Auton Agent Multi-Agent Syst (2008) 16:214–248
DOI 10.1007/s10458-008-9036-y

2APL: a practical agent programming language

Mehdi Dastani

Published online: 16 March 2008

Abstract This article presents a BDI-based agent-oriented programming language, called 2APL (A Practical Agent Programming Language). This programming language facilitates the implementation of multi-agent systems consisting of individual agents that may share and access external environments. It realizes an effective integration of declarative and imperative style programming by introducing and integrating declarative beliefs and goals with events and plans. It also provides practical programming constructs to allow the generation, repair, and (different modes of) execution of plans based on beliefs, goals, and events. The formal syntax and semantics of the programming language are given and its relation with existing BDI-based agent-oriented programming languages is discussed.

1 Introduction

Multi-agent systems constitute a promising software engineering approach for the development of distributed intelligent systems. The agent-oriented software engineering paradigm provides social and cognitive concepts and abstractions in terms of which software systems can be specified, designed, and implemented. Examples of such concepts and abstractions are roles, organisational rules and structures, communication, norms, beliefs, goals, plans, actions, and events. In order to develop multi-agent systems in an effective and systematic way, different analysis and design methodologies [4], specification languages (e.g., BDI_{CTL} [7,22] and KARO [19]), and programming languages and development tools [5,6,12–14,17,18,20,24,25,30] have been proposed. While most agent-oriented analysis and design methodologies assist system developers to 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 agent concepts and abstractions.

M. Dastani (✉)
Intelligent Systems Group, Utrecht University, Utrecht, The Netherlands
e-mail: mehdi@cs.uu.nl

Existing BDI-based agent-oriented programming languages differ in detail, despite their apparent similarities. They differ from each other in the way that they provide programming constructs for specific, sometimes overlapping, sets of agent concepts and abstractions. The expressivity of the programming constructs for overlapping agent concepts and abstractions may differ from one programming language to another. This is partly due to the fact that they are based on different logics and use different technologies. Moreover, some of these programming languages have formal semantics, but others provide only an informal explanation of the intended meaning of their programming constructs. Also, some programming languages capture in their semantics specific rationality principles that underlie agent concepts, while such principles are assumed to be implemented by agent programmers in other programming languages. Finally, some agent-oriented programming languages are based on declarative style programming, some are based on imperative style programming, and yet others combine these programming styles.

The aim of this work is to propose a BDI-based agent-oriented programming language that (1) 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, (2) creates a balance between the expressivity of programming constructs designed to represent and reason with agent concepts, and the expressivity of constructs designed to implement the dynamics (update and revision) of those concepts, (3) has formal semantics such that it is possible to verify whether agent programs satisfy their (formal) specifications, (4) captures important and intuitive rationality principles underlying agent concepts, and (5) realizes an effective integration of declarative and imperative programming styles. In our view, 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 to facilitate practical and effective implementation of agent related concepts and abstractions.

The structure of the article is as follows. In the next section, we present a general description of a BDI-based agent-oriented programming language called 2APL (A Practical Agent Programming Language) and discuss some of its characterizing features. In Sect. 3, we provide the complete syntax of 2APL and explain the intuitive meaning of its ingredients. In Sect. 4 the formal semantics of the programming language is given. We define in Sect. 5 possible executions of 2APL agents. In Sect. 6, we compare 2APL with some existing BDI-based agent-oriented programming languages and discuss their similarities and differences. Finally, we conclude the article in Sect. 7 and discuss some future directions to extend this programming language.

2 2APL: a general description

One of the features of 2APL is the separation between multi-agent and individual agent concerns. 2APL provides two distinguished sets of programming constructs to implement *multi-agent* and *individual agent* concepts. The multi-agent programming constructs are designed to create individual agents and external environments, assign unique names to individual agents, and specify the agents' access relations to the external environments. In 2APL, individual agents can be situated in one or more environments, which are assumed to be implemented as Java objects. Each environment (Java object) has a state and can execute a set of actions (method calls) to change its state. 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 allows us to design and

add a set of programming constructs to implement a wide range of social and organizational concepts in a principled and modular way.

At the individual agent level, 2APL agents are implemented in terms of beliefs, goals, actions, plans, events, and three different types of rules. The beliefs and goals of 2APL agents are implemented in a declarative way, while plans and (interfaces to) external 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 mental 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.

2APL agents can perform different types of actions such as belief update actions, belief and goal test actions, external actions (including sense actions), actions to manage the dynamics of goals, and communication actions. Plans consist of actions that are composed by a conditional choice operator, iteration operator, sequence operator, or non-interleaving operator. The first type of rule is designed to generate plans for achieving goals, the second type of rule is designed to process (internal and external) events and received messages, and the third type of rule is designed to handle and repair failed plans. It should be noted that a 2APL agent can observe an environment either actively by means of a sense action or passively by means of events generated by the environment. More sophisticated model of environments are discussed in [29,23].

A 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. Conversely, 2APL agents can observe their environments and update their beliefs and goals accordingly.

Another key feature of 2APL is the distinction between declarative goals and events. In 2APL, 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 that may trigger an agent to react and execute plans. As argued in [26], the use of declarative goals in an agent programming language adds flexibility in handling failures. If the execution of a plan does not achieve its corresponding declarative goal, then the goal persists and can be used to select a different plan. However, an event is usually used to select and execute a plan; the event is dropped just before or immediately after its corresponding plan is executed.

2APL also provides a programming construct to implement so-called atomic plans. Because an agent can have a set of concurrent plans, an arbitrary interleaving of plans may be problematic in some cases. A programmer may therefore want to specify that (a certain part of) a plan should be considered as atomic and executed at once without being interleaved with the actions of other plans.

Finally, it should be emphasized that 2APL is not designed for specific applications. We have used 2APL in various academic courses, research projects, and to participate in the Agent Contest [2]. 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 [28]. 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.

3 2APL: syntax

2APL is a multi-agent programming language that provides programming constructs to implement both multi-agent as well as individual agent concepts. The multi-agent concepts are implemented by means of a specification language. Using this language, one can specify which agents should be created to participate in the multi-agent system and to which external environments each agent has access. The syntax of this specification language is presented in Fig. 1 using the EBNF notation. In the following, we use $\langle ident \rangle$ to denote a string and $\langle int \rangle$ to denote an integer.

In this specification, $\langle agentname \rangle$ is the name of the individual agent to be created, $\langle filename \rangle$ is the name of the file containing the 2APL program that specifies the agent to be created, and $\langle int \rangle$ is the number of agents that should to be created. When the number of agents is $n > 1$, then n identical agents are created. The names of these agents are $\langle agentname \rangle$ extended with a unique number. Finally, $\langle environments \rangle$ is the list of environment names to which the agent(s) have access. Note that this language allows one to create a multi-agent system consisting of different number of different agents. The following is an example of a multi-agent system specification in which one explorer agent and three carrier agents are created. In this example, the explorer agent does not have access to any environment while the carrier agents have access to the `blockworld` and `itemdatabase` environments. The specification of a multi-agent system constitutes a multi-agent program, stored in a file with `.mas` extension.

```
explorer   : explorer.2apl
carrier    : carrier.2apl    3   @blockworld, itemdatabase
```

Individual 2APL agents are implemented by means of another specification language. The EBNF syntax of this specification language is illustrated in Fig. 2. In this specification, we use $\langle atom \rangle$ to denote a Prolog like atomic formula starting with lowercase letter, $\langle Atom \rangle$ to denote a Prolog like atomic formula starting with a capital latter, $\langle ground\_atom \rangle$ to denote a ground atom and $\langle Var \rangle$ to denote a string starting with a capital letter.

An individual 2APL agent program can be composed of various ingredients that specify different aspects of an individual agent. In particular, an agent can be programmed in 2APL by implementing the initial state of these ingredients. The state of some of these ingredients will change during the agent's execution while the state of other ingredients remains the same during the execution of the agent. The program of an individual agent is stored in a file with `.2apl` extension. In the following, we will discuss each ingredient and give examples to illustrate them.

3.1 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

$\langle MAS\_Prog \rangle$ = ($\langle agentname \rangle$ `":"` $\langle filename \rangle$ [$\langle int \rangle$] [$\langle environments \rangle$])$^+$;
$\langle agentname \rangle$ = $\langle ident \rangle$;
$\langle filename \rangle$ = $\langle ident \rangle$`".2apl"` ;
$\langle environments \rangle$ = `"@"`$\langle ident \rangle$ {`","`$\langle ident \rangle$} ;

Fig. 1 The EBNF syntax of 2APL multi-agent systems

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

Fig. 2 The EBNF syntax of 2APL individual agents

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

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

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 five gold items and the `blockworld` is clean. Note that this single conjunctive goal is different than having two separate goals '`hasGold(5)`' and '`clean(blockworld).`' In the latter case, the agent wants to achieve two situations independently of each other, i.e., one in which the agent has a clean `blockworld`, not necessarily with a gold item, and one in which it has five gold items and perhaps a `blockworld` which is not clean. The second goal of the agent indicates that the agent desires a state in which it has 10 gold items, independent of the state of the environment. Note that different goals in the goal base are separated by a comma.

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

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 as well. An agent's beliefs and goals change during the agent's execution.

3.2 Basic actions

Basic actions specify capabilities of agents, i.e., actions that an agent can perform to achieve its desirable situation. 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.

3.2.1 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 store temporarily 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*⟩.

```
BeliefUpdates:

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

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

3.2.2 Test action

A *test action* performed by an agent checks whether the agent has certain beliefs and goals. A test action is an expression of the form ⟨*test*⟩ consisting of belief and goal query expressions. A belief query expression has the form `B(`ϕ`)`, where ϕ consists of *literals* composed by conjunction or disjunction operators. A goal query expression has the form `G(`ϕ`)`, where ϕ consists of *atoms* composed by conjunction or disjunction operators. A belief query expression is basically a (Prolog) query to the belief base and generates a substitution for the variables that are used in the belief query expression. A goal query expression 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.

3.2.3 Goal dynamics actions

The *adopt goal* and *drop goal* actions are used to adopt and drop a goal to and from an agent's goal base, respectively. The adopt goal action ⟨*adoptgoal*⟩ can have two different forms: `adopta(`ϕ`)` and `adoptz(`ϕ`)`. These two actions can be used to add the goal ϕ

(a conjunction of atoms) to the 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 ϕ are instantiated before these actions are executed since the goal base should contain only ground formula. Finally, the drop goal action $\langle dropgoal \rangle$ can have three different forms: `dropgoal(`ϕ`)`, `dropsubgoals(`ϕ`)`, and `dropsupergoals(`ϕ`)`. These actions can be used to drop from an agent's goal base, respectively, the goal ϕ, all goals that are a logical subgoal of ϕ, and all goals that have ϕ as a logical subgoal. Note that the action dropsupergoals is also proposed in [16].

3.2.4 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 Sect. 3.4.2 for a description of PC-rules). As we will see in Sect. 3.4.2, 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 $\langle abstractaction \rangle$ is an expression of the form $\langle atom \rangle$ (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.

3.2.5 Communication action

A *communication action* passes a message to another agent. A communication action $\langle sendaction \rangle$ can have either three or five parameters. In the first case, the communication action is the expression `send(Receiver, Performative, Language, Ontology, Content)` where `Receiver` is 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 FIPA compliant JADE platform. For this reason, the name of the receiving agent can be a local name or a full JADE name. A full JADE name has the form `localname@host:port/JADE` where `localname` is the name as used by 2APL, `host` is the name of the host running the agent's container and `port` is the port number where the agent's container should listen to (see [3] for more information on JADE standards).

3.2.6 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(east(),L)` (go one step east in the blockworld environment). The effect of this action is that the position of the agent in the blockworld environment is shifted one slot to the right. The list `L` is expected as the return value. The programmer determines the content of this list.

3.3 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 ⟨*if plan*⟩ plans of the form `if` ϕ `then` π_1 `else` π_1, where π_1 and π_2 are arbitrary plans. The condition part of this expression (i.e., ϕ) is a test that should be evaluated with respect to an agent's belief and goal bases. Such a plan can be interpreted as to perform the if-part of the plan (i.e., π_1) when the test ϕ succeeds, otherwise perform the else-part of the plan (i.e., π_2). The conditional iteration operator generates ⟨*whileplan*⟩ plans of the form `while` ϕ `do` π, where π is an arbitrary plan. The condition ϕ is also a test that should be evaluated with respect to an agent's belief and goal bases. The iteration expression is then interpreted as to perform the plan π as long as the test ϕ succeeds.

The (unary) non-interleaving operator generates ⟨*atomicplan*⟩ plans, which are expressions of the form `[`π`]`, where π is an arbitrary plan. This plan is interpreted as an atomic plan π, which should be executed at once ensuring that the execution of π is not interleaved with actions of other plans. Note that an agent can have different plans at the same time and that plans cannot be composed by an explicit parallel operator. As there is no explicit parallel composition operator, the nested application of the unary operator has no effect, i.e., the executions of plans `[`π_1`;`π_2`]` and `[`π_1`;[`π_2`]]` generate 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 agent. The first plan is an atomic plan ensuring that the agent updates its belief base with its initial position `(5,5)` immediately after performing the external action `enter` in the `blockworld` environment. The second plan is a single action by which the agent requests the administrator to register him. An agent's plans may change during the execution of the agent.

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

3.4 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*, *procedure call rules*, and *plan repair rules*. In the following subsections, we explain these three types of rules.

3.4.1 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 $\langle pgrule \rangle$ 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 may 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: [$\langle goalquery \rangle$] "<-" $\langle belquery \rangle$ " | " $\langle plan \rangle$.

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

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

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

3.4.2 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), 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 sent 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 consist of three entries. The only difference is that the head of the procedure call rules is an atom $\langle atom \rangle$, rather than a goal query expression $\langle goalquery \rangle$. 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 belief base. The resulted substitution for variables are applied in order to instantiate the generated plan. A procedure call rule is of the form: ⟨*atom*⟩ "<-" ⟨*belquery*⟩ "|" ⟨*plan*⟩. The following are examples of procedure call rules.

```
PC-rules:
  message(A,inform,La,On,goldAt(X2,Y2)) <-  not carry(gold) |
         { getAndStoreGold(X2,Y2) }

  event(gold(X2,Y2),blockworld) <-  not carry(gold) |
        { getAndStoreGold(X2,Y2) }

  getAndStoreGold(X,Y) <- pos(X1,Y1) |
        { [goTo(X1,Y1,X,Y);@blockworld(pickup(),_);PickUp();
           goTo(X,Y,3,3);@blockworld(drop(),_);StoreGold()]
        }
```

The first rule indicates that if an agent `A` informs that there is some gold at position `(X2,Y2)` and the agent believes it does not carry a gold item, then the agent has to get and store the gold item in the depot. The second rule indicates that if the environment `blockworld` notifies the agent that there is some gold at position `(X2,Y2)` and the agent believes it does not carry a gold item, then the agent has to do the same, i.e., get and store the gold item in the depot. Finally, the last rule indicates that the abstract action `getAndStoreGold` should be performed as a certain sequence of actions, i.e., go from its current position (obtained through the condition of the rule) to the gold position, pick up the gold item, go to the depot position (i.e., position `(3,3)`), and store the gold item in the depot. The `PickUp()` and `StoreGold()` are belief update actions to administrate the facts that the agent is carrying gold and has certain amount of stored gold, respectively. The `goTo` action is also an abstract action that should be associated with a plan by means of PC-rules. Note that the plan is implemented as an atomic plan. The reason is that in this plan external actions and belief update actions are executed consecutively such that an unfortunate interleaving of other actions can have undesirable effects.

3.4.3 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 π, π_1, π_2 are plans and X is a plan variable, then the abstract plan $\pi_1; X; \pi_2$ can be unified with the failed plan $\pi_1; \pi; \pi_2$ resulting the substitution $X = \pi$. The resulted 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(east(),_);@blockworld(east(),_)` fails, then the plan should be replaced by a plan in which the agent first goes one step to north, then makes two steps to east, and goes one step back to south. This repair can be done without a specific belief condition.

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

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

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

3.5 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 *environment interface* can be used as a 2APL environment. The 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 constructor of the environment must require exactly one parameter of the type $ExternalEventListener$. This object listens to external events.

The execution of action `@env(m(`$a_1, \ldots, a_n$`),R)` calls a method `m` with arguments $a_1, \ldots, a_n$ in environment `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 a method that can be called by executing an external action.

```
public Term move(String agent, String direction)
   throws ExternalActionFailedException
{
   if (direction.equals("north") {moveNorth();}
   else if (direction.equals("east") {moveEast();}
   else if (direction.equals("south") {moveSouth();}
   else if (direction.equals("west") {moveWest();}
   else throw
    new ExternalActionFailedException("Unknown direction");
   return getPositionTerm();
}
```

3.6 Events and exceptions

Information between agents and environments can also be passed through *events* and *exceptions*. The main use of events is to pass information from environments to agents. When implementing 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 generation and throwing of exceptions. In fact, exceptions are semantical entities that cannot be used by 2APL programmers.

3.7 Including 2APL files

Different agents may share certain initial beliefs, goals, plans, belief updates, and practical reasoning rules. The 2APL programming language provides an encapsulation mechanism to include these shared initial ingredients in different 2APL programs. This is done by allowing a 2APL program to be included in other 2APL programs through the `Include: filename` construct. The 2APL program that includes other 2APL programs inherits all ingredients of the included programs. This programming construct makes it possible to specify shared ingredients as one 2APL program.

4 2APL: semantics

In this section, we present the operational semantics of 2APL in terms of a transition system. A transition system is a set of transition rules for deriving transitions. A transition

is a transformation of one configuration into another and it corresponds to a single computation/execution step. In the following subsections, we first present the configuration of individual 2APL agent programs (henceforth agent configuration) which are defined in terms of mental attitudes such as beliefs, goals, events, plans and practical reasoning rules. Then, the configuration of multi-agent system programs (henceforth multi-agent system configuration) is presented, which consists of the configurations of the individual agents, the state of the external shared environments, and the agents' access relation to environments. Finally, we present transition rules from which possible execution steps (i.e., transitions) for both individual agents as well as multi-agent systems can be derived. The transition rules for individual agents capture transitions for plan execution and rule applications, while transition rules for multi-agent systems capture synchronized communication between agents and the execution of external actions in the shared environments.

4.1 2APL configuration

The configuration of an individual agent consists of its identifier, beliefs, goals, plans, specifications of belief update actions, practical reasoning rules, substitutions that are resulted from queries to the belief and goal bases, and the received events. Since the specification of practical reasoning rules and belief update actions do not change during an agent's execution, we do not include them in the agent's configuration. Moreover, additional information is assigned to an agent's plan. In particular, a unique identifier is assigned to each plan which can be used to identify and repair failed plans. Also, the practical reasoning rules by means of which plans are generated are assigned to plans in order to avoid redundant applications of practical reasoning rules, e.g., to avoid generating multiple plans for one and the same goal.

Definition 1 (*individual agent configuration*) The configuration of an individual 2APL agent is defied as $A_\iota = \langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle$ where ι is a string representing the agent's identifier, σ is a set of belief expressions $\langle belief \rangle$ representing the agent's belief base, γ is a list of goal expressions $\langle goal \rangle$ representing the agent's goal base, Π is a set of plan entries $(\langle plan \rangle, \langle pgrules \rangle, \langle ident \rangle)$ representing the agent's plan base, θ is a ground substitution[1] that binds domain variables to ground terms, and $\xi = \langle E, I, M \rangle$ is the agent's event base, where

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

Each plan entry is a tuple (π, r, p), where π is the executing plan, r is the instantiation of the PG-rule through which π is generated, and p is the plan identifier. It should be noted that the belief base and each goal in the goal base are consistent as only positive atoms are used to represent them. In the following, we use ξ_E, ξ_I and ξ_M to refer to E, I and M component of the event base ξ, respectively.

The configuration of a multi-agents system is defined in terms of the configuration of individual agents, the state of their shared external environments, and the agents' access relations

[1] A substitution is ground if the terms assigned to variables by the substitution do not contain variables.

to the external environments. Since the agents' access relations do not change during the execution of multi-agent systems, we do not include them in the corresponding configurations. The state of a shared environment is a set of facts that hold in that environment.

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

The initial configuration of a multi-agent system is determined by its corresponding multi-agent system program and consists of the initial configuration of its individual agents, the initial state of the shared external environments, and the agents' access relation to external environments. The initial configuration of each individual agent is determined by its corresponding 2APL program, i.e., by beliefs, goals, plans, belief update actions, and practical reasoning rules that are specified in the 2APL program. The components of the initial event base as well as the substitution of an 2APL agent are empty sets. These sets will be updated during the execution of the multi-agent system. The initial state of the shared external environment is set by the programmer. For example, the programmer may initially place gold items or trash at certain positions in a blockworld environment.

Definition 3 (initial configuration) Let i be the identifier of an agent that is implemented by a 2APL program. Let σ be the set of $\langle belief \rangle$-expressions specified in the 2APL program and all its included programs, γ be the list of $\langle goal \rangle$-expressions from the same programs, and Π be a set of $\langle plan \rangle$-entries (π, r, id), where π is a plan-expression from the same programs, $r = (\texttt{true<-true} \mid \texttt{skip})$,[2] and id is a new plan identifier. Then, the initial configuration of agent i is defined as tuple $\langle i, \sigma, \gamma, \Pi, \emptyset, \langle \emptyset, \emptyset, \emptyset \rangle \rangle$. Let also χ be a set of sets of facts, each of which represents the initial state of an environment specified in the multi-agent system program, and $A_1, \ldots, A_n$ be the initial configurations of agents $1, \ldots, n$ that are specified in the same multi-agent system program. The initial configuration of the multi-agent systems is defined as tuple $\langle A_1, \ldots, A_n, \chi \rangle$.

The execution of a 2APL multi-agent program modifies its initial configuration by means of transitions that are derivable from the transition rules presented in the following subsections. In fact, each transition rule indicates which execution step (i.e., transition) is possible from a given configuration. It should be noted that for a given configuration there may be several transition rules applicable. As we will discuss in the Sect. 5, an interpreter is a deterministic choice of applying transition rules in a certain order.

In Sects. 4.2, 4.3, and 4.4, we present the transition rules that capture the execution of basic actions, plans, and the application of practical reasoning rules of individual agents, respectively. Then, in Sect. 4.5 we present the transition rules for multi-agent systems. These rules capture the synchronized transitions of individual agents as well as the interactions between individual agents and the shared environments.

4.2 Transition rules for basic actions

A basic action can be executed at individual agent level. For some types of basic actions additional transition rules are given that specify when an action's execution fails and which transition an individual agent should make if the execution of the action fails. In general, when the execution of a basic action fails, an exception (i.e., its corresponding plan identifier)

[2] As we will see later on, assigning this rule to an *initial* plan makes it possible to execute the plan.

is added to the ξ_I component of the event base. This exception can later be used to repair the corresponding plan by means of plan repair rules. It should be noted that the application conditions of the transition rules that capture the success and the failure of a basic action exclude each other.

In the rest of this article, we will use $\models$ as a first-order entailment relation (we use Prolog engine for the implementation of this relation) and use $\gamma \models_g \kappa$ to indicate that there exists a goal expression in γ which entails the goal κ, i.e.,

$$\gamma \models_g \kappa \Leftrightarrow_{def} \gamma = [\gamma_1, \ldots, \gamma_i, \ldots, \gamma_n] \,\&\, \gamma_i \models \kappa \quad \text{for } 1 \leq i \leq n$$

We assume $\gamma \models_g \texttt{true}$ for any goal base γ. Moreover, we use $\gamma - \{\gamma_1, \ldots, \gamma_m\}$ to indicate that the elements of the set $\{\gamma_1, \ldots, \gamma_m\}$ are removed from the list γ. Finally, we will use $\mathcal{G}(r)$, $\mathcal{B}(r)$, and $\mathcal{P}(r)$ to indicate the head κ, the belief condition β, and the plan π of the rule $r = (\kappa \texttt{<-} \beta \mid \pi)$.

4.2.1 Skip action

The execution of `skip` action has no effect on an agent's configuration. The execution of this action always succeeds resulting in its removal from the plan base.

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

This transition rule indicates that the execution of the plan entry $(\texttt{skip}, r, id)$ proceeds resulting in the removal of this entry from the plan base. It is important to note that this transition rule can be applied only if the goal that is assigned to the `skip` action (or the plan in which this action occurs) is entailed by the agent's goal base. This condition, which is also the condition of other transition rules in this article, ensures that a plan is not executed if its purpose (i.e., the goal for which the plan is generated) is not desirable anymore.

4.2.2 Belief update action

A belief update action, which is specified in terms of a pre- and a post-condition, modifies the belief base when it is executed. In fact, a belief update action can be executed if its pre-condition is entailed by the belief base. After the execution of the action, its post-condition should be entailed by the belief base. The modification of the belief base to entail the post-condition is realized by adding positive literals of the post-condition to the belief base and removing the atoms of negative literals of the post-condition from the belief base. Let T be the function that takes a belief update action and a belief base, and returns the modified belief base if the pre-condition of the action is entailed by the agent's belief base. This function can be defined based on the specification of the belief update actions. A successful execution of a belief update action α is then defined as follows.

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

where $\sigma' \neq \bot$ and $\gamma' = \gamma - \{\gamma_i \mid \sigma' \models \gamma_i\}$. Since the parameters of a belief update action may be variables that are bound by the substitution θ, the substitution θ is applied to the belief update action before it is used to modify the belief base.

The execution of a belief update action can fail if its pre-condition is not entailed by the belief base or if it is not specified at all. In these cases, we assume that the function T returns

$\perp$. The consequence of failing to execute a belief update action is that the action remains in the plan base and an exception is generated to indicate the failure of this plan.

$$\frac{T(\alpha\theta, \sigma) = \perp \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \langle E, I \cup \{id\}, M\rangle\rangle} \quad (3)$$

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

Property 1 *The belief update actions preserve the consistency of the belief base. Moreover, goals will be removed from the goal base as soon as they are believed to be achieved. All other goals of an agent remain unachieved in the goal base.*

4.2.3 Test actions

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

Definition 4 Let φ and φ' be $\langle test\rangle$ expressions, ϕ be a $\langle belquery\rangle$ expression, ψ be a $\langle goalquery\rangle$ expression, and τ, τ_1 and τ_2 be substitutions. The entailment relation $\models_t$, which evaluates test expressions with respect to an agent's belief and goal bases (σ, γ), is defined as follows:

- $(\sigma, \gamma) \models_t \mathtt{B}(\phi)\tau \;\Leftrightarrow\; \sigma \models \phi\tau$
- $(\sigma, \gamma) \models_t \mathtt{G}(\psi)\tau \;\Leftrightarrow\; \gamma \models_g \psi\tau$
- $(\sigma, \gamma) \models_t (\varphi \;\&\; \varphi')\tau_1\tau_2 \;\Leftrightarrow\; (\sigma, \gamma) \models_t \varphi\tau_1$ and $(\sigma, \gamma) \models_t \varphi'\tau_1\tau_2$

Note that the test action $\mathtt{G}(\psi_1)$ `&` $\mathtt{G}(\psi_2)$ succeeds if the agent has either one single goal that entails (ψ_1 `and` ψ_2), or two different goals such that one entails ψ_1 and the second entails ψ_2. Note also that the test action $\mathtt{G}(\psi_1$ `and` $\psi_2)$ succeeds if the agent has one single goal that entails ψ_1 `and` ψ_2.

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

$$\frac{(\sigma, \gamma) \models_t \varphi\theta\tau \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\varphi, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{\}, \theta \cup \{\tau\}, \xi\rangle} \quad (4)$$

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

$$\frac{\neg\exists\tau : (\sigma, \gamma) \models_t \varphi\theta\tau \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\varphi, r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\varphi, r, id)\}, \theta, \langle E, I \cup \{id\}, M\rangle\rangle} \quad (5)$$

4.2.4 Goal dynamics actions

Goals can be adopted and added to the agent's goal base by means of basic actions `adopta(`ϕ`)` and `adoptz(`ϕ`)`. The first action adds the goal ϕ to the beginning of the goal base (recall that the goal base is a list) and the second action (`adoptz(`ϕ`)`) adds the goal ϕ to the end of the goal base. In the following transition rule, we use `adoptX(`ϕ`)` which can be either of these two actions and use `ground(`φ`)` to indicate that φ is ground.

$$\frac{\sigma \not\models \phi\theta \ \&\ \texttt{ground}(\phi\theta) \ \&\ \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{adoptX}(\phi), r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma', \{\}, \theta, \xi\rangle} \tag{6}$$

where $\gamma' = \phi\theta \bullet \gamma$ if `adoptX` is `adopta` (indicating that the goal $\phi\theta$ is added to the beginning of the list γ) and $\gamma' = \gamma \bullet \phi\theta$ if `adoptX` is `adoptz` (indicating that the goal $\phi\theta$ is added to the end of γ). The first condition of this transition rule ensures that the added goal is not already achieved by the agent.

However, if the agent believes that the goal to be adopted is already achieved (i.e., if the goal is entailed by the agent's belief base) or if the goal to be adopted is not ground, then the action is considered as failed. The action remains in the agent's plan base and an exception is generated.

$$\frac{(\sigma \models \phi\theta \ \vee\ \neg\texttt{ground}(\phi\theta)) \ \&\ \gamma \models_g \mathcal{G}(r)}{\begin{array}{l}\langle \iota, \sigma, \gamma, \{(\texttt{adoptX}(\phi), r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \\ \langle \iota, \sigma, \gamma, \{(\texttt{adoptX}(\phi), r, id)\}, \theta, \langle E, I \cup \{id\}, M\rangle\rangle\end{array}} \tag{7}$$

Goals can be dropped and removed from the goal base by means of `dropgoal(`ϕ`)`, `dropsubgoals(`ϕ`)`, and `dropsupergoals(`ϕ`)` actions. The first action removes from the goal base the goal ϕ, the second removes all goals that are subgoals of ϕ, and the third action removes all goals that have ϕ as a subgoal.

$$\frac{\gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{dropX}(\phi), r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma', \{\}, \theta, \xi\rangle} \tag{8}$$

where

- $\gamma' = \gamma - \{f \mid f \equiv \phi\theta \ \&\ \texttt{ground}(\phi\theta)\}$ if `dropX(`ϕ`)` is `dropgoal(`ϕ`)`
- $\gamma' = \gamma - \{f \mid \phi\theta \models f \ \&\ \texttt{ground}(\phi\theta)\}$ if `dropX(`ϕ`)` is `dropsubgoals(`ϕ`)`
- $\gamma' = \gamma - \{f \mid f \models \phi\theta \ \&\ \texttt{ground}(\phi\theta)\}$ if `dropX(`ϕ`)` is `dropsupergoals(`ϕ`)`

The three dropping goal actions will always succeed, even if there is no goal to be removed from the goal base.

Property 2 *If a goal is successfully adopted by an adopt goal action, then it is unachieved, i.e., it is not entailed by the belief base. The goal base remains ground after adopting and dropping goals. Dropping a goal, which is not ground, does not affect the goal base.*

4.2.5 Abstract action

Abstract actions are representations of plans.[3] The execution of an abstract action replaces the action with the plan it represents. The relation between an abstract action and the plan it represents is specified by means of a procedure call rule (PC-rule) that has a head unifiable with the abstract action. Abstract actions and the head of PC-rules are represented by

[3] Abstract actions can be used for different purposes. It can be used as a mechanism to reuse plans that occur in many other plans, or they can be used to implement recursion.

expressions of the form $\langle atom \rangle$. Let α be an abstract action, $\varphi <- \beta \mid \pi$ be a variant (i.e., all variables in $\varphi <- \beta \mid \pi$ are fresh) of a PC-rule of agent ι and $Unify$ be a function that returns the most general unifier of α and φ if they are unifiable, otherwise it returns $\bot$. A successful execution of an abstract action will replace it with a plan.

$$\frac{Unify(\alpha\theta, \varphi) = \tau_1 \;\&\; \sigma \models \beta\tau_1\tau_2 \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\pi\tau_1\tau_2\ , r, id)\}, \theta, \xi \rangle} \tag{9}$$

Since the abstract action may contain variables that are bound by the substitution θ, we apply θ to it before unifying it with the head of the PC-rule. Note that the resulting substitution τ_1 is applied to the belief condition of the PC-rule before testing it against the agent's belief base. This test may result in a second substitution τ_2 which together with τ_1 are applied to the plan of the PC-rule before replacing the abstract action with the plan. The applications of substitutions to the belief condition and the plan of the PC-rule ensures that the value of the shared variables are passed from the head of the rule to the belief condition, and subsequently to the plan of the rule.

However, if there is no PC-rule applicable (i.e., a PC-rule is not applicable if either its head cannot be unified with the abstract action or its belief condition is not entailed by the belief base), then the execution of the abstract action is considered as failed. As a consequence, the abstract action remains in the agent's plan base and an exception is generated. Let α be an abstract action and PC be the set of PC-rules of agent ι.

$$\frac{\forall r' \in PC : (\ Unify(\alpha\theta,\ \mathcal{G}(r')\ ) = \bot \ \vee\ \sigma \not\models \mathcal{B}(r')\ ) \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \langle E, I, M \rangle \rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \langle E, I \cup \{id\}, M \rangle \rangle} \tag{10}$$

4.2.6 Communication action

Agents can communicate with each other by sending messages to each other. An agent can send a message to another agent by means of the $\texttt{send}(j, p, l, o, \phi)$ action. The execution of the send action broadcasts a message which will be added to the event base of the receiving agent. The broadcasted message will include the name of the sending agent.

$$\frac{\varphi = \langle \iota, j, p, l, o, \phi \rangle \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{send}(j, p, l, o, \phi), r, id)\}, \theta, \xi \rangle \xrightarrow{\varphi\theta!} \langle \iota, \sigma, \gamma, \{\}, \theta, \xi \rangle} \tag{11}$$

where $\varphi\theta = \langle \iota, j\theta, p\theta, l\theta, o\theta, \phi\theta \rangle$. The broadcasting of the message is indicated by the exclamation mark in $\varphi\theta!$ which means that the transition proceeds by broadcasting the message event. Note that the substitution θ is applied to the broadcasted message because the arguments of the send action can be variables bound by the substitution. The send action will always succeed.

A 2APL agent is assumed to be able to receive a message that is sent to it at any time. The received message is added to the event base of the agent.

$$\frac{\varphi = \langle j, \iota, p, l, o, \phi \rangle \;\&\; \psi = \texttt{message}(j, p, l, o, \phi)}{\langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I, M \rangle \rangle \xrightarrow{\varphi?} \langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I, M \cup \{\psi\} \rangle \rangle} \tag{12}$$

The reception of a broadcasted message is indicated by the question mark in $\varphi?$ which means that the transition cannot proceed unless a message φ is received. The received message is used to create a message event added to the agent's event base. Note that an agent can always receive a broadcasted messages because this transition rule has no condition that fail and thus can be applied at each moment in time. We still need to specify that the

broadcasted messages by a sending agent are captured and passed to the receiving agents. This is, however, a transition at the multi-agent level and will be presented in Sect. 4.5.

4.2.7 External action

The execution of an external action by an individual agent affects the external environments that may be shared by other agents. An environment returns in turn a value back to the agent, which can be either some result of the successfully performed action (in the form of a list of terms) or $\perp$ indicating that the execution of the action is failed.[4] We assume that there is no time between performing an action in an environment and the reception of the return value, i.e., no other external action can be executed between performing an external action and the reception of its corresponding return value.

Although this assumption implies that external actions are blocking actions (i.e., block an agent's plan execution), it does not mean that the execution of an agent's plan should wait until the actual and intended effect of the action is realized. If it is desirable to continue the execution of plans and not to wait for the realization of the actual effect of an external action, a programmer may implement the environment in such a way that it starts a new thread to realize the actual effect of the action, while immediately gives the control back to the agent (the returned value can be an empty list). Later on the environment can return values (e.g., related to the actual effect of the action) back to the agent by means of events.

In order to manage the exchange of information between an individual agent and an environment, and to make sure that no other external action can be executed between performing an external action and the reception of the return value, the transition for external actions broadcasts an event $env(\iota, \alpha(t_1\theta, \ldots, t_n\theta), t)$ to the multi-agent system level (indicating that agent ι performs action $\alpha(t_1\theta, \ldots, t_n\theta)$ in environment env) and waits for the return value t from environment env in one transition step. In Sect. 4.5, we will introduce a rule that synchronizes the transition of external actions and the transition of the environments caused by the external actions. A successful execution of an external action will receive a return value which is not falsum. The returned value is assigned to the return variable V.

$$\frac{t \neq \perp \;\&\; \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(@env(\alpha(t_1, \ldots, t_n), V), r, id)\}, \theta, \xi \rangle \xrightarrow{env(\iota, \alpha(t_1\theta, \ldots, t_n\theta), t)} \langle \iota, \sigma, \gamma, \{\}, \theta \cup \{V/t\}, \xi \rangle} \tag{13}$$

If the return value is a failure exception, then the execution of the external action is considered as failed. As a consequence, the action is not removed from the plan base and its identifier is added to the agent's event base.

$$\frac{t = \perp \;\&\; \gamma \models_g \mathcal{G}(r)}{\begin{array}{r}\langle \iota, \sigma, \gamma, \{(@env(\alpha(t_1, \ldots, t_n), V), r, id)\}, \theta, \langle E, I, M \rangle \rangle \xrightarrow{env(\iota, \alpha(t_1\theta, \ldots, t_n\theta), t)} \\ \langle \iota, \sigma, \gamma, \{(@env(\alpha(t_1, \ldots, t_n), V), r, id)\}, \theta, \langle E, I \cup \{id\}, M \rangle \rangle\end{array}} \tag{14}$$

4.3 Transition rules for plans

In this section, we present the transition rules that capture the execution of plans consisting of basic actions composed by sequence, conditional choice, conditional iteration, and non-interleaving operators.

[4] This is the case when the Java environment throws the exception `ExternalActionFailedException`.

4.3.1 Sequence plan

The execution of a sequence plan $\alpha; \pi$ consists of the execution of the basic action α followed by the execution of plan π. Thus, an agent with a sequence plan $\alpha; \pi$ can make a transition through which the first basic action α is executed. The rest π of the plan remains in the resulting configuration.

$$\frac{\langle \iota, \sigma, \gamma, \{(\alpha, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma', \gamma', \{\}, \theta', \xi'\rangle}{\langle \iota, \sigma, \gamma, \{(\alpha; \pi, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma', \gamma', \{(\pi, r, id)\}, \theta', \xi'\rangle} \tag{15}$$

4.3.2 Conditional plan

The execution of a conditional plan $\texttt{if}\ \varphi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2$ consists of a choice between plans π_1 and π_2. In particular, the condition φ is evaluated with respect to the agent's belief and goal bases. If the agent's belief and goal bases entails the condition, then plan π_1 will be selected, otherwise plan π_2 is selected. If the conditional plan has no 'else' part, the 'else' part is considered as a $\texttt{skip}$ plan.

$$\frac{(\sigma, \gamma) \models_t \varphi\theta\tau\ \&\ \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{if}\ \varphi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\pi_1\tau, r, id)\}, \theta, \xi\rangle} \tag{16}$$

$$\frac{\neg\exists\tau : (\sigma, \gamma) \models_t \varphi\theta\tau\ \&\ \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{if}\ \varphi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\pi_2, r, id)\}, \theta, \xi\rangle} \tag{17}$$

In the first transition rule, the test of the condition $\varphi\theta$ against the agent's belief and goal bases may result in an additional substitution τ (in the case $\varphi\theta$ contains unbound variables). This substitution will only be applied to plan π_1 (and not added to the general substitution θ) as the scope of the condition φ is the if-part of the conditional plan. In the second transition rule, there will be no additional substitution. Note that the execution of conditional plans will always succeed.

4.3.3 While plan

The execution of a while plan $\texttt{while}\ \varphi\ \texttt{do}\ \pi$ depends on its condition φ. In particular, if φ is entailed by the agent's belief and goal bases, then the plan π should be executed after which the while plan should be tried again. However, if the condition φ is not entailed by the agent's belief and goal bases, then the while plan should be removed from the plan base entirely.

$$\frac{(\sigma, \gamma) \models_t \varphi\theta\tau\ \&\ \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{while}\ \varphi\ \texttt{do}\ \pi, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\pi\tau; \texttt{while}\ \varphi\ \texttt{do}\ \pi, r, id)\}, \theta, \xi\rangle} \tag{18}$$

$$\frac{\neg\exists\tau : (\sigma, \gamma) \models_t \varphi\theta\tau\ \&\ \gamma \models_g \mathcal{G}(r)}{\langle \iota, \sigma, \gamma, \{(\texttt{While}\ \varphi\ \texttt{do}\ \pi, r, id)\}, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{\}, \theta, \xi\rangle} \tag{19}$$

In the first transition rule where the condition φ is entailed by the agent's belief and goal bases, the substitution τ resulted from testing the condition φ against the agent's belief and goal bases will be applied to plan π. This substitution will not be added to the general substitution θ as its scope is limited to the body of the while plan. The execution of a while loop always succeed.

4.3.4 Atomic plan

The execution of an atomic plan is the non-interleaved execution of the maximum number of actions of the plan. Let $[\alpha_1; \ldots; \alpha_n]$ be an atomic plan. We need to define a transition rule that allows the derivation of a transition from a configuration $A_1 = \langle \iota, \sigma_1, \gamma_1, \{([\alpha_1; \ldots; \alpha_n], r, id)\}, \theta_1, \xi_1 \rangle$ to a configuration $A_m = \langle \iota, \sigma_m, \gamma_m, \Pi, \theta_m, \xi_m \rangle$ such that either $\Pi = \{([\alpha_k; \ldots; \alpha_n], r, id)\}$ and α_k is the first action whose execution fails, or $\Pi = \{\}$, i.e., all actions of the atomic plan are successfully executed. Let $A_i = \langle \iota, \sigma_i, \gamma_i, \{[(\pi, r, id)]\}, \theta_i, \xi_i \rangle$ and $A_{i+1} = \langle \iota, \sigma_{i+1}, \gamma_{i+1}, \{[(\pi', r, id)]\}, \theta_{i+1}, \xi_{i+1} \rangle$. In order to specify the transition rule for atomic plans, we define $transition(A_i, A_{i+1})$ to indicate that the following one-step transition is derivable (the execution of one step of plan π results in plan π'):[5]

$A_i = \langle \iota, \sigma_i, \gamma_i, \{(\alpha; \pi, r, id)\}, \theta_i, \xi_i \rangle \longrightarrow \langle \iota, \sigma_{i+1}, \gamma_{i+1}, \{(\pi', r, id)\}, \theta_{i+1}, \xi_{i+1} \rangle = A_{i+1}$

The following transition rule specifies the execution of atomic plan $[\alpha_1; \ldots; \alpha_n]$.

$$\frac{(\forall_i : 1 \leq i \leq m \rightarrow transition(A_i, A_{i+1})) \;\&\; \forall A : \neg transition(A_{m+1}, A)}{\langle \iota, \sigma_1, \gamma_1, \{([\alpha_1; \ldots; \alpha_n], r, id)\}, \theta_1, \xi_1 \rangle \longrightarrow \langle \iota, \sigma_{m+1}, \gamma_{m+1}, \Pi, \theta_{m+1}, \xi_{m+1} \rangle} \quad (20)$$

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

Property 3 *The event base of the last configuration A_{m+1} resulted from performing atomic plan $[\alpha_1; \ldots; \alpha_n]$ contains an exception if action α_m of the atomic plan was executed and failed. The failed action α_m and all its subsequent actions will remain in the plan base.*

4.3.5 Multiple concurrent plans

An agent executes its plans concurrently by interleaving the executions of their constituent actions. This idea is captured by the following transition rule which states that an agent executes one of its plans at each computation step.

$$\frac{\langle \iota, \sigma, \gamma, \rho, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma', \gamma', \rho', \theta', \xi' \rangle}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma', \gamma', \Pi', \theta', \xi' \rangle} \quad (21)$$

where $\rho = \{(\pi, r, id)\} \subset \Pi$ and $\Pi' = (\Pi \setminus \rho) \cup \rho'$. Note that the condition $\rho \subset \Pi$ ensures that this transition is used only if the agent's plan base contains more than one plan. In the case that the agent's plan base contains one single plan, then the transition rules 1 to 20, where the plan base is required to be a singleton set, are used.

4.4 Practical reasoning rules

In this section, we present the transition rules that captures the application of the three types of practical reasoning rules.

4.4.1 Planning goal rules

A 2APL agent generates plans by applying PG-rules of the form κ<-$\beta \mid \pi$. There are two types of PG-rules. The first type of PG-rule, characterized by $\kappa \neq \texttt{true}$, indicates that

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

plan π should be generated to achieve goal κ. The second type of PG-rule, characterized by $\kappa = \texttt{true}$, indicates that plan π should be generated when the agent believes β. However, as goals and beliefs may persist through time, we need to specify when the rules can be re-applied in order to avoid generating multiple (identical) plans for the same beliefs and goals.

In our opinion, a PG-rule of the first type should be re-applied for a specific goal only if the generated plan for that specific goal is fully executed and removed from the plan base. In order to illustrate this idea suppose an agent with goals `clean(blockworld1)` and `clean(blockworld2)`, and the following PG-rule:

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

If this rule is applied for the goal `clean(blockworld1)`, then it can be re-applied for `clean(blockworld1)` only if the generated plan for this goal is executed and removed from the plan base. Note that this PG-rule can be re-applied to goal `clean(blockworld2)` even if the plan for the goal `clean(blockworld1)` is in the plan base. A PG-rule of the second type (i.e., a PG-rule with `true` as the head) can be re-applied only if the plan generated by this rule is executed and removed from the plan base, regardless of the specific instantiation of its belief condition. So, if the PG-rule `true <- trash(X,Y) and pos(X,Y) | RemoveTrash()` is applied because the agent believes `trash(2,3)` and `pos(2,3)`, then it cannot be re-applied if the agent also believes `trash(5,1)` and `pos(5,1)`.

In general, an agent can apply one of its PG-rules $r = \kappa \texttt{<-} \beta \mid \pi$, if κ is entailed by one of the agent's goals, β is entailed by the agent's belief base, and there is no plan in the plan base that has been generated (and perhaps partially executed) by applying the same PG-rule to achieve the same goal. Let P be the set of all possible plans, id be a new unique plan identifier, and $r' = \kappa' \texttt{<-} \beta' \mid \pi'$ be a variant of r.[6] Applying the PG-rule r' will add an instantiation of the plan π' to the agent's plan base.

$$\frac{\gamma \models_g \kappa'\tau_1 \;\&\; \sigma \models \beta'\tau_1\tau_2 \;\&\; \neg\exists \pi^* \in P : (\pi^*,\ (\kappa'\tau_1 \texttt{<-} \beta \mid \pi)\ , id')\in \Pi}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi\rangle \longrightarrow \langle \iota, \sigma, \gamma, \Pi \cup \{(\pi'\tau_1\tau_2\ ,\ (\kappa'\tau_1 \texttt{<-} \beta \mid \pi),\ id)\}, \theta, \xi\rangle} \tag{22}$$

Note that $(\kappa'\tau_1 \texttt{<-} \beta \mid \pi)$ is the same as the applied PG-rule r except that its head is instantiated (its guard and body are not instantiated and are the same as r). This allows the application of one PG-rule for different instantiations of goals, but blocks the re-application of PG-rules with the head `true`. The existential quantification in the condition of the transition rule guarantees that there is no plan π^* (which may be either the same as $\pi'\tau_1\tau_2$ or a remainder/executed part of it) which is already generated for the goal $\kappa'\tau_1$. Note that once the plan entry is added to the plan base, the PG-rule r cannot be re-applied to achieve the goal $\kappa'\tau_1$ until the plan $\pi'\tau_1\tau_2$ is fully executed and removed from the plan base. Note also that a PG-rule with the head `true` cannot be re-applied if it is already applied and the generated plan entry is still in the plan base.

4.4.2 Procedure call rules

In Sect. 4.2.5, we presented the transition rule for the execution of abstract actions. This transition rule was based on application of PC-rules that generate plans to be replaced for the executed abstract actions. In this subsection, we present the transition rules in which PC-rules are applied for other purposes, i.e., to generate plans in order to react to either the received

[6] The reason to require a variant of r is that the generated plan instantiation can contain variables (as it may include test actions, conditional choice and iteration statements) that may later be instantiated by θ.

events from the environments or the received messages from other agents. As we will see in Sect. 4.5, events are broadcasted when the state of the environments change. These events are catched and included in the E component of the event base. The messages received from other agents are included in the M component of the event base.

Let $\texttt{event}(\phi, env) \in E$ be an event broadcasted by environment env, $\texttt{message}(s, p, l, o, e) \in M$ be a message broadcasted by agent s, and $Unify$ be a function that returns the most general unifier of two atomic formulas (or returns $\perp$ if there is no unification possible). Let $\varphi \texttt{<-} \beta \mid \pi$ be a variant of a PC-rule of agent ι and $\xi = \langle E, I, M \rangle$ be the event base of agent ι. The transition rule for applying PC-rules to the events from E or M is defined as follows:

$$\frac{\psi \in E \cup M \;\&\; Unify(\psi, \varphi) = \tau_1 \;\&\; \sigma \models \beta\tau_1\tau_2}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma, \gamma, \Pi', \theta, \xi' \rangle} \tag{23}$$

where id is a new unique plan identifier, $r = (\texttt{true<-}\beta \mid \pi)$, $\Pi' = \Pi \cup \{(\pi\tau_1\tau_2, r, id)\}$, and $\xi' = \langle E \setminus \{\psi\}, I, M \rangle$ if $\psi = \texttt{event}(\phi, env)$ or $\xi' = \langle E, I, M \setminus \{\psi\} \rangle$ if $\psi = \texttt{message}(s, p, l, o, e)$. Note in the generated plan entry that `true` is used as the head of the rule that has been applied to generate plan $\pi\tau_1\tau_2$. This is because plans can only be executed if their associated goals (the purpose for their generation) are entailed by the agent's goal base. As an event/message is different from a goal and is not entailed by the agent's goal base, we use `true` which is always entailed by the agent's goal base (see Sect. 4.2).

However, if there is no PC-rule the head of which is unifiable with an event or message from the agent's event base (E or M), then the event or message will be removed from E and M, respectively.

$$\frac{\psi \in E \cup M \;\&\; \forall r \in PC : Unify(\psi, \mathcal{G}(r)) = \perp}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle \longrightarrow \langle \iota, \sigma, \gamma, \Pi, \theta, \xi' \rangle} \tag{24}$$

where PC is the set of PC-rules of agent ι, $\xi' = \langle E \setminus \{\psi\}, I, M \rangle$ if $\psi = \texttt{event}(\phi, env)$ or $\xi' = \langle E, I, M \setminus \{\psi\} \rangle$ if $\psi = \texttt{message}(s, p, l, o, e)$.

Property 4 *An event ψ for which there exists a rule r such that $Unify(\psi, \mathcal{G}(r)) \neq \perp$ remains in the event base until it is processed.*

4.4.3 Plan repair rules

A plan repair rule can be applied to replace a plan if an exception is received indicating that the execution of the first action of the plan is failed. Suppose the execution (of the first action) of a plan π of an agent fails. Then, a plan repair rule $\pi_1 \texttt{<-} \beta \mid \pi_2$ can be applied to repair π if the abstract plan expression π_1 matches π and β is entailed by the agent's belief base. The result of the match will be used to instantiate the abstract plan expression π_2 and to generate a new plan that replaces the failed plan.

We assume a plan unification operator $PlanUnify(\pi, \pi_1)$ that implements a prefix matching strategy for unifying plan π with abstract plan expression π_1. Roughly speaking, a prefix matching strategy means that the abstract plan expression is matched with the prefix of the failed plan. The plan unification operator returns a tuple (τ_d, τ_p, π^*) where τ_d is a substitution that binds domain variables, τ_p is a substitution that binds plan variables, and π^* is the postfix of π that did not play a role in the match with π_1 (e.g., $PlanUnify(\alpha(a); \alpha(b); \alpha(c),\ X; \alpha(Y)) = ([Y/b], [X/\alpha(a)], \alpha(c))$). This result is used

to generate a new plan by applying both substitutions τ_d and τ_p to the abstract plan expression π_2 in the body of the applied PR-rule followed by the rest plan π^*.

Let $\pi_1 <\!\!- \beta \mid \pi_2$ be a variant of a PR-rule of agent ι. The following transition rule specifies the application of a PR-rule to a failed plan.

$$\frac{\mathit{Plan\ Unify}(\pi, \pi_1) = (\tau_d, \tau_p, \pi^*)\ \&\ \sigma \models \beta\tau_d\tau\ \&\ id \in I}{\langle \iota, \sigma, \gamma, \{(\pi, r, id)\}, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \{(\pi_2\tau_d\tau\tau_p; \pi^*, r, id)\}, \theta, \langle E, I \setminus \{id\}, M\rangle\rangle} \tag{25}$$

Note that substitutions τ_d, τ_p (resulted by the plan unification operator) and τ are applied to abstract plan expression π_2. The resulted instantiated plan $\pi_2\tau_d\tau\tau_p$ followed by the unmatched rest plan π^* form a new plan that replaces the failed plan.

Let PR be the set of plan repair rules of agent ι. If there is no rule in PR applicable to repair a failed plan (i.e., the head of the rule cannot match with the failed plan or its belief condition is not entailed by the belief base), then the exception is deleted from the event base and the failed plan remains in the plan base.

$$\frac{id \in I\ \&\ (\pi, r, id) \in \Pi\ \&\ \forall(\pi_1 <\!\!- \beta \mid \pi_2) \in PR : (PlanUnify(\pi, \pi_1) = \bot \text{ or } \sigma \not\models \beta)}{\langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I, M\rangle\rangle \longrightarrow \langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I \setminus \{id\}, M\rangle\rangle} \tag{26}$$

Plan repair rules should be used cautiously. If the repaired part of a failed plan includes an action through which variables are bound (e.g., test or external actions), then the variables should not occur in the unrepaired part of the plan (i.e., plan part bound to plan variables or the plan part π^*).

4.5 Multi-agent transition rules

The execution of a 2APL multi-agent system is the interleaved executions of the involved individual agents and the environments. Moreover, we assume that the external shared environments can change either by the execution of an agent's external action in one of the environments or by the internal dynamics of the environments (e.g., clock changes every second and resources can grow or shrink). Therefore, the configuration of a multi-agent system can be modified when either the configuration of one of the involved individual agents is modified or when the shared environments change.

4.5.1 Interleaved executions of individual agents

The following transition rule indicates that if an individual agent makes a transition step by executing a basic action (except external and communication actions), then the multi-agent system that involves the agent can make a transition too. As explained in previous sections, the transition steps made by individual agents are through either the execution of basic actions, plans, or the application of practical reasoning rules.

$$\frac{A_i \rightarrow A_i'}{\langle A_1, \ldots, A_i, \ldots, A_n, \chi\rangle \rightarrow \langle A_1, \ldots, A_i', \ldots, A_n, \chi\rangle} \tag{27}$$

4.5.2 Execution of environment by external actions

As explained in Sect. 4.2.7, the execution of external actions will affect the shared external environments. In particular, the execution of an external action $@env(\alpha(t_1, \ldots, t_n), V)$ has

two different effects: (1) the state of the shared environments may be changed, and (2) the variable V binds to a term. We assume that the effect of an external action on environments is determined by the designer of the environments. We capture the effect of performing an external action by an agent in an environment through a function that takes these ingredients and returns a tuple consisting of the new state of the environments[7] and a return value. The return value can be a list, which is used to bind variable V, or it can be $\perp$ indicating that the execution of the action is failed, or the agent has no access to the environment (this information is taken from the `.mas` file), or the action is not defined in the environment. Let $F^{env}_{\iota,\alpha}(t_1, \ldots, t_n, \chi)$ be the function that determines the effects of external action α with arguments $t_1, \ldots, t_n$ performed by agent ι in the environment $env \in \chi$ and returns a tuple (t, χ'), where t is either a list of terms that binds output variable V or $\perp$, and χ' is the updated set of environments.

$$\frac{F^{env}_{\iota,\alpha}(t_1, \ldots, t_n, \chi) = (t, \chi')}{\chi \xrightarrow{env(\iota,\alpha(t_1,\ldots,t_n),t)} \chi'} \tag{28}$$

This transition rule indicates that when an external action is broadcasted, the environment can make a transition according to the determined effect of the action. Note that an external action is broadcasted as a consequence of the execution of the external action by an individual agent (see transition rules 13 and 14).

In order to coordinate the execution of external actions and their corresponding effects on the shared external environments, we need to synchronize the transition of individual agent configuration, caused by executing an external action, and the transition of the shared external environments, caused by the corresponding effect of the external action.

$$\frac{A_i \xrightarrow{env(\iota,\alpha(t_1,\ldots,t_n),t)} A'_i \;\&\; \chi \xrightarrow{env(\iota,\alpha(t_1,\ldots,t_n),t)} \chi'}{\langle A_1, \ldots, A_i, \ldots, A_n, \chi\rangle \longrightarrow \langle A_1, \ldots, A'_i, \ldots, A_n, \chi'\rangle} \tag{29}$$

This transition rule indicates that if an individual agent executes an external action and the environments changes according to the effect of the external action simultaneously, then the multi-agent system can make a transition through which only the agent that has executed the external action and the environments are changed.

4.5.3 Execution of environment by internal dynamics

A multi-agent configuration can be modified if a change occurs in one of its environments. As noted we assume that environments are dynamic and can be changed independent of any agent action. A change in an environment can broadcast an event which can be used by agents to generate and execute plans (using PC-rules, see Sect. 4.4.2). A change in an environment can thus cause a multi-agent transition through which the configuration of the individual agents can be modified. Let $env \xrightarrow{(\phi,\{k,\ldots,l\})!} env'$ indicate that environment env is changed to env' through which event $(\phi, \{k, \ldots, l\})$ is broadcasted. This event indicates that the information represented by atom ϕ is generated for agents $\{k, \ldots, l\}$. The following transition defines the multi-agent execution step through which (1) the environments $env \in \chi$ is updated resulting in env', and (2) the event ϕ is added to the event base of agents $\{k, \ldots, l\}$.

[7] We assume that a change in one environment $env \in \chi$ may cause changes in other environments in χ as well.

$$\frac{env \in \chi \;\&\; env \overset{(\phi,\{k,\ldots,l\})!}{\Longrightarrow} env'}{\langle A_1, \ldots, A_n, \chi\rangle \longrightarrow \langle A'_1, \ldots, A'_n, \chi'\rangle} \tag{30}$$

where

$\chi' = (\chi \setminus \{env\}) \cup \{env'\}$
$A_\iota = \langle \iota, \sigma, \gamma, \Pi, \theta, \langle E, I, M\rangle\rangle$
$A'_\iota = \langle \iota, \sigma, \gamma, \Pi, \theta, \langle E \cup \{\texttt{event}(\phi, env)\}, I, M\rangle\rangle$ if $\iota \in \{k, \ldots, l\}$
$A'_\iota = A_\iota$ otherwise.

It is important to note that an event is directed to a subset of individual agents. Which agents receives an event is the decision of the environment designer. Also, we will not further explain the transitions of external environments denoted by the transition relation $\Longrightarrow$ and assume that these transitions are determined by the implementation of the environments.

4.5.4 Synchronous communication

In Sect. 4.2.6, transition rules 11 and 12 are presented that indicate (1) an agent can send a message to another agent, and (2) an agent can receive a message. In order to ensure that the sent message is received by the addressee, we present a transition rule at multi-agent level which indicates that the sending and receiving agents exchange the message synchronously. Let $\varphi = \langle i, j, p, l, o, \phi\rangle$ be a broadcasted massage. The transition rule for synchronized communication is defined as follows:

$$\frac{A_i \overset{\varphi!}{\longrightarrow} A'_i \;\&\; A_j \overset{\varphi?}{\longrightarrow} A'_j}{\langle A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n, \chi\rangle \longrightarrow \langle A_1, \ldots, A'_i, \ldots, A'_j, \ldots, A_n, \chi\rangle} \tag{31}$$

This transition rule indicates that if one agent can make a transition by sending a message and another agent can make a transition by receiving the message, then the multi-agent system can make a transition through which the message is exchange between the sending and receiving agents synchronously.

5 Executing multi-agent systems

The execution of a 2APL multi-agent system is determined by the 2APL transition system that is specified by the transition rules presented in Sect. 4. It consists of a set of so called computation runs.

Definition 5 (*computation run*) Given a transition system and an initial configuration s_0, a computation run $\texttt{CR}(s_0)$ is a finite or infinite sequence $s_0, \ldots, s_n$ or $s_0, \ldots$ where s_i is a configuration, and $\forall_{i>0}: s_{i-1} \rightarrow s_i$ is a transition in the transition system.

We can now use the concept of a computation run to define the execution of 2APL multi-agent systems.

Definition 6 (*execution of 2APL multi-agent systems*) The execution of a 2APL multi-agent system with initial configuration $\langle A_1, \ldots, A_n, \chi\rangle$ is the set of computation runs $\texttt{CR}(\langle A_1, \ldots, A_n, \chi\rangle)$ of the 2APL transition system.

Note that the computation runs of a 2APL multi-agent system consist of multi-agent system transitions which can be derived by means multi-agent system transition rules. The multi-agent system transition rules are in turn defined in terms of transitions of individual agents and the environments, which can be derived by means of transition rules for individual agents and the environments.

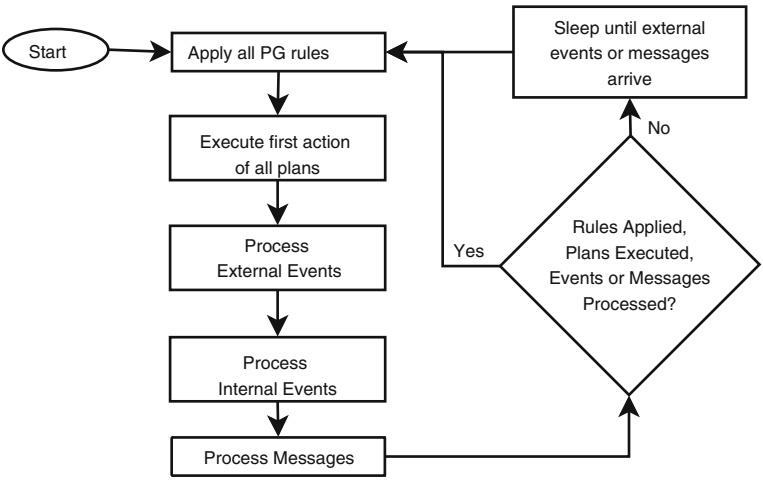

Fig. 3 The deliberation cycle for individual 2APL agents

The execution of a 2APL multi-agent system is thus the set of all possible computational runs. An interpreter that executes a 2APL multi-agent system is one of these computational runs. In particular, the interpreter of 2APL executes individual agents and the environment in parallel in an interleaving mode. Moreover, the execution of each individual agent is determined by the so-called deliberation process, which is specified as a cyclic process explained in the next subsection.

5.1 2APL deliberation process

In order to execute an individual agent, the 2APL interpreter follows a certain order of deliberation steps repeatedly and indefinitely. Each deliberation step is specified by a set of transition rules indicating which transitions an individual agent can make at each moment in time. This repeated and indefinite sequence of the deliberation steps can be generated by the agent's deliberation cycle. The 2APL deliberation cycle is illustrated in Fig. 3.

Each cycle starts by applying all applicable PG-rules, each rule only one time. Note that a PG-rule can be applied more than one time 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 $g(X) \texttt{<-} b(Y) \mid \pi(X, Y)$ 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 $g(a)$[8] in one cycle and to the goal $g(b)$ `and` $g'(c)$ in the next cycle.

The deliberation cycle proceeds by executing only the first actions of all plans. 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 environment, all internal events indicating the failure of plans, and all messages received from other agents, respectively. An event from external environment is processed by applying the first applicable PC-rule (with the head `event`) to it. An internal event, 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 generates and add plans to the corresponding agent's plan base.

[8] Note that the goal base is a list of goals.

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.

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 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 Fig. 3 because it has some interesting properties. Some of these properties can be formulated as follows:

Property 5 *If the execution of a plan fails, then the plan will either be repaired in the same deliberation cycle or get re-executed in the next deliberation cycle.*

This is the direct consequence of the fact that in each deliberation cycle plans are executed before internal events are processed and that internal events are removed if there is no plan repair rules to modify the failed plans (transition rule 26). It should be noted that a plan fails if the execution of its first action fails. The failed action remains as the first action of the plan. See the transition rules 3, 5, 7, 10, and 14 of failed actions.

Property 6 *If the first action of a failed plan is a belief update action, a test action, an adopt goal action, an abstract action, or an external action, and there is no plan repair rule to repair it, then the failed plan may be successfully executed in the next deliberation cycle.*

The reason is that the belief and goal bases can be modified in one deliberation cycle such that belief and goal test actions can executed successfully in the next deliberation cycle. Also, the state of the environment can change such that failed external actions can be executed successfully in the next deliberation steps.

6 2APL and related works

Existing BDI-based agent-oriented programming languages such as Jason [5], Jack [30], Jadex [20], 3APL [14], KGP [17,6,24], Minerva [18], (Concurrent) MetateM [11,12] and the family of Golog languages [13,25] are designed to implement agents in terms of BDI concepts. Except Jack and Jadex, which are based on Java, these agent-oriented programming languages enjoy explicit formal semantics. Among the programming languages with formal semantics, KGP and Minerva are mainly based on logic programming and exploit related techniques. Other languages are based on different theories and techniques such as temporal logic (e.g., (Concurrent) MetateM), situation calculus (e.g., Golog family of programming languages), or a mixture of logic programming and rule based systems (e.g., Jason and 3APL). In this section, we provide a general comparison between 2APL and these BDI-based agent-oriented programming languages. Although some of these programming languages such as Jack, Jadex, Jason, and 3APL come with a development environment, the following comparison is based on the language part only, ignoring their similarities and differences with respect to available development tools.

2APL extends and modifies the original version of 3APL [14] in many different ways. While the original version of 3APL is basically a programming language for single agents, 2APL is designed to implement multi-agent systems. Moreover, a 3APL agent state consists of beliefs and plans, where plans consist of belief update, test, and abstract actions composed

by the sequence and choice operator. The original version of 3APL provides only plan revision rules that can be applied to revise an agent's plan. 2APL includes these programming constructs and adds new ones to implement a set of agents, a set of external environments, the access relation between agents and environments, events, goals, and a variety of action types such as external actions, goal related actions, and communication action. Moreover, two additional rule types are introduced to implement an agent's reactive and proactive behavior. The plan repair rules of 2APL have similar syntax as the plan revision rules of 3APL. However, while 3APL rules can be applied to revise any arbitrary plan (also plans that are not failed), 2APL plan repair rules are applied only to repair *failed* plans. In our view, it does not make sense to modify a plan if the plan is executable. Finally, 2APL proposes a new plan constructs to implement a non-interleaving execution of plans. The execution of 2APL and 3APL programs is based on a cyclic process. In 3APL, at each cycle the agent's plans are executed and revised, while in 2APL at each cycle goals, (internal and external) events, and messages are processed as well.

Jack [30] and Jadex [20] are Java-based agent-oriented programming languages. They extend Java with programming constructs to implement BDI concepts such as beliefs, goals, plans, and events. In both Jack and Jadex a number of syntactic constructs are added to Java to allow programmers to declare beliefsets, 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.

Although Jack, Jadex and 2APL share the basic BDI concepts, they give different semantics[9] to the BDI concepts and realize different rationality principles for these concepts. For example, beliefs and goals in Jack and Jadex have no logical semantics such that the agents cannot reason about their beliefs and goals. A consequence of this is that a Jack or Jadex agent is not able to generate plans that can contribute to the achievement of its goals, but not necessarily achieve them. The goals in 2APL are logical formula such that an agent can generate a plan if it can reason that the plan contributes to the achievement of one of its goals, i.e., if the goal that can be achieved by the plan is logically entailed by one of the agent's goals. Moreover, the consistency of an agent's state in Jack and Jadex, as far as they are defined, is left to agent programmers, i.e., the agent programmer is responsible to make sure that state updates preserve the state consistency. Also, there is no logical relation between beliefs and goals in Jack and Jadex. In these programming languages, a goal or event are used to generate a plan, while beliefs and goals in 2APL refer to the same set of states. In 2APL, a goal state represented by a logical formula is thought to be achieved when the formula is entailed by the beliefs. A similarity between 2APL and Jadex is that they both provide a programming construct to implement non-interleaving execution of plans.

Jason [5] is introduced as an interpreter of an extension of AgentSpeak, which is originally proposed by Rao [21]. Like 2APL, Jason distinguishes multi-agent system concerns from individual agent concerns, though it does not allow the specification of agents' access relation to external environments. An individual agent in Jason is characterized by its beliefs, plans and the events that are either received from the environment or generated internally. A plan in Jason is designed for a specific event and belief context. The execution of individual agents in Jason is controlled by means of a fixed (not programmable) cycle of operations encoded in its semantics. In each cycle, events from the environment are processed, an event is selected, a plan is generated for the selected event and added to the intention base, and

[9] The semantics of Jack and Jadex are not explicitly presented, but explained informally.

finally a plan is selected from the intention base and executed (through which new internal events can be generated).

A plan in Jason is similar to the so-called procedure call rules in 2APL. Such a rule indicates that a plan should be generated by an agent if an event is received/generated and the agent has certain beliefs. Like 2APL, Jason is based on first-order representation for beliefs, events, and plans. In contrast to 2APL, Jason has no explicit programming construct to implement declarative goals, though goals can be indirectly simulated by means of a pattern of plans. The beliefs of a Jason agent is a set of literals with strong negation, while in 2APL an agent's beliefs is a set of Horn clauses. Moreover, the beliefs and plans in Jason can be annotated with additional information that can be used in belief queries and plan selection process, while such information can be encoded in the beliefs and plans in 2APL. Finally, 2APL has two additional rule types that generates plans to achieve declarative goals and repair plans when their executions fail. In Jason, plan failure can be modeled by means of plans that react to the so-called deletion events. The comparison between 3APL and AgentSpeak provided in [15] may be useful for comparing 2APL and Jason.

Due to the computational complexity of planning, the family of Golog languages [13,25] propose high-level program execution as an alternative to control agents and robots that operate in dynamic and incompletely known environments. In fact, the high-level (agent) program consists of a set of actions, including the sense action (in IndiGolog [25]), composed by means of conditionals, iteration, recursion, concurrency and non-deterministic operators. Instead of finding a sequence of actions to achieve a desired state from an initial state, the problem is to find a sequence of actions that constitute a legal execution of the high-level program. When there is no non-determinism in the agent program, then the problem is straight forward. However, if the agent program consists of actions that are composed only by non-deterministic operators, then the problem is identical to the planning problem.

In the Golog language family, the state of an agent is a set of fluents, while the state of a 2APL agents consists of beliefs, goals, events, and plans. While the main focus of the Golog family of programming languages is planning, 2APL emphasizes on the dynamics of its internal state by providing a variety of update and revision actions on its belief, goals, events, and plans. Moreover, in contrast to the Golog family of programming languages, 2APL combines logic programming and imperative programming. Finally, the execution of an agent program in 2APL is a cyclic execution of transitions through which its internal state changes, while the execution of Golog programs is on-line planning and plan execution. It should be noted that in contrast to KGP that uses abductive logic programming for planning, the Golog language family use situation calculus for planning.

KGP [17,6,24] is based on a model of agency characterized by a set of modules. The model has an internal state module consisting of a collection of knowledge bases, the current agent's goals and plan. The knowledge bases represent different types of knowledge such as the agent's knowledge about observed facts, actions, and communication, but also knowledge to be used for planning, goal decision, reactive behavior, and temporal reasoning. The KGP agent model includes also a module consisting of a set of capabilities such as planning, reactivity, temporal reasoning, and reasoning about goals. These capabilities are specified by means of abductive logic programming or logic programming with priorities. Another KGP module contains a set of transitions to change the agent's internal state. Each transition performs one or more capabilities, which in turn use different knowledge bases, in order to determine the next state of the agent. Finally, the KGP model has a module, called cyclic theory, that determines which transition should be performed at each moment of time. Different cycle theories can be used to specify different behavior profiles. An agent programmer can implement only some part of the KGP model, i.e., some of the contents of the knowledge

bases, the content of the cycle theory, and some selection operators used in the cycle theory. The rest of the model is fixed and already implemented.

The KGP model is similar to the underlying model of 2APL as it provides constructs to implement an agent's belief/knowledge, goals and plans. The model differs from 2APL as it allows agents to perform capabilities such as planning and goal decision, though the computational complexity of these capabilities are not discussed in [17,6,24]. Moreover, the KGP model is based on propositional language, while 2APL use the first-order representation for beliefs, goals, and plans. Although KGP authors indicate that the choice for propositional language is for the simplicity reasons, it is not clear whether capabilities such as planning and goal decisions can be maintained for a more expressive representation of knowledge bases. Another difference is that a KGP agent's plan is a partially ordered set of primitive actions, while 2APL plans consists of actions that are composed by a variety of complex composition operators such as conditional choice, iteration, and non-interleaving operators. 2APL provides also a larger variety of actions such as belief and goal update actions, test actions, and external actions. The actions in KGP plans are similar to the 2APL external actions that can be performed in external environments. Finally, the KGP cyclic theory is closely related to the 2APL deliberation cycle, which also can be modified by the agent programmers to a certain extent. In summary, while KGP model aims at presenting an agent model with a variety of capabilities such as planning and goal decision, 2APL aims at providing a more expressive representation for the agent's internal state.

Like 2APL and KGP, the general idea of Minerva [18] is to specify an agent's state and its dynamics. Minerva agents consists of a set of specialized sub-agents manipulating a common knowledge base, where sub-agents (i.e., planner, scheduler, learner, etc.) evaluate and manipulate the knowledge base. These subagents are assumed to be implemented in arbitrary programming languages. Also, like 2APL and KGP, Minerva gives both declarative and operational semantics to agents allowing the internal state of the agent, represented by logic programs, to modify. However, in contrast to KGP, Minerva is based on multidimensional dynamic logic programming and uses explicit rules for modifying its knowledge bases. The comparison between KGP and Minerva and the comparison between 3APL and Minerva that are provided in [17] and [18], respectively, can be useful for the further comparison between 2APL and Minerva.

Finally, the programming language (Concurrent) MetateM [12] is mainly based on the direct execution of propositional temporal logic specifications. In (Concurrent) MetateM, the beliefs of agents are propositions extended with modal belief operators (allowing agents to reason about each others' beliefs), goals are temporal eventualities, and plans are primitive actions. In 2APL, we use more expressive representation for beliefs, goals, and plans by using first-order formula with variables. The variables play an essential role in programming as they can be used to pass information from beliefs, goals and events to actions and plans. Moreover, 2APL proposes a varieties of actions and plans with complex structures. Thus, while (Concurrent) MetateM has decided for the full logic rather than having first-order representations for beliefs, goals and plans [12], 2APL restricts the logic to enable more expressive representation of these ingredients. Moreover, 2APL provides facility to handle plan failures, which is absent in the (Concurrent) MetateM.

7 Conclusion and future works

In this article, we presented a BDI-based multi-agent programming language called 2APL, which provides practical constructs for the implementation of cognitive agents. The complete

syntax and semantics of this programming language are presented and discussed. Because of the space limitation we could not formulate and discuss properties of the presented semantics in formal details, but could only briefly and informally explain them. However, we have studied and shown some properties of agent programming languages that are closely related to 2APL in [9]. Moreover, we presented in [1] a logic of agent programs that allows us to formulate and verify safety and liveness properties of specific 2APL-like agent programs. The main purpose here was to present 2APL rather than discussing its formal properties.

We have implemented this semantics in the form of an interpreter that executes 2APL multi-agent programs. The execution of a multi-agent program is the interleaving execution of individual agent programs and the shared environments. The execution of an individual agent program is based on the deliberation cycle. Each cycle determines which transitions in which order should take place. This interpreter is integrated in the 2APL platform that can be downloaded from http://www.cs.uu.nl/2apl. This 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 (`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.

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 aim is to introduce programming constructs at multi-agent level to allow the implementation of social and organisational issues such as norms, obligations, prohibition, permissions, power relation, delegations of tasks, responsibility, and trust. These constructs should enable a programmer to implement under which condition individual agent transitions are allowed and what are the consequences of such transitions. 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 transition takes place and to update it after the transition took place.

We are also working on programming constructs at individual agent level to allow the implementation of various goals types such as perform, achieve, and maintenance goals. We have studied these goal types and discussed their formal properties elsewhere [8]. Moreover, as goals can conflict with each other (in the sense that goals *can* or *would* not be pursued simultaneously), we are working on programming constructs that enable the specification of conflicting goals at individual agent level. These constructs are designed to manage the pursuance of goals, i.e., to determine which goals and in which order they can be pursued/planned [27]. Finally, the relation between goals and plans should be established through the so-called intentions in order to allow the implementation of various commitment strategies such as blindly-minded, single-minded, and open-minded.

We are currently extending the 2APL platform with (visual) debugging tools that allow an agent programmer to monitor the execution of multi-agent programs, halt their executions by means of break points, inspect the mental states of individual agents, and analyze the interactions among individual agents. The monitoring tools are based on temporal expressions that should hold in multi-agent states during the executions of multi-agent programs. In particular, these tools are triggered and activated as soon as temporal expressions do not hold in the multi-agent configurations. The execution of multi-agent programs can also be

halted explicitly by means of break points that are placed in the codes of multi-agent and individual agent programs.

Acknowledgements Thanks to John-Jules Ch. Meyer for many discussions and comments about the design of this multi-agent programming language. We published a short version of 2APL in ProMAS'07 [10].

References

1. Alechina, N., Dastani, M., Logan, B., & Meyer, J.-J. (2007). A logic of agent programs. In *Proceedings of the twenty-second conference on Artificial Intelligence (AAAI-07)*. AAAI press.
2. Astefanoaei, L., Mol, C., Sindlar, M., & Tinnemeier, N. (2008). Going for Gold with 2APL. In *Proceedings of the fifth international workshop on programming multi-agent systems*. Springer.
3. Bellifemine, F., Bergenti, F., Caire, G., & Poggi, A. (2005). JADE—a java agent development framework. In *Multi-agent programming: languages, platforms and applications*. Kluwer.
4. Bergenti, F., Gleizes, M.-P., & Zambonelli, F. (Eds.). (2004). *Methodologies and software engineering for agent systems*, Vol. 11 of Multiagent systems, artificial societies, and simulated organizations. Kluwer Academic Publisher.
5. Bordini, R. H., Wooldridge, M., & Hübner, J. F. (2007). *Programming multi-agent systems in agentspeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.
6. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Sadri, P. M. F., Stathis, K., Terreni, G., & Toni, F. (2004). The KGP model of agency for global computing: Computational model and prototype implementation. In *Global computing*, Vol. 3267 of *Lecture notes in computer science* (pp. 340–367). Springer.
7. Cohen, P. R., & Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42.
8. Dastani, M., van Riemsdijk, M. B., & Meyer, J.-J. C. (2006). Goal types in agent programming. In *Proceedings of the 17th European conference on artificial intelligence (ECAI'06)*.
9. Dastani, M., van Riemsdijk, M. B., & Meyer, J.-J. C. (2007). A grounded specification language for agent programs. In *Proceedings of the sixth international joint conference on autonomous agents and multiagent systems (AAMAS'07)*. ACM Press.
10. Dastani, M., & Meyer, J.-J. (2007). A practical agent programming language. In *Proceedings of the fifth international workshop on programming multi-agent systems (ProMAS'07)*.
11. Fisher, M. (1994). A survey of concurrent METATEM—the language and its applications. In D. M. Gabbay & H. J. Ohlbach (Eds.), *Temporal Logic—Proceedings of the first intemational conference (LNAI Vol. 827)* (pp. 480–505). Springer-Verlag: Heidelberg, Germany.
12. Fisher, M. (2005). METATEM: The story so far. In *Proceedings of the third international workshop on programming multiagent systems (ProMAS-03)*, Vol. 3862 of *lecture notes in artificial intelligence* (pp. 3–22). Springer Verlag.
13. Giacomo, G. D., Lesperance, Y., & Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence, 121*(1–2), 109–169.
14. Hindriks, K. V., de Boer, F. S., van der Hoek, W., & Meyer, J.-J. C. (1999). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems, 2*(4), 357–401.
15. Hindriks, K. V., de Boer, F. S., van der Hoek, W., & Meyer, J.-J. C. (1998). A formal embedding of AgentSpeak(L) in 3APL. In: *Advanced topics in artificial intelligence (LNAI 1502)* (pp. 155–166).
16. Hindriks, K. V., de Boer, F. S., van der Hoek, W., & Meyer, J.-J. C. (2001). Agent programming with declarative goals. In *Proceedings of the 7th international workshop on intelligent agents VII. agent theories architectures and languages* (pp. 228–243). Springer-Verlag.
17. Kakas, A., Mancarella, P., Sadri, F., Stathis, K., & Toni, F. (2004). The KGP model of agency. In *The 16th European conference on artificial intelligence* (p. 3337).
18. Leite, J. A., Alferes, J. J., & Pereira, L. M. (2001). Minerva—A dynamic logic programming agent architecture. In J.-J. Meyer & M. Tambe (Eds.), *Pre-proceedings of the eighth international workshop on agent theories, architectures, and languages (ATAL-2001)* (pp. 133–145).
19. Meyer, J.-J. C., van der Hoek, W., & van Linder, B. (1999). A logical approach to the dynamics of commitments. *Arificial Intelligence, 113*, 1–40.

20. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In *Multi-agent programming: Languages, platforms and applications*. Kluwer.
21. Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In R. van Hoe (Ed.), *Proceedings of the seventh European workshop on modelling autonomous agents in a multi-agent world (MAAMAW'96)*. The Netherlands: Eindhoven.
22. Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes & E. Sandewall (Eds.), *Proceedings of the second international conference on principles of knowledge representation and reasoning (KR'91)* (pp. 473–484). Morgan Kaufmann.
23. Ricci, A., Piunti, M., Acay, L. D., Bordini, R., Hnbner, J., & Dastani, M. (2008). Integrating heterogeneous agent-programming platforms within artifact-based environments. In *Proceedings of the seventh international joint conference on autonomous agents and multiagent systems (AAMAS'08)*. ACM Press.
24. Sadri, F. (2005). Using the KGP model of agency to design applications. In *CLIMA VI* (Vol. 3900, pp. 165–185). Springer.
25. Sardina, S., Giacomo, G. D., Lespérance, Y., & Levesque, H. J. (2004). On the semantics of deliberation in indigolog ù from theory to implementation. *Annals of Mathematics and Artificial Intelligence, 41*(2–4), 259–299.
26. Thangarajah, J., Padgham, L., & Winikoff, M. (2003). Detecting & avoiding interference between goals in intelligent agents. In *Proceedings of the 18th international joint conference on artificial intelligence*.
27. Tinnemeier, N., Dastani, M., & Meyer, J.-J. C. (2007). Goal selection strategies for rational agents. In *Proceedings of the LADS workshop (languages, methodologies and development tools for multi-agent systems)*.
28. Vergunst, N., Steunebrink, B., Dastani, M., Dignum, F., & Meyer, J. (2007). Towards programming multimodal dialogues. In *Proceedings of the workshop on communication between human and artificial agents (CHAA'07)*. IEEE Computer Society Press.
29. Weyns, D., & Holvoet, T. (2004). A formal model for situated multi-agent systems. In B. Dunin-Kêplics & R. Verbrugge (Eds.), *Fundamenta Informaticae, 63*(2–3), 125–158. http://fi.mimuw.edu.pl/abs63.html.
30. Winikoff, M. (2005). JACK^{TM} intelligent agents: An industrial strength platform. In *Multi-agent programming: Languages, platforms and applications*. Kluwer.