>"] ")" ;

<plans>       =  <plan> {"," <plan>} ;

<plan>        =  <baction> | ["sequenceplan"] | ["ifplan"] | ["whileplan"]
| ["atomicplan"] | ["mifplan"] | ["mwhileplan"]
;

<beliefupdate> =  <Atom> ;

<sendaction>  =  
| "send(" <iv> ")" | "<atom>")" 
| ["send(" <iv> ")" | "<atom>")" | ["send(" <iv> ")" | "<atom>")"
| ["send(" <iv> ")" | "<atom>")" ;

<externalaction> =  ["true"] | ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
;

<atom>        =  <ident> | <Var> ;

<goalvar>     =  <atom> ;

<planvar>     =  <plan> | <Var> | <planvar> ;

<belquery>    =  ["true"] | ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
;

<literal>     =  <atom> | ["not"] <atom> ;

<literals>    =  ["true"] | ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
;

<ground_atom> =  <ground_atom> | ["not"] <ground_atom> ;

<ground_literal> =  ["true"] | ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
;

<beliefquery> =  ["true"] | ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
;

<goalquery>   =  ["true"] | ["true"] | ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
| ["true"] | ["true"] | ["true"]
;

<iv>          =  <ident> | <Var> ;
```
Test Action

A test action performed by an agent checks if 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 \( B(\varphi) \) where \( \varphi \) consists of literals composed by conjunction or disjunction operators. A goal query expression has the form \( G(\varphi) \), where \( \varphi \) consists of atoms composed by conjunction or disjunction operators. A belief query expression is basically a (Prolog) query to the belief base (which is a Prolog program) and generates a substitution for the variables that are used in the belief query expression. A goal query expression 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 variables that are involved in the goal query expression. 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, if 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(\( \varphi \)) and adoptz(\( \varphi \)). These two actions can be used to add the goal \( \varphi \) (a conjunction of atoms) to the beginning and to the end of an agent's goal base, respectively. Recall that the goal base is a list such that the goals are ordered. Note that the programmer has to ensure that the variables in \( \varphi \) are instantiated before these actions are executed since the goal base should contain only ground formula. Finally, the drop goal action <dropgoal> can have three different forms: dropgoal(\( \varphi \)), dropsubgoals(\( \varphi \)), and dropsupergoals(\( \varphi \)). These actions can be used to drop from an agent’s goal base, respectively, the goal \( \varphi \), all goals that are a logical subgoal of \( \varphi \), and all goals that have \( \varphi \) as a logical subgoal. Note that the action dropsupergoals is also proposed in (Hindriks et al., 2001).

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 so-called PC-rules, which stands for procedure call rules (see subsection ‘Procedure Call Rules’ for a description of PC-rules). As we will see in that subsection, 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 one. 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.

Communication Action

A communication action passes a message to another agent. A communication action <send-action> can have either three or five parameters. In the first case, the communication action is the expression send(Receiver, Performa-
where Receiver is the name of 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 FIPA1 (Foundation for Intelligent Physical Agents) compliant JADE platform2 (Java Agent Development Framework). 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 (Bellifemine et al., 2005) for more information on JADE standards).

**External Action**

An external action is supposed to change the state of an external environment. 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. It is up to the programmer to determine how the effects of actions should be perceived by the agent. An external action `<externalaction>` is an expression of the form @env(ActionName,Return), where env is the name of the agent’s 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. 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. An example of an external action is @blockworld(pickUpGold(),L) (pick up the goal item in the blockworld environment). The effect of this action could be implemented in such a way that the gold item in the blockworld environment is picked up and removed. The list L is expected as the return value. The programmer of the environment determines the content of this list.

**Module Actions**

The first construct related to modules is the use of keywords public, private and singleton. The beliefs and goals of an instance of a public module can be accessed, its rules can be applied, and its plans can be executed. In contrast, an instance of a private module can only be executed (its goals and beliefs cannot be accessed). The modules used to initialize and create individual agents are private modules as an agent’s internals cannot be accessed by any other module instance. Singleton modules are introduced to allow different agents to use one instance of a module, though not at the same time. An instance of a singleton module can be created and released by an agent. The state of the singleton module instance is preserved after the release operation such that the subsequent creation of the same singleton module instance (possibly by a different agent) results a module instance with the same state as it was released.
Create Action

The create(mod-name, mod-ident) action can be used to create an instance of a module that is declared in the multi-agent program with the name mod-name. The name that is assigned to the created module instance is given by the second argument mod-ident. A module instance can be created either by an individual agent or by a module instance (which is originally created by an agent). A created module instance can conceptually be considered as constituting (being part of) the agent by which it is created. In the sequel, we say that a module instance is created by an agent if it is directly created by the agent or indirectly by a module instance that is originally created by that agent. An agent (or module instance) that creates a module instance becomes its owner. The owner of a module instance can use the name mod-ident to perform further operations on it. An agent (or a module instance created by it) can create several instances of one and the same module. Also, two agents (or module instances) can create two instances of one and the same module. The owner of a module instance is the only entity that can operate on the created module instance until the created module instance is released.

Release Action

A module instance m can be released by means of the release(m) action. If the module is not a singleton, then the state of its instance will be lost after it is released. However, if the module is a singleton, then the state of its instance will be maintained such that it can be used to instantiate an instance of the same module when it is created again. It is important to note that a singleton module can only have one instance at a time such that it can always be accessed by means of the module name mod-name. It is also important to note that the subsequent creation of an instance of a singleton module, which may assign different names to the created instances, will refer to the same instance of the module as when it was released by its previous owner.

Execute Actions

When an instance of a public or private module m is created, it can be executed by its owner through the action m.execute(<test>) or m.executeasync([<test>]). The execution of a module instance by means of the execute operation starts the 2APL deliberation process based on the internals of the module instance m. The execution of a module instance can thus result in applying the rules of the module instance and executing its plans. The execution thread of the owner (who performs the execute operation) halts until the execution of the owned module instance halts and a return event is received from the owned module instance. In order to notify a module instance that it should stop its execution (the owning module cannot do this because its execution is halted) the test condition (i.e., the argument of the execute action) is evaluated by the overall multi-agent system interpreter and a stop event stop! is generated and sent to the module instance. The module instance that receives the stop event may start a clean-up operation and send a return event back when it is ready. We introduce a return action that can be executed by a module instance as the last action after which its execution is halted. The execution of this action broadcasts an event return! that can be received by the overall multi-agent system interpreter after which 1) the owner is notified about the return event, and 2) the execution (i.e., the deliberation process) of the module’s owner is started again. After the reception of the return event, the owner may decide to release the owned module.

The execution of a module instance by means of the executeasync action is identical to execute, except that the owner does not have to wait until the execution of the module instance halts. In fact, the owner continues with its own execution thread (deliberation process) in parallel
Modular Rule-Based Programming in 2APL

with the execution of the owned module instance. The fact that there will be different threads of activities should not be considered as a problem as an agent may generate different behaviors in parallel. The execution of the module instance can be halted by providing a test expression as the argument of the executeasync action, or by performing the stop action on the module instance. Like the execute action, the test will be evaluated at the multi-agent system interpreter and based on the internals of the module instance. A dedicated stop action can be performed by the owner, which will send the stop! event to the owned module.

Test and Update Actions

The owner of a public module instance can access and update the internals of the module instance. In particular, it can be tested whether certain beliefs and goals are entailed by the beliefs and goals of a public module instance \( m \) through action \( m.B(\phi) \land G(\varphi) \). Also, the beliefs of a public module instance \( m \) can be updated by means of \( m.updateBB(\phi) \) action. A goal can be added to the goals of a public module instance \( m \) by means of \( m.adopta(\phi) \) and \( m.adoptz(\phi) \) actions. Finally, the goals of a public module instance \( m \) can be dropped by means of \( m.dropgoal(\phi) \), \( m.dropsubgoals(\phi) \) and \( m.dropsupergoals(\phi) \) actions. As explained in (Dastani et al., 2007, Dastani, 2008), these actions can be used to drop from an agent’s goal base, respectively, all goals identical to \( \phi \), all goals that are a logical subgoal of \( \phi \), and all goals that have \( \phi \) as a logical subgoal.

Given our simple example, the following code implements the creation of an instance of a public module with the name carrier.

```plaintext
create(carrier, mycar);
mycar.updateBB( gold(L) );
mycar.execute( B(done() or error()) );
...
```

The unique identifier that is assigned to the created module is mycar. Subsequently, the belief base of the created module instance is updated with the information about the gold items (the variable \( L \) stands for a list of gold items). After this belief update, the module instance is executed and possibly other operations (e.g., communication or external actions) are performed. Note that the module instance mycar will be halted when it believes it has done the task of carrying gold items, or it believes an error has been occurred.

Plans

In order to reach its goals, a 2APL agent adopts plans. A plan consists of basic actions composed by sequence operator, conditional choice operator, conditional iteration operator, and a non-interleaving operator. The sequence operator \( ; \) is a binary operator that takes two plans and generates one <sequenceplan> plan. The sequence operator indicates that the first plan should be performed before the second plan. The conditional choice operator generates <ifplan> plans of the form if \( \varphi \) then \( \pi_1 \) else \( \pi_2 \), where \( \pi_1 \) and \( \pi_2 \) are arbitrary plans. The condition part of this expression (i.e., \( \varphi \)) 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., \( \pi_1 \)) when the test \( \varphi \) succeeds, otherwise perform the else-part of the plan (i.e., \( \pi_2 \)). The conditional iteration operator generates <whileplan> plans of the form while \( \varphi \) do \( \pi \), where \( \pi \) is an arbitrary plan. The condition \( \varphi \) 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 \( \pi \) as long as the test \( \varphi \) succeeds. The (unary) non-interleaving operator generates <atomicplan> plans, which are expressions of the form \([\pi]\), where \( \pi \) is an arbitrary plan. This plan is interpreted as an atomic plan \( \pi \), which should be executed at once ensuring that the execution of \( \pi \) is not interleaved with 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 \([\pi_1;\pi_2]\) and \([\pi_2;\pi_1]\) 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 initial plan base of a 2APL module.

The first plan is an atomic plan ensuring that the database is updated with the gold items \(L_1\) that the worker agent \(w_1\) should collect, and the gold items \(L_2\) that the worker agent \(w_2\) should collect. These operations are done as atomic and in a non-interleaving mode in order to prevent the database to be accessed between these two operations. The second plan is a single action by which the worker \(w_1\) is asked to play the explorer role. A plan in the plan base may change during the execution of the agent.

```
Plans:
    [ @database(addGold(L1,w1),_) ;
      @database(addGold(L2,w2),_) ],
    send( w1,request,play(exp) )
```

### Practical Reasoning Rules

The 2APL programming language provides constructs to implement practical reasoning rules that can be used to implement the generation of plans during an agent’s execution. In particular, three types of practical reasoning rule are proposed: planning goal rules (generate plans for achieving goals), procedure call rules (generate plans to process internal and external events including received messages), and plan repair rules (generate plans to replace failed plans). These rules are introduced to model and implement the interaction between different agent concepts such as beliefs, goals, events, and plans. 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 when 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 goal and belief query expressions used to check if the agent has a certain goal and belief, respectively. The body of the rule is 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 entailed by 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 because 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 store gold is to pick up a gold item in the blockworld first, store them in the same blockworld, and finally update the belief base to remove the information about the gold item from the belief base.

```
PG-rules:
    goldStored()  <-  gold([X|R])  |
      { @blockworld(pickUpGold(X),_);  
        @blockworld(storeGold(X),_);  
        Remove(X)  }
```
Procedure Call Rules (PC-Rules)

The procedure call rule is introduced for various reasons and purposes. Besides their use as procedure definition (used for executing abstract actions; see previous section on Basic Actions), they can also be used to respond to messages and to handle external events. In fact, a procedure call rule can be used to generate plans as a response to 1) the reception of messages send by other agents, 2) the reception of events generated by the external environments, and 3) the execution of abstract actions. Like planning goal rules, the specification of procedure call rules consists 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, an event, or if it executes an abstract action, and moreover, the belief condition of the rule is entailed by the agent’s beliefbase. The resulted substitution for variables is applied in order to instantiate the generated plan. A procedure call rule is of the form:

```
<atom> "-" <belquery> "|" <plan>
```

Box 2 displays examples of procedure call rules. The first rule indicates that if the manager agent `A` sends a request message to play the explorer functionality (or role), then an instance of the explorer module is created and executed until some gold items are found (note that the halt condition is to have the belief `gold(L)`). The gold items are then reported to the manager agent by a send action. Before releasing the module instance, the information about the gold items is administrated by means of the abstract action `adminGold(L)`. This action is processed by another PC-rule with the same atom as head. Note also the recursive use of the PC-rule that administrates the list of gold items by means of the belief update action `GoldInf(X)` which adds the atom `gold(X)`.

```plaintext
Box 2.

PC-rules:
message(A,request,play(exp)) <- manager(A) | {
    create(explorer, myexp);
    myexp.execute( B(gold(L)) );
    send(A, inform, gold(L));
    adminGold(L);
    release(myexp)
}

adminGold(L) <- true | {
    if B( L=[X|R] ) then
    { GoldInf(X);
      adminGold(R)
    }
    else skip
}
event(stop) <- true | { Remove();return }
```
to the belief base (See section Basic Actions for the formal specification of this action). The last PC-rule is programmed to execute a clean-up operation followed by the return action when the stop event is received.

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 instantiated with plans. 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 in the head of a plan repair rule) fails and the agent has a certain belief, then the failed plan could be replaced by another plan (i.e., by an instantiation of the abstract plan in the body of the plan repair rule). 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 condition of the rule is entailed by 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 \( \pi_1, \pi_2, \pi_3 \) are plans and \( X \) is a plan variable, then the abstract plan \( \pi_1;X;\pi_2 \) can be unified with the failed plan \( \pi_1;\pi_2;\pi_3 \) resulting the substitution \( X=\pi \). The resulted substitution will be applied to the abstract plan expression in the body of the rule 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 \( @blockworld(pickUpGold(E),_ ) \) fails, then the plan should be replaced by a plan that performs a belief update action to administrate that an error has been occurred. The rest of the plan unified with the plan variable \( X \) will be deleted. We assume the action \( Error() \) to be specified by the agent programmer. This repair can be done without a specific belief condition.

PR-rules:

\[
@blockworld(pickUpGold(E),_ );X <- true | \{ Error() \}
\]

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 the action. The execution of a belief update action fails if the pre-condition of the action is not entailed by the belief base or if the action is not specified. An abstract action fails if there is no applicable procedure call rule. An external action fails if the corresponding environment throws an ExternalActionFailedException (see next section for more details) or if the agent has no access to that environment or if the action is not defined in that environment. A test action fails if the test expression is not entailed by the belief and goal bases. A goal adopt action fails if the goal is already entailed by the belief base or the goal to be adopted is not ground. Finally, an atomic plan fails if one of its actions fails. The executions of all other actions are always successful. When the execution of an action fails, then the execution of the whole plan is blocked. The failed action will not be removed from the failed plan such that it can be repaired.

Programming External Environment

An agent can perform actions in different external environments, each implemented as a Java class. In
particular, any Java class that implements the 2APL
environment interface can be used as a 2APL
environment. The 2APL environment interface
contains two methods, addAgent(String
name) to add an agent to the environment and
removeAgent(String name) to remove
an agent from the environment. The construc-
tor of the environment must require exactly one
parameter of the type ExternalEventListener. This object listens to external events. The
execution of action $@\text{env}(m(a_1,\ldots,a_n),R)$ calls
a method $m$ with arguments $a_1,\ldots,a_n$ in environ-
ment $\text{env}$. 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 $m$ is ready and the
return value is accessible to the rest of the plan.
Methods may throw the special exception
ExternalActionFailedException. If they
throw this exception, the corresponding external
action is considered as failed. The following is an
example of the method $\text{move}$ that can be called
by executing an external action (with the same
name) (see Box 3).

### Events and Exceptions

The main use of events is to pass information
from environments to agents. When implement-
ing a 2APL environment, the programmer should
decide when and which information from the
environment should be passed to agents. This can
be done in an environment by calling the method
notifyEvent(AF event, String agents) in the ExternalEventListener
which was 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 represent
local names of agents. The events can be received
by agents whose name is listed 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 receive the
event. Such a mechanism of generating events
by the environment and receiving them 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

### Box 3.

```java
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();
}
```
MULTI-AGENT SYSTEM EXECUTION AND AGENT DELIBERATION

The execution of a 2APL multi-agent system consists of the executions of individual agents and the environments in parallel (in an interleaving mode). Moreover, the execution of each 2APL individual agent is determined by the so-called the agent’s deliberation process. This is a cyclic process that implements the sense-reason-act behavior of individual agents and constitutes the agent’s interpreter (see figure 2). In order to execute an individual agent, the 2APL interpreter follows a certain order of deliberation steps repeatedly and indefinitely. Each cycle starts by applying all applicable PG-rules, each rule only one time. Note that a PG-rule can be applied more than once since it can be applied to generate plans for two different goals. For example, for an agent with two goals “g(a)” and “g(b) and g'(c)”, the PG-rule “g(X) <- b(Y) | π(X,Y)” can be applied to both “g(a)” as well as “g(b) and g'(c)”. However, according to the 2APL deliberation cycle the PG-rule will be applied to the first goal “g(a)” in one cycle and to the goal “g(b) and g'(c)” in the next cycle. Note that the goal base is a list imposing an order on the goals.

The deliberation cycle proceeds by executing only the first action of each plan. This is in order to make the deliberation process fair with respect to the execution of all plans, i.e., in order to allow all plans to get a chance to be executed. The next deliberation steps are to process all events received from the external environments, all exceptions indicating the failure of plans, and all messages received from other agents, respectively. An event from an external environment is processed by applying the first applicable PC-rule (with the head event) to it. An exception, which identifies a failed plan, is processed by applying the first applicable PR-rule to the failed plan. A received message is then processed by applying the first applicable PC-rule (with the head message) to it. Note that the application of rules to process events and exceptions generates and add plans to the corresponding agent’s plan base.

After these deliberation steps, it is checked if it makes sense to do a new cycle of deliberation steps. In fact, if in a deliberation cycle no rule could be applied, no plan could be executed, and no event could be processed, then it makes no sense to try again a new cycle of deliberation steps, except when a new event or message has arrived. The 2APL deliberation cycle is illustrated in Figure 5.

Of course, the order of the deliberation steps in a cycle can be defined in different ways. For example, at each deliberation cycle one may apply only one applicable PG-rule (instead of applying all applicable PG-rules), execute one plan (instead of the first action of all plans), and process one event. It is also possible to process events before applying PG-rules or execute plans after processing events. We selected the deliberation cycle as presented in Figure 5 because it has some interesting properties. An example of such a property is that a plan, the execution of which is failed, will be repaired in the same deliberation cycle or re-executed in the next deliberation step.

Figure 5. The 2APL deliberation cycle

43
This realizes a repair or retries behavior when plans are executed.

**EXAMPLE REVISITED**

In the Multi-Agent Programming section, we gave an example of a 2APL multi-agent program that includes four modules and implements a multi-agent system consisting of three agents: one manager and two workers agents. In previous sections, we gave examples to illustrate various programming constructs that can be used to implement modules. In this section, we give the complete implementation of all four modules and discuss the behavior of the three agents when the multi-agent system is executed.

As illustrated below, the goal of the manager \( m \) is to have gold items. Moreover, it has one initial plan through which it sends a request to worker \( w_1 \) to explore the blockworld environment. The first PC-rule of the manager agent indicates that when it receives a list of detected gold items (i.e., \( \text{gold}(L) \)) from worker \( w_1 \), then it divides the received list into two equal lists of gold items and stores them in its database (the manager agent could also add these lists to its belief base, but the aim of the example is to show the use of different environments). The second PC-rule of the manager indicates that when a worker informs that it is ready with collecting its corresponding gold items, then the manager removes the goal items from its database and updates its belief base with the fact that the worker is ready to carry new gold items again. In order to achieve its goal, the manager agent checks its database to see if it has information about gold items to be collected by one of the worker agents that is not carrying gold (note that the information about gold items should be received from the worker agent that initially was asked to explore the blockworld). This is done applying its PG-rule whenever possible. If it can find such information (non-empty list of gold items) from its database, then it will send a request to the corresponding agent asking to carry the gold and store them in the blockworld. The manager agent will update its belief base with the information that the agent is busy carrying a gold item.

The worker agent is an agent that waits for requests to either explore the blockworld environment or carry the gold items and store them. When it receives a request to explore the blockworld environment from the manager, then it creates an explorer module instance and executes it (i.e., it plays the explorer role). Note that the halting condition of this module instance is the belief that gold items are detected. When the execution of the module instance is halted, then the worker agent sends the information about the detected gold items to the manager, updates its beliefs with the information about its detected gold items (this action illustrates the use of abstract action), and finally releases the explorer module instance. The third PC-rules implements the execution of the abstract \( \text{adminGold}(L) \) by going recursively through the list of gold items and adds each of them to its belief base. Finally, the second PC-rule of the worker agent is responsible for carrying gold items by creating a carrier module instance, adding the gold item information to its belief base, and executing it until either it has found the gold items (\( \text{done()} \) condition) or an error has been occurred (\( \text{error()} \) condition).

The explorer module, which is a public module, has the goal to find gold items. In order to achieve this goal, it performs a sense-gold action in the blockworld and adds the information about the detected gold items (i.e., \( \text{gold}(L) \)) to its belief base. Note that this belief information is the halting condition of the module instance. The final PC-rule is to react to the \( \text{stop} \) event that will be received as soon as the module instance is released by the worker agent. The reception of this event performs a clean-up operation by deleting all information about gold items from its belief base and performs a \( \text{return} \) action. This action gives the execution back to the worker agent that had
released this module instance. Note that the goal foundGold() is achieved as soon as gold(L) is added to its belief base.

Finally, the carrier module (also a public module) has a goal to store a list of gold items safely. This goal can be achieved by picking one gold item from the list, store it in the blockworld, and remove that gold item from the list of stored gold items. Note the use of two PG-rules to handle empty and non-empty lists of gold items. Similar to the explorer module, the carrier module does a clean-up operation and performs the return action when it receives a stop event.

The plan repair rule adds error information (i.e., error()) to the belief base when the execution of the pickUpGold action in the blockworld environment fails. Note that error() in the belief base was one of the halting conditions to stop the execution of the carrier module instance. It is also important to note that it is up to the blockworld programmer to determine when the execution of the pickUpGold action fails.

Manager.2apl

Worker.2apl

Explorer.2apl
CONCLUSION AND FUTURE RESEARCH

In this chapter, we presented the syntax of a modular and rule-based multi-agent programming language, explained informally the semantics of its programming constructs, and illustrated the use of the programming language by a simple example of a 2APL multi-agent program. The formal (operational) semantics of 2APL without modules is presented elsewhere (Dastani, 2008). The formal semantics of some of the module related constructs is presented in (Dastani et al., 2008). In fact, the formal semantics of the module related constructs is not included in this chapter because the main purpose of this chapter was to present 2APL rather than its formal semantics and in order to make this chapter accessible to a broader audience. A key characteristic of 2APL is the use of three different types of rules for generating plans. In our opinion, these rules realize an effective integration of declarative and imperative programming styles as they relate imperative plans with declarative beliefs, goals, and events. Another characterizing feature of 2APL is related to the use of variables in general, and their role in the integration of declarative and imperative programming in particular. In fact, 2APL allows agents to query their beliefs and goals, and pass the resulting substitutions to actions and plans in order to modify their external environments.

We have already implemented an interpreter that executes 2APL multi-agent programs without modules. This interpreter is implemented in Java and makes use of a Prolog engine that is also implemented in Java. This Prolog engine is used to implement the belief and goal bases of individual agents, and to implement the belief and goal test actions as Prolog queries. Finally, since the interpreter is implemented in Java, the connection with Java-based external environments is easy and straightforward. This interpreter is integrated in the 2APL development platform that can be downloaded from http://www.cs.uu.nl/2apl. We are working to extend this interpreter and its corresponding development platform with modules and module related programming constructs. The current platform provides a graphical interface through which an agent programmer can load, edit, run, and debug a 2APL multi-agent program. Moreover, a state tracer tool is provided through which one can browse through an agent’s states generated by the execution of its corresponding agent program. For each state, one can observe the agent’s beliefs, goals, plans, and deliberation actions. Finally, the platform allows communication among agents that run on several machines connected in a network. Agents hosted on different 2APL platforms can communicate with each other. The 2APL platform is built on the top of JADE (http://jade.tilab.com), which is a software framework that facilitates the implementation of multi-agent systems through a middleware that complies with the FIPA specifications.

It should be emphasized that 2APL is not designed for specific multi-agent system applications. We have used 2APL in various academic courses, research projects, and to participate in the Agent Contest (Astefanoaei et al., 2008). In
particular, we have used 2APL to implement different auction types (e.g., English and Dutch auctions), negotiation mechanisms (e.g., contract net and monotonic concession protocol), cooperative problem solving tasks (e.g., multi-agent surveillance system), and to control robots such as iCat (Vergunst et al., 2007). Despite these applications, we believe that a real evaluation of the proposed programming language should be done by the agent community and by using it for more sophisticated applications.

We are currently working on various extensions of both 2APL programming language as well as development tools to be integrated in the 2APL platform. Our primary aim is to introduce programming constructs at multi-agent level to allow the implementation of social and organizational issues such as norms, obligations, prohibition, permission, power relation, delegation of tasks, responsibility, and trust. These constructs should enable a programmer to implement under which condition individual agent are allowed to perform actions and what are the consequences of such actions. This can be realized by adding an organization component (specified by the introduced constructs at multi-agent level) to the multi-agent configuration. The idea is to check this component before an individual agent performs an action and to updates it after the action is performed. A preliminary version of such an extension is presented in (Tinnemeier et al., 2008, Dastani et al., 2008).

REFERENCES


Cohen, P. R., & Levesque, H. J. (1990). Intention is choice with commitment. Artificial Intelligence, 42.


Modular Rule-Based Programming in 2APL


Agent: An agent is a computational system that is situated in some dynamic environment and can autonomously decide which actions to perform in order to achieve its objectives.

Agent Deliberation: An agent deliberation is a cyclic process through which the agent continuously senses its environment (processing the received internal and external events), updates its state, reasons about its state to decide which actions to perform, and executes those actions. Such a cyclic process constitutes the interpreter of agent programming languages.

Agent Profile: An agent profile is the representation of an agent. An agent can build and maintain the profile of other agents. In a modular agent programming language, an agent profile may be implemented by means of a module.

Agent Role: An agent role is a specific functionality or task that an agent can realize. In a modular agent programming language, an agent role may be implemented by a specific module.

BDI Agents: A BDI agent decides its actions based on its mental state consists of beliefs, goals, events, and plans. In particular, it decides actions that he believes they either contribute to the achievement of its goals or react to its received events and messages.

BDI-Based Agent Programming Language: A BDI-based agent programming language is designed to support the implementation of BDI agents. Such a programming language provides constructs to implement an agent’s beliefs, goals, events, plans, and decisions making process.

Modular Agent Programming Language: An agent programming language is modular if it supports the implementation of agents in terms of separate modules each of which implements a separation of concern.

ENDNOTES

1 http://www.fipa.org
2 http://jade.tilab.com