Chapter 2

PROGRAMMING MULTI-AGENT SYSTEMS IN 3APL

Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer

Institute of Information and Computing Sciences
Utrecht University
The Netherlands
{mehdi,birna,jj}@cs.uu.nl

Abstract This chapter presents 3APL, which is a multi-agent programming language, and its corresponding development platform. The 3APL language is motivated by cognitive agent architectures and provides programming constructs to implement individual agents directly in terms of beliefs, goals, plans, actions, and practical reasoning rules. The syntax and semantics of the 3APL programming language is explained. Various features of the language and platform and some software engineering issues are discussed.

Keywords: Multi-Agent Programming Language, Cognitive Agents, Multi-Agent Systems

2.1 Motivation

In research on agents, besides architectures, the areas of agent theories and agent programming languages are distinguished. Theories concern descriptions of (the behavior of) agents. Agents are often described using logic [181, 224]. Concepts that are commonly incorporated in such logics are for instance knowledge, beliefs, desires, intentions, commitments, goals and plans.

It has been argued in the literature that it can be useful to analyze and specify a system in terms of these concepts [58, 182]. However, if the system would then be implemented using an arbitrary programming language, it will be difficult to verify whether it satisfies its specification: if we cannot identify what for instance the beliefs, desires and intentions of the system are, it will be hard to check the system against its specification expressed in these terms. This is referred to by Wooldridge as the problem of ungrounded semantics for agent specification languages [238][1]. It will moreover be more difficult to go from specification to implementation if there is no clear correspondence between the concepts used for specification and those used for implementation.

To support the practical development of intelligent agents, several programming languages have thus been introduced that incorporate some of the concepts from agent logics. 3APL ("triple-a-p-l") is one such language. The first version of 3APL was designed by Hindriks et al. [107]. In this version, beliefs, plans[2], and rules for revising plans are the basic building blocks of 3APL agents. An extension to this first version was the addition of declarative goals [54, 228]. Declarative goals[3] describe the state an agent wants to reach and can be used to program pro-active behavior. Plans form the procedural part of an agent and can be executed by the agent in order to achieve its goals. The notion of a goal is important in agent logics and the extension of 3APL with goals is thus important if we are to deal with the issue of ungrounded semantics. Together with the addition of goals, rules were introduced to generate plans on the basis of these goals (and beliefs). Another extension to 3APL was the addition of communication to allow describing multi-agent 3APL systems [53], in the vein of work on ACPL [225].

A 3APL agent thus consists of beliefs, plans, goals and reasoning rules. Given these mental attitudes, issues arise with respect to the operation of the agent; these are issues such as which plan should be executed at a certain point, which goal(s) should be pursued, which (type of) rule should be applied, etc. The choices made affect the operation of the agent and it is thus an important point to consider. To be able to make these kinds of choices explicit, Hindriks et al. introduced a meta-language on top of basic 3APL [107]. This deliberation language was extended by Dastani et al. [52] and includes constructs for tests, planning, and different types of selection functions by means of which plans and rules can be selected.

In this paper, we present the concrete syntax and semantics of the 3APL programming language and give examples to illustrate how cognitive agents can be implemented. The presented version of 3APL is extended with a shared environment in which 3APL agents can perform actions. We then discuss the use of the 3APL programming language from a software engi-

[1]Note that the way the problem is named suggests the problem resides in the specification language, which uses terms that do not relate to computational notions, and should therefore be changed). Although we agree that there is a problem here, we believe that it might also be solved by introducing the notions used in the specification language into the implementation (viz. the programming language), thus in effect *grounding* the specification language.
[2]What we refer to as plans are called "goals" in [107].
[3]From now on, we will use the term "goal" to refer to the notion of declarative goal.

neering point of view and describe the 3APL platform that supports the development of 3APL multi-agent systems.

2.2 Language

In general, the implementation of a multi-agent system requires two programming languages: one single-agent programming language to implement individual agents, and one multi-agent programming language to implement multi-agent aspects, such as which and how individual agents should be executed. The multi-agent programming language can be used to implement organization and coordination of multi-agent systems directly and explicitly. Using the multi-agent programming language one can, for example, implement sequential or parallel execution of individual agents or block the execution of individual agents when their actions are not permitted.

A 3APL multi-agent system consists of a set of concurrently executed 3APL agents that can interact with each other either directly through communication or indirectly through the shared environment. In order to implement a 3APL multi-agent system, the 3APL platform has been built to support the design, implementation, and execution of a set of 3APL agents that share an external environment. The 3APL platform thus allows the implementation and parallel execution of a set of 3APL agents and therefore it fulfills the function of a 3APL multi-agent programming language. This choice implies that all organization and coordination issues should be implemented implicitly through the implementations of individual 3APL agents.

The individual 3APL agents can be implemented by the 3APL programming language that facilitates direct implementation of various aspects of cognitive agents, and the shared environment can be implemented in the Java [99] programming language. In particular, the shared environment is implemented as a Java class such that its methods correspond with the actions that agents can perform in the environment. Besides the interaction with the environment, the agents can interact with each other through direct communication. Using 3APL, one can implement agents that observe the shared environment, communicate with each other, reason about and update their states, and execute actions in the shared environment.

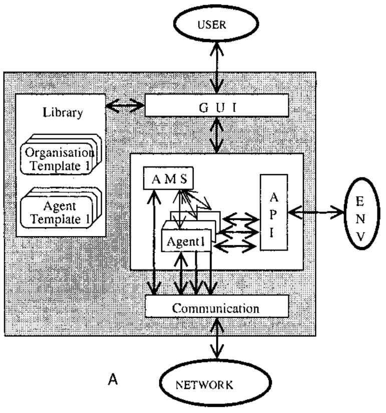
In designing the 3APL programming language, a separation was created between mental attitudes (data structures) and the deliberation process (programming instructions) that manipulate the mental attitudes. Therefore, the 3APL programming language consists of programming constructs to implement the agent's mental attitudes, represented as data structures, as well as the agent's deliberation process, represented as instructions, to manipulate the mental attitudes. In particular, 3APL allows direct specification of mental attitudes such as beliefs, goals, plans, actions and reasoning rules. Actions form the basic building blocks of plans and can be internal mental actions, external actions, or communication actions. The deliberation constructs allow the implementation of selection and execution of actions and plans through which an agent's belief base can be updated and through which the shared environment can be modified. It also allows the selection and application of reasoning rules through which the goal and plan bases can be modified.

The basic deliberation constructs can be composed by means of sequential composition and by using if-then-else and while constructs, forming the deliberation language (see [52] for the formal specification). This enables the programmer to implement, for example, a deliberation program that consists of (the iteration of the sequential composition of) two conditional iterations (while-loops) such that the condition of the first holds as long as there is no emergency situation while the condition of the second holds as long as there is an emergency situation. The body of the first iteration could then be used to plan new goals, while the body of the second could generate emergency plans and execute them. This example illustrates that the language is expressive enough to implement important aspects of subsumption architectures [36], in which emergency behavior can be realized at the reactive layer while complex behavior can be realized at higher deliberative layers. Note that also the usual 'standard' sense-reason-act cycle can be implemented in this deliberation language.

This view on programming multi-agent systems has resulted in the 3APL multi-agent platform architecture and the 3APL agent architecture, as illustrated in figure 2.1. The 3APL platform consists of a number of agents, a directory facilitator called agent management system (AMS), a message transport system which delivers messages between agents, a shared environment, and a plugin interface that allows agents to execute actions in the shared environment. The function of the agent management system is to register agents that are loaded and executed on the platform and it answers a set of questions from agents about other agents that are present on the platform. These questions can be, for example, about the names of agents, their functions, and the services they provide. Each individual 3APL agent consists of a belief base, a goal base, a plan base, an action base for the specification of internal mental actions, a base for goal planning rules (which can be applied to plan a goal), and a base for plan revision rules (which can be used to revise, adopt, and drop plans).

2.2.1 Specifications and Syntactical Aspects

In the following subsections, we explain how various ingredients of the individual 3APL agent architecture and the 3APL platform can be imple-



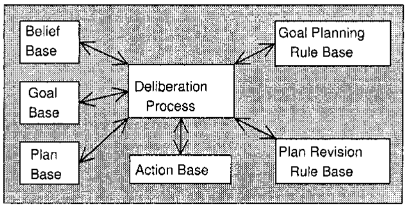


Figure 2.1. The architectures of 3APL platform (A) and individual 3APL agents (B)

mented. In particular, we describe the programming constructs to implement individual agents, explain how the deliberation cycle of individual agents can be implemented, and discuss the implementation of the shared environment. Before starting to describe the programming constructs for implementing individual agents, we present the EBNF grammar of the language.

The EBNF specification of the 3APL programming language for individual agents is illustrated in Figure 2.2.1. In this specifications, we use $\langle atom \rangle$ to denote an atomic formula[4] the terms of which can include Prolog-like list representations of the form `[a,b,[3,f]]`, `[X|T]`, and `[a,[4,d]|T]`, etc. Moreover, we use $\langle ground\_atom \rangle$ to denote a ground atomic formula, which is an atomic formula that contains no variables. The terms of ground atomic formulae can include Prolog-like list representations such as `[a,b,c]`, `[e,[9,d,g],3]`. Finally, we use $\langle Atom \rangle$ to denote atomic formulae where the predicate letter starts with a capital letter, $\langle ident \rangle$ to denote a string, and $\langle var \rangle$ to denote a variable.

Beliefs and goals

The *beliefs* of a 3APL agent describe the situation the agent is in. Beliefs are implemented by the belief base, which contains information the agent believes about the world. The *goals* of the agent on the other hand denote

[4]A predicate name parameterized with a number of terms, e.g. `on(a,b)`.

```
⟨Program⟩ ::=          "Program" ⟨ident⟩
                       ( "Load" ⟨ident⟩ )?
                       "Capabilities :" ( ⟨capabilities⟩ )?
                       "BeliefBase :" ( ⟨beliefs⟩ )?
                       "GoalBase :" ( ⟨goals⟩ )?
                       "PlanBase :" ( ⟨plans⟩ )?
                       "PG − rules :" ( ⟨p_rules⟩ )?
                       "PR − rules :" ( ⟨r_rules⟩ )?

⟨capabilities⟩ ::=     ⟨capability⟩ ( "," ⟨capability⟩ )*
⟨capability⟩ ::=       "{" ⟨query⟩ "}" ⟨Atom⟩ "{" ⟨literals⟩ "}"
⟨beliefs⟩ ::=          ( ⟨belief⟩ )*
⟨belief⟩ ::=           ⟨ground_atom⟩ "." | ⟨atom⟩ ": −" ⟨literals⟩"."
⟨goals⟩ ::=            ⟨goal⟩ ( "," ⟨goal⟩ )*
⟨goal⟩ ::=             ⟨ground_atom⟩ ( "and" ⟨ground_atom⟩ )*
⟨plans⟩ ::=            ⟨plan⟩ ( "," ⟨plan⟩ )*
⟨plan⟩ ::=             ⟨basicaction⟩ | ⟨composedplan⟩
⟨basicaction⟩ ::=      "ϵ" | ⟨Atom⟩ | "Send("⟨iv⟩,⟨iv⟩,⟨atom⟩")" |
                       "Java("⟨ident⟩,⟨atom⟩,⟨var⟩")" | ⟨wff⟩"?" | ⟨atom⟩
⟨composedplan⟩ ::=     "if" ⟨wff⟩ "then" ⟨plan⟩ ( "else" ⟨plan⟩ )? |
                       "while" ⟨query⟩ "do" ⟨plan⟩ |
                       ⟨plan⟩ ";" ⟨plan⟩
⟨p_rules⟩ ::=          ⟨p_rule⟩ ( "," ⟨p_rule⟩ )*
⟨p_rule⟩ ::=           ⟨atom⟩ "<−" ⟨query⟩ "|" ⟨plan⟩
⟨p_rule⟩ ::=           "<−" ⟨query⟩ "|" ⟨plan⟩
⟨r_rules⟩ ::=          ⟨r_rule⟩ ( "," ⟨r_rule⟩ )*
⟨r_rule⟩ ::=           ⟨plan⟩ "<−" ⟨query⟩ "|" ⟨plan⟩
⟨literals⟩ ::=         ⟨literal⟩ ( "," ⟨literal⟩ )*
⟨literal⟩ ::=          ⟨atom⟩ | "not("⟨atom⟩")"
⟨wff⟩ ::=              ⟨literal⟩ | ⟨wff⟩ "and" ⟨wff⟩ | ⟨wff⟩ "or" ⟨wff⟩
⟨query⟩ ::=            ⟨wff⟩ | "true"
⟨iv⟩ ::=               ⟨ident⟩ | ⟨var⟩
```

Figure 2.2. The EBNF specification of the 3APL language for programming individual agents.

the situation the agent wants to realize, which is implemented by an agent's goal base.

The belief base is implemented by a Prolog program consisting of Prolog facts and rules. The initial belief base of a 3APL agent is preceded by the keyword "BeliefBase :". Note that the syntax of Prolog is in accordance with the specification of ⟨*beliefs*⟩ as given above. The following is an example of the initial belief base of a 3APL agent which indicates that blocks a and b are on the floor, block c is on block a, and that a block is clear if there is no block placed on top of it.

```
BeliefBase:
on(a,fl).
on(b,fl).
on(c,a).
clear(Y) :- not(on(X,Y)).
```

Note that, like in Prolog, the specification of beliefs allows the use of negation in the body of the rules. The `not` in these rules stands for negation-as-failure.

We allow individual agents to load a separate file containing the background knowledge. The syntax of the background knowledge is the same as the syntax of beliefs and is implemented by a Prolog program that can be loaded into the initial belief base of an agent through the optional `"Load"` construct. The argument of the load construct is the name of a file that contains a Prolog program. Such a file can be loaded by different agents. In this way, one can implement the background knowledge once and allow different agents to load it as part of their initial beliefs.

The goal base of a 3APL agent is a set of goals, each of which is implemented by a conjunction of ground Prolog atoms. The initial goal base of a 3APL agent is preceded by the keyword `"GoalBase:"`. The following is an example of the initial goal base of a 3APL agent which indicates that the agent has two goals. The first goal is to have block `a` on block `b` and block `b` on block `c`, and the second goal is to have block `d` on the floor.

```
GoalBase:
on(a,b) and on(b,c) , on(d,fl)
```

The difference between the two goals in this goal base and the single goal `on(a,b) and on(b,c) and on(d,fl)` is that the two separate goals in the goal base may be fulfilled at different times, whereas the three conjuncts of the single goal have to be satisfied at the same time.

As we will see below, it is useful to be able to check whether a formula follows from the belief base or the goal base, for example for test actions, for the application of reasoning rules, or for performing mental actions. For these purposes, we use the so-called belief and goal query expressions (i.e. $\langle query \rangle$) which are either the special atomic formulae `true` or a well-formed formula (i.e. $\langle wff \rangle$) constructed from atoms and logical connectors. In the implementation of 3APL, the keywords `and`, `or`, and `not` are used as logical connectives. For example, `(on(X,b) and on(b,Y)) or not(on(b,fl))` can be a belief query expression which is derivable from the belief base if either `on(X,b) and on(b,Y)` is derivable from the belief base or `on(b,fl)` is not derivable[5].

Basic Actions

In order to reach its goals, a 3APL agent adopts plans. A plan is built from basic actions that can be composed through co-called program operators. We first discuss the various kinds of basic actions and then explain how they can be composed to form plans. In 3APL, beside the neutral action (denoted by ϵ) that does not change the current state of affairs, five other types of actions are distinguished: mental actions, communication actions, external actions, test actions, and so-called abstract plans.

The *mental actions* can update the belief base of agents, if successfully executed. A mental action has the form of an atomic formula and thus consists of a predicate name and a list of terms with the exception that the first letter of the predicate name is a capital letter (i.e. $\langle Atom \rangle$). The effect of the execution of a mental action is a change in the agent's belief base. The conditions under which a mental action can be successfully executed (also called the precondition of the mental action), and its effects on the belief base (also called the post-condition of the mental action) should be specified in the 3APL program.

The pre- and post-conditions of mental actions are specified through so-called capabilities which consist of three parts: the mental action itself (i.e. $\langle Atom \rangle$), a pre-condition which is a belief query expression (i.e. a $\langle query \rangle$), and a post-condition which is a list of literals (i.e. $\langle literals \rangle$). An agent can execute a mental action if the pre-condition of the corresponding capability holds. The effect of the execution of a mental action is then a change in the agent's belief base such that the post-condition of the corresponding capability holds. In order to realize this effect, a function is defined in the interpreter that adds the positive literals to the belief base and retracts the atoms of the negative literals from the belief base, if present. In the implementation of 3APL, the specification of capabilities is preceded by the keyword `"Capabilities:"`. The following is an example of a capability that defines the effect of the mental `Move` action.

```
Capabilities:
{on(X,Y)} Move(X,Y,Z) {not(on(X,Y)) , on(X,Z)}
```

The idea is, that the action `Move(X,Y,Z)` moves a block `X` from

[5]Note that as we use the Prolog reasoning engine to implement the evaluation of the query expressions, the `or` and `and` operators are not commutative.

block Y to block Z. If this Move(X,Y,Z) action is executed, the variables X, Y and Z will be instantiated with a value. Assume for example that X = a, Y = b and Z = c. The action can then be executed in case on(a,b) is derivable from the belief base, i.e., if block a is on b. The result should be that not(on(a,b)) and on(a,c) are derivable from the belief base. This is implemented by removing fact on(a,b) and adding on(a,c).

A *send action* can be used to pass a message to another agent. A message contains the name of the receiver of the message, the speech act or performative (e.g. inform, request, etc.) of the message, and the content. The send action is like an atomic formula which has Send as the predicate name and has three arguments. The first argument is either an identifier or a variable (i.e. $\langle iv \rangle$) denoting the name of the receiving agent, the second argument is also either an identifier or a variable (i.e. $\langle iv \rangle$) denoting the performative of the message, and the third argument is an atomic formula (i.e. $\langle atom \rangle$), which specifies the content of the message. If the receiver or the performative is a variable, they should be instantiated with constants denoting the name of the receiver and the performative, respectively, before the send action is executed. An example of a send action is Send(ag_2, inform, on(a,b)), which specifies that agent ag_1 informs agent ag_2 that block a is on block b.

If an agent sends a message Send(Receiver, Performative, Content) to another agent, the belief base of the sender is updated with the formula sent(Receiver, Performative, Content) and the belief base of the receiver is updated with the formula received(Sender, Performative, Content). Agents can receive a message in their belief base at each moment in time. Note that unlike the mental actions, the send actions can always be executed.

The *external actions* are means to change the external environment in which the agents operate. The effects of external actions are assumed to be determined by the environment and might not be known to the agents. The agent thus decides to perform an external action and the external environment determines the effect of this action. The agent can come to know the effects of an external action by performing a sense action. This sense action can be defined as an external action in an agent's plan, or it could be a pre-defined operation that is part of the sense-reason-act loop of the agent's deliberation cycle.

External actions are performed by 3APL agents with respect to an environment which is assumed to be implemented as a Java class. In particular, the actions that can be performed in this environment are determined by the methods of the Java class (i.e., the methods specify the effect of those actions in that environment), and the state of the environment is represented by the instance variables of the class.

The external actions that can be performed by 3APL agents have the form `Java(Classname, Method, List)` where `Classname` is the name of the Java class that implements the environment, `Method` is the action to be performed in the environment, and `List` is a list of returned values. The parameter `Method` corresponds with a parameterized method of the Java class `Classname` and `List` is a list of values returned by `Method`. The method can be implemented to return the result of the action in the list, or the list could for example be empty. In that case, an explicit sense action would have to be executed to obtain the result of the action.

An example of an external action is `Java(BlockWorld, east(), L)` where the external action `east()` is performed in the environment `BlockWorld`.[6] The effect of this action is that the position of the agent in the block world environment is shifted one slot to the east.

A *test action* checks whether a well-formed formula (i.e. $\langle wff \rangle$) is derivable from the belief base. Such an action, which consists of a well-formed formula followed by a question mark, will be blocked if the formula is not derivable from the belief base. Note that the derivation relation is implemented by the Prolog reasoning engine. If the arguments of a test action are variables and the well-formed formula is derivable from the belief base, then the effect of the test action is a substitution that assigns terms to the variables. The assignment is useful for retrieving information from the belief base and passing it to other actions for further manipulation.

An example of a test action is `(on(a,X) and on(X,c))?` which will be successfully executed if the agent believes that there is a block `X` placed on top of block `c` such that block `a` is placed on top of it. The result of a successful execution is a substitution such as `{X/b}` which indicates that the relevant block is block `b`.

An *abstract plan*, which is represented as an atomic formula (i.e. $\langle atom \rangle$), is an abstract representation of a plan which can be instantiated with a (more concrete) plan during execution. An abstract plan cannot be executed directly and should be rewritten into another plan, possibly (and even probably) containing executable basic actions, through application of reasoning rules (see below for a detail description of these rules). The application of rules to abstract plans involves a unification of abstract plans with the head of rules through which values can be passed to the instantiated plan.

[6] `BlockWorld` is in this case a two-dimensional grid with obstacles in which the agents may move in any direction that is not blocked by obstacles (or walls).

Plans

Basic actions, as discussed above, can be composed to build plans through so-called program operators. There are three 3APL program operators: the sequential operator (denoted by `;`), the iteration operator (denoted by a `while-do` construct), and the conditional choice operator (denoted by an `if-then-else` construct). In particular, if β is a well-formed formula, β' is a query expression (i.e. a well-formed formula or `true`), and *Actions* is the set of basic actions as defined above, then the set of plans, denoted by *Plans* is defined as follows:

- $Actions \subseteq Plans$
- if $\pi, \pi' \in Plans$, then `if` β `then` π `else` π' $\in Plans$
- if $\pi \in Plans$, then `while` β' `do` π $\in Plans$
- if $\pi, \pi' \in Plans$, then $\pi ; \pi' \in Plans$

We use ϵ to denote the empty plan and we identify $\epsilon;\pi$ with π.

The plan base of a 3APL agent consists of a set of plans. In the implementation of 3APL, the specification of the initial plan base of an agent is preceded by the keyword `"PlanBase :"` and consists of a number of plans separated by a comma. The following is an example of the initial plan base of a 3APL agent.

```
PlanBase:
while (on(X,fl) and not(on(V,X)) do {
        (on(Y,Z) and not(Z==fl))?;
        Move(X,fl,Y)
}
```

This plan base consists of one plan which will find all free blocks (blocks with no block on top) that are placed on the floor and move them to an existing block which itself is not placed on the floor.

Reasoning Rules

In order to reason with goals and plans, 3APL has two types of rules: goal planning rules and plan revision rules. These rules are conditionalized by beliefs. Let β be a query expression, κ be an atomic formula, and π, π_h, π_b be plans. The set of goal planning rules (*PG*) and the set of plan revision rules (*PR*) are then defined as follows:

$$\kappa \leftarrow \beta \mid \pi, \;\; \leftarrow \beta \mid \pi \in PG$$
$$\pi_h \leftarrow \beta \mid \pi_b \in PR.$$

The *goal planning rules* are used to generate plans to achieve goals. In the first goal planning rule, the belief condition β indicates when the plan π could be generated to achieve the specified goal κ. The second goal planning rules can be used to model reactive behavior by omitting the head of the rule. This special kind of goal planning rule states that under the belief condition β, a plan can be adopted. The specification of the set of goal planning rules is preceded by the keyword "PG − rules :". The following is an example of the specification of a goal planning rule of a 3APL agent.

```
PG-rules:
on(X,Z) ← on(X,Y)   |  Move(X,Y,Z)
```

This rule states that if the agent wants to have block X on block Z, but it believes that X is on block Y, then it plans to move X from Y onto Z.

The *plan revision rules* are used to revise plans from the plan base. The specification of the set of plan revision rules is preceded by the keyword "PR − rules :". The following is an example of the specification of a plan revision rule of a 3APL agent.

```
PR-rules:
Move(X,Y,Z) ← not(clear(X))   |
                    on(U,X)?;Move(U,X,fl);Move(X,Y,Z)
```

This plan revision rule informally means that if the agent plans to move block X from block Y onto block Z, but it cannot move X because (it believes that) there is a block on X, then the agent should revise its plan by finding out which block (U) is on X, moving U onto the floor, and finally moving X from Y onto Z.

A plan revision rule $\pi_h \leftarrow \beta \mid \pi_b$ can be applied to a plan π, if π_h can be matched to a prefix of π, i.e., if π is of the form $\pi_h;\pi'$. For example, a plan $Move(a,b,c);Move(b,fl,a)$ can be revised into a plan $Move(a,b,fl);Move(b,fl,a)$ by applying the plan revision rule $Move(a,b,c) \leftarrow \texttt{true} \mid Move(a,b,fl)$. Note that a plan revision rule could be used to drop (part of) a plan if its body π_b is the empty plan ϵ.

Deliberation Cycle

The beliefs, goals, plans and reasoning rules form the mental attitudes or data structures of 3APL agents. These data structures can be modified by deliberation operations such as applying a rule or executing a plan. These deliberation operations constitute the deliberation process of individual agents. The deliberation process or program can be viewed as the interpreter, as it determines which deliberation operations should be performed in which order. For example, it can be programmed to determine whether a goal should be dropped if it is not reachable using any possible plan and plan revision rule. A deliberation process programmed in this way could be viewed as an implementation of "single minded" agents [182]. Some more moderate alternatives are also possible. Moreover, the interpreter can determine if and when to check the relation between plans and goals. For example, the interpreter can check whether a goal still exists during plan execution to avoid continuing with a plan of which the goal is reached (or dropped) already. The interpreter can also perform a kind of "garbage collection" and remove a left-over plan for a goal that no longer exists. If this would not be done, the left-over plan could become active again at a later time and this might not be desired behavior.

Another issue that the interpreter can determine is related to multiple (parallel) goals and/or plans. For example, it can decide whether only one or more plans can be adopted for the same goal at any time. It seems not unreasonable to allow only one plan at a time for each goal, which coincides with the idea that we try different plans consecutively and not in parallel, because this might lead to a lot of unnecessary interactions between plans and also a waste of resources. If we allow only one current plan for each goal, the plans in the plan base will all be for different goals. Also in this case one has to determine whether the plans will be executed interleaved or consecutively. Interleaving might be beneficial, but can also lead to resource contention between plans in a way that no plan executes successfully anymore (see also [222, 221, 220]). E.g., a robot needs to go to two different rooms that are in opposite directions. If it has a plan to arrive in each room and interleaves those two plans, it will keep oscillating around its starting position. Many of the existing work on concurrent planning can however be applied in this setting to avoid most problems in this area.

For 3APL, a set of deliberation operations is proposed [52], including `SelectPlanningGoalrule`, `SelectPlanRevisionrule`, `SelectPlan`, `ExecutePlan`, `ApplyPlanningGoalrule`, and `ApplyPlanRevisionrule`. These operations can be composed to form a deliberation program by using operators such as sequential composition, test (on both belief, goal and plan bases), conditional choice (if-then-else construct), and conditional iteration (while loop).

In order to facilitate the implementation of a deliberation process and since the 3APL interpreter is implemented in Java, we have implemented each mental attitude as a Java class, i.e., a Java class for the belief base, one for the capabilities, one for the goal base, one for the plan base, one for the goal planning rule base, and one for the plan revision rule base. Each of these classes has an internal representation for its specific mental attitude, which will initially be set by parsing the input 3APL program. The parser is part of the Java implementation of the 3APL interpreter.

Each class implementing a mental attitude has a set of methods. These methods implement the deliberation operations that are relevant for that mental attitude. For example, the class that implements the belief base has a method for updating the belief with new facts, and the class that implements the goal planning rule base has a method for selecting a goal planning rule and another method for applying that rule. In order to implement a deliberation process for 3APL agents, a programmer should thus have the source code of the interpreter and implement a Java class that calls the methods of the classes that correspond to the mental attitudes.

Although the idea is that the agent programmer implements the deliberation process, an interpreter is provided that implements a cyclic order of deliberation operations as illustrated in figure 2.3. According to this deliberation program, an agent starts with searching for an applicable planning rule (in their order of occurrence) to generate a plan for one of its goals and applies the first applicable planning rule that it finds. The agent then continues with searching for an applicable plan revision rule (in their order of occurrence) to revise one of its plans. A plan needs to be revised when, for example, it starts with an abstract plan which is not executable. The agent applies the first applicable plan revision rule that it finds. Then, the agent continues with searching for the executable plans (in their order of occurrence) and executes the first plan it finds. Note that a plan that starts, for example, with a mental action of which the pre-condition does not hold, cannot be executed. Finally, the agent continues with either the same cycle of operations or it suspends its activities until a message is arrived. The agent suspends its activities if no sensible operation could be performed during the previous cycle, i.e. if no rules could be applied and no plan could be executed. Note that the arrival of a message may make either a rule applicable or a plan executable.

This order of operations is by no means universal, since it does not guarantee the proper agent behavior for all kinds of situations. For example, in an emergency situation it may be more plausible that an agent does not continue executing its current plans, but starts adopting and executing emergency plans. As we have argued in [52], we believe that an agent's interpreter should be programmable to allow the implementation of different types of behavior. The proposed interpreter for 3APL is an example which can in principle be modified by the agent programmers to generate different types of behavior. At this moment, the source code of 3APL is under development and is not available for modifying and implementing the deliberation cycle. However, we hope to make this possible in the near future.

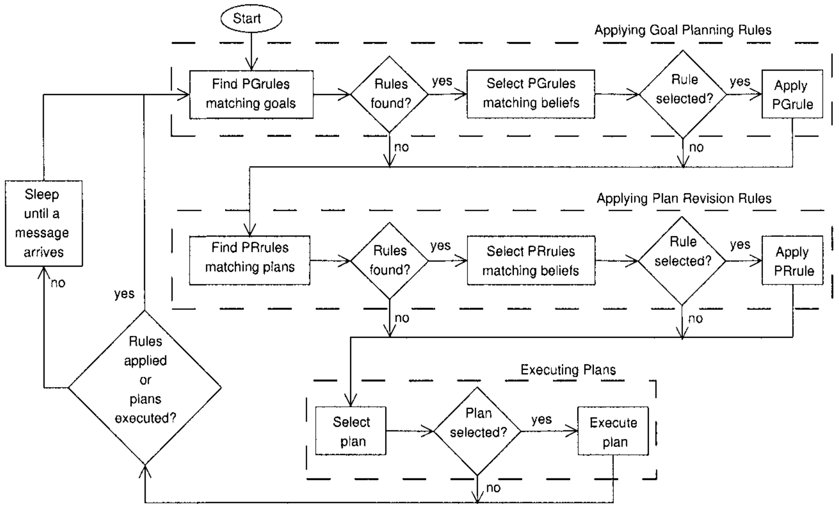

Start
Applying Goal Planning Rules
Find PGrules matching goals
Rules found?
yes
Select PGrules matching beliefs
Rule selected?
yes
Apply PGrule
no
no
Applying Plan Revision Rules
Find PRrules matching plans
Rules found?
yes
Select PRrules matching beliefs
Rule selected?
yes
Apply PRrule
no
no
Sleep until a message arrives
no
yes
Rules applied or plans executed?
Executing Plans
Select plan
Plan selected?
yes
Execute plan
no

Figure 2.3. A cyclic interpreter (deliberation cycle) for the 3APL agents.

3APL Platform

The 3APL platform provides a user interface that allows 3APL agents to be programmed, loaded, and executed. During execution there are various facilities in the interface such as the sniffer, which allows monitoring the exchanges of messages between agents, and specific windows, which allow monitoring the changes of all mental attitudes of individual agents. Also, there are various icons in the interface that allow monitoring the execution of agents, either step by step or continuously. The graphical user interface of the 3APL platform is illustrated in Figure 2.4 and described in section 2.3. A detailed description of the platform interface can be found in the 3APL user guide [51].

The only part of the platform architecture that is programmable to this date is the shared environment. As noted, the environment of 3APL agents is assumed to be implemented as a Java class, the actions that can be performed in this environment are determined by the methods of the Java class (i.e., the methods specify the effect of those actions in that environment), and the state of the environment is represented by the instance variables of the class. In particular, the environment is modelled as plugin to the platform. This is a systematic way to interface between the 3APL platform and Java classes. The plugin facilitates the interaction between individual agents running on the platform and the instantiation of the Java classes. These interactions include method calls from agents to Java classes and event notification from the platform interface. To create a plugin you need to implement three interfaces.

1. `ics.TripleApl.Plugin`: factory class
2. `ics.TripleApl.Instance`: product class
3. `ics.TripleApl.Method`: plugin method (function).

At startup, the platform loads all Plugin-implementing classes from the `plugins/` directory (this directory is created when the 3APL platform is downloaded and unpacked). It then queries the found plugin classes for their external functionalities (Java methods) they provide to individual agents. This is done by the platform through invocation of the method `getMethods` of the Plugin interface. The idea behind the plugin is to systematize the relation between agent platform and environment that can be used by the agents. In particular, the environment should be linked to the *individual* agents running on the platform such that the effect of any change on individual agents (create, reset or remove) on the platform can be realized and passed on to the environment.

For example, consider a two-dimensional grid such as the block world environment in which the agents running on the platform can be present and move around. In such a case, if the user creates, resets or removes an agent on or from the platform, the agent should be added to, reset (moved to initial position), or removed from the block world environment, respectively. The effects of the mentioned events (on the platform) are realized by the platform through invocation of one of the following methods from the Plugin interface: createInstance, resetInstance, and removeInstance. The downloadable version of the 3APL platform comes with an implementation of a block world environment. The details of this environment and its Java implementation are described in the 3APL user guide [51]. Note that this environment is just an example and that the programmer can implement its own environment.

2.2.2 Semantics and Verification

To program a 3APL multi-agent system is to program individual 3APL agents and to specify the initial state of their shared environment. To program an agent means to specify its initial beliefs, goals, plans and capabilities, and to specify sets of goal planning rules and plan revision rules. The initial state of the shared environment is specified by a set of facts.

DEFINITION 2.1 (3APL AGENT) *An individual 3APL agent is a tuple* $\langle \iota, \sigma_0, \gamma_0, Cap, \Pi_0, PG, PR, \xi \rangle$ *where* ι *is the agent identifier,* σ_0 *is the initial belief base,* γ_0 *is the initial goal base,* Cap *is the capability base,* $\Pi_0 \subseteq Plans \times \{\texttt{true}\}$ *is the initial plan base,* PG *is a set of goal planning rules,* PR *is a set of plan revision rules, and* ξ *is the environment the agent shares with other agents, which is represented by a set of ground atoms.*

The plan base of a 3APL agent consists of a set of plan-goal pairs. The goal for which a plan is selected is recorded with the plan, because this for instance provides for the possibility to drop a plan of which the goal is reached. The initial plan base of a 3APL agent consists of a set of plans, rather than a set of plan-goal pairs. We take these initial plans as having the associated goal `true`[7]. Furthermore, goals may be revised or dropped and one might want to remove a plan associated with a goal which has been dropped, from the plan base (see also the discussion on the deliberation cycle of section 2.2.1).

The beliefs, goals and plans of individual agents and their shared environment are the elements that change during the execution of the agent, while the capabilities and the reasoning rules remain unchanged. Together with a *substitution* component, these changing components of the agent constitute a *3APL agent configuration.* The substitution part of the configuration is used to store values or bindings associated with variables.

DEFINITION 2.2 ((GROUND) SUBSTITUTION, BINDING, DOMAIN, FREE VARIABLES) *A substitution* θ *is a finite set of the form* $\{x_1/t_1, \ldots, x_n/t_n\}$, *where* $x_i \in Var$ *and* $t_i \in Term$ *and* $\forall i \neq j : x_i \neq x_j$. θ *is called a ground substitution if all* t_i *are ground terms. Each element* x_i/t_i *is then called a binding for* x_i. *The set of variables* $\{x_1, \ldots, x_n\}$ *is the domain of* θ *and will be denoted by* $dom(\theta)$. *The application of a substitution* θ *to a syntactic expression* e *is denoted as* $e\theta$. *It refers to the expression resulting from simultaneously replacing all occurrences of variable* x *in* e *for which* $x/t \in \theta$ *by* t.

Below, we first define the configuration of an individual 3APL agent in terms of the elements that change during the execution of the agent. Then, we define the configuration of a 3APL multi-agent system in terms of the configurations of the involved agents and their shared environment.

DEFINITION 2.3 (CONFIGURATION) *A configuration of an individual 3APL agent is a tuple* $\langle \iota, \sigma, \gamma, \Pi, \theta, \xi \rangle$, *where* ι *is an agent identifier,* σ *is the belief base of the agent,* γ *is the goal base of the agent,* Π *is the plan base of the agent,* θ *is a ground substitution that binds domain variables to domain terms,*

[7]Although `true` as a logical formula cannot be an agent's goal according to the 3APL semantics, we use it only to indicate that there is no specific goal associated to a plan.

and ξ is the environment it interacts with, where ξ is a set of ground atoms. The goal base in a configuration is such that for any goal $\phi \in \gamma$ it holds that ϕ is not entailed by the agent's beliefs.

A configuration of a 3APL multi-agent system is a tuple $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n, \xi \rangle$ where $\mathcal{A}_i$ for $1 \leq i \leq n$ is the configuration of individual agent i and ξ is the shared environment. This shared environment is the same as the environment of each individual agent.

The rationale behind the condition on the goal base is the following. The beliefs of an agent describe the state the agent is in and the goals describe the state the agent wants to realize. If an agent believes ϕ is the case, it cannot have the goal to achieve ϕ, because the state of affairs ϕ is already realized. This is thus an implementation of achievement goals, as opposed to maintenance goals.

Transition system

In the following, we present the general idea of the type of semantics that is given to the 3APL programming language. It is an operational semantics which is defined in terms of a transition system [169]. A transition system is a set of derivation rules for deriving transitions. A transition is a transformation of one configuration into another and it corresponds to a single computation step. For the purpose of this paper, we present only a subset of derivation rules. A complete set of derivation rules is presented in [54].

We define first a derivation rule for transitions between multi-agent configurations. This derivation rule, which captures the parallel execution of the set of individual agents, forms the only transition at the multi-agent level.

DEFINITION 2.4 (MULTI-AGENT EXECUTION) *Let $\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_n, \mathcal{A}'_i$ be agent configurations and let ξ and ξ' be specifications of the environment. Further, let $\mathcal{A}_i = \langle \sigma, \gamma, \Pi, \theta, \xi \rangle$ and let $\mathcal{A}'_i = \langle \sigma', \gamma', \Pi', \theta', \xi' \rangle$. Then the derivation rule for multi-agent configurations is defined as follows.*

$$\frac{\mathcal{A}_i \rightarrow \mathcal{A}'_i}{\langle \{\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_n\}, \xi \rangle \rightarrow \langle \{\mathcal{A}_1, \ldots, \mathcal{A}'_i, \ldots, \mathcal{A}_n\}, \xi' \rangle}$$

This derivation rule states that a transition between multi-agent configurations can be defined in terms of a transition between single-agent configurations. This amounts to an *interleaved* execution of the agents in the system. Note that the environment of the multi-agent configuration is shared among all individual agents.

We now define transition rules that can derive transitions transforming single-agent configurations. These derivation rules specify the semantics of the execution of plans and the application of reasoning rules.

The first derivation rule specifies the execution of the plan base of a 3APL agent. The plan base of the agent is a set of plan-goal pairs. This set can be executed by executing one of the constituent plans. The execution of a plan can change the agent's configuration.

DEFINITION 2.5 *(plan base execution) Let* $\Pi = \{(\pi_1,\kappa_1),\ldots,(\pi_i,\kappa_i),\ldots,(\pi_n,\kappa_n)\}$ *and* $\Pi' = \{(\pi_1,\kappa_1),\ldots,(\pi_i',\kappa_i),\ldots,(\pi_n,\kappa_n)\}$ *be plan bases,* θ,θ' *be ground substitutions, and* ξ,ξ' *be environment specifications. Then, the derivation rule for the execution of a set of plans is specified in terms of the execution of individual plans as follows.*

$$\frac{\langle \iota,\sigma,\gamma,\{(\pi_i,\kappa_i)\},\theta,\xi\rangle \rightarrow \langle \iota,\sigma',\gamma',\{(\pi_i',\kappa_i)\},\theta',\xi'\rangle}{\langle \iota,\sigma,\gamma,\Pi,\theta,\xi\rangle \rightarrow \langle \iota,\sigma',\gamma',\Pi',\theta',\xi'\rangle}$$

Now we will introduce some of the derivation rules for the execution of individual plans. We introduce derivation rules for external actions, communication actions and tests.

An external action `Java(Classname,` $\alpha(t_1,\ldots,t_n)$`,` x`)` has two functionalities. First, based on the input terms and the state of the environment, it generates a term and assigns it to variable x. The term assigned to x is the output of the action which is returned to the agent from the environment. For sense actions, this output can be programmed to be the sensed information. For other actions, the output could for example be information such as whether the action has been performed, or the result of the action. Note that this term can be a list of terms. Second, actions are assumed to have effects on the environment.

In order to capture these two functionalities, i.e., calculating a value for x and updating the current environment, we assume for each external action with a method name α a function F_α which maps terms $t_1,\ldots,t_n$ and the environment ξ to a term which will be assigned to variable x. Further, we assume a function G_α which maps terms $t_1,\ldots,t_n$ and the environment ξ to a new environment ξ'. An agent can execute an external action only if the goal associated to the action is still a goal of the agent.

DEFINITION 2.6 *(external action execution) Let* $t,t_1,\ldots,t_n$ *be terms,* x *be a variable, let* ξ,ξ' *be agent environments,* α *be the method name of an external action, and assume functions* F_α *and* G_α *as explained above. The execution of an external action is then defined as follows:*

$$\frac{\gamma \models \kappa}{\langle \iota,\sigma,\gamma,(\texttt{Java(Classname,}\alpha(t_1,\ldots,t_n),x),\kappa),\theta,\xi\rangle \rightarrow \langle \iota,\sigma,\gamma,(\epsilon,\kappa),\theta',\xi'\rangle}$$

where $\theta' = \theta \cup \{x/t\}$ *with* $t = F_\alpha(t_1,\ldots,t_n,\xi)$*, and* $\xi' = G_\alpha(t_1,\ldots,t_n,\xi)$.

Note that the execution of an external action thus influences only the substitution and the environment component of the configuration.

The next type of basic action is the communication action *Send*(r, p, ϕ). We assume that each agent can receive a message at any moment in time. We use then a synchronization mechanism for sending and receiving messages. This synchronization mechanism takes care of simultaneously taking a message from the sending agent and putting it in the belief base of the receiving agent. How these messages are then handled by the receiving agent is done in a completely asynchronous fashion.

The semantics of a *Send*(r, p, ϕ) action affects both sending and receiving agents. The communication action *Send*(r, p, ϕ) is removed from the plan base of the sending agent and the formula *sent*(r, p, ϕ) is added to its belief base. Moreover, the formula *received*(s, p, ϕ) is added to the belief base of the receiving agent, where s is the name of the sending agent. This information about incoming and outgoing messages can respectively be used by the receiving and sending agents for their future deliberations. In order to be able to identify the sending agent when defining the addition of a fact of the form *received*(s, p, ϕ) to the belief base of the receiver, we add the name of the sending agent to messages.

DEFINITION 2.7 (COMMUNICATION ACTION EXECUTION) *Let $\langle s, r, p, \varphi \rangle$ be the format of the message that is sent and received by the agents, where s is the name of the sending agent, r is the name of the receiving agent, p is the communication performative, and ϕ is the message content. The following three transition rules specify the semantics for sending and receiving messages between agents, and their synchronization, respectively.*

- *The transition rule for the sending agent:*

$$\frac{\gamma \models \kappa}{\langle s, \sigma, \gamma, (Send(r, p, \phi), \kappa), \theta, \xi \rangle \xrightarrow{<s,r,p,\phi>!} \langle s, \sigma', \gamma, (\epsilon, \kappa), \theta, \xi \rangle}$$

  where $\sigma' = \sigma \cup \{sent(r, p, \phi)\}$.

- *The transition rule for the receiving agent:*

$$\frac{}{\langle r, \sigma, \gamma, \Pi, \theta, \xi \rangle \xrightarrow{<s,r,p,\phi>?} \langle r, \sigma', \gamma, \Pi, \theta, \xi \rangle}$$

  where $\sigma' = \sigma \cup \{received(s, p, \phi)\}$.

- *The transition rule for synchronization:*

$$\frac{\langle \mathcal{A}_i, \xi \rangle \xrightarrow{\varphi?} \langle \mathcal{A}_i', \xi \rangle \;,\; \langle \mathcal{A}_j, \xi \rangle \xrightarrow{\varphi!} \langle \mathcal{A}_j', \xi \rangle}{\langle \{\mathcal{A}_1, \ldots, \mathcal{A}_i, \ldots, \mathcal{A}_j, \ldots, \mathcal{A}_n\}, \xi \rangle \rightarrow \langle \{\mathcal{A}_1, \ldots, \mathcal{A}_i', \ldots, \mathcal{A}_j', \ldots, \mathcal{A}_n\}, \xi \rangle}$$

Note that the second transition rule guarantees that each agent can receive the messages that are directed to the agent at any moment in time. More discussion on communication between 3APL agents can be found in [53].

Next, we specify the derivation rule for the execution of the test action. A test action can bind the free variables that occur in the test formula for which no bindings have been computed yet.

DEFINITION 2.8 (TEST EXECUTION) *Let β be a well-formed formula and let τ be a ground substitution.*

$$\frac{\sigma \models \beta\theta\tau \ \&\ \gamma \models \kappa}{\langle \iota, \sigma, \gamma, \{(\beta?, \kappa)\}, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma, \{(\epsilon, \kappa)\}, \theta\tau, \xi\rangle}$$

The entailment relation $\models$ in the condition $\sigma \models \beta\theta\tau$ is implemented by the Prolog inference engine. When posing a query β, the substitution θ is first applied to β. The substitution τ is the substitution returned by Prolog and should bind the variables of $\beta\theta$. The entailment relation $\models$ in $\gamma \models \kappa$ is implemented in a similar fashion.

The derivation rules for the execution of composite plans are defined in a standard way.

Next, we define the transition rule for the goal planning rule. A goal planning rule $\kappa \leftarrow \beta \mid \pi$ specifies that the goal κ can be achieved by plan π if β is derivable from the agent's beliefs. A goal planning rule only affects the plan base of the agent.

DEFINITION 2.9 (GOAL PLANNING RULE APPLICATION) *Let $\kappa \leftarrow \beta \mid \pi$ be a goal planning rule. Let also τ_1, τ_2 be ground substitutions.*

$$\frac{\gamma \models \kappa\tau_1 \ \&\ \sigma \models \beta\tau_1\tau_2}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma, \Pi \cup \{(\pi\tau_1\tau_2, \kappa\tau_1)\}, \theta, \xi\rangle}$$

Note that the goal $\kappa\tau_1$ that should be achieved by the plan $\pi\tau_1\tau_2$ is associated with it. It is only this rule that associates goals with plans. The goal base of the agent does not change because the plan $\pi\tau_1\tau_2$ is not executed yet; the goals of agents may change only after execution of plans: goals are removed if believed to be achieved. We do not add substitutions τ_1, τ_2 to θ since these substitutions should only influence the new plan π.

Finally, the transition rule for the goal planning rule that defines reactive behavior, i.e. the goal planning rule in which the head is omitted, is a modification of the above transition rule.

DEFINITION 2.10 (REACTIVE GOAL PLANNING RULE APPLICATION) *Let $\leftarrow \beta \mid \pi$ be a reactive goal planning rule and let also τ be a ground substitution.*

$$\frac{\sigma \models \beta\tau}{\langle \iota, \sigma, \gamma, \Pi, \theta, \xi\rangle \rightarrow \langle \iota, \sigma, \gamma, \Pi \cup \{(\pi\tau, \mathtt{true})\}, \theta, \xi\rangle}$$

Note that the goal associated to the generated plan is set to true, which means that the plan is not generated to achieve a specific goal.

Semantics of a 3APL agent

The semantics of an individual 3APL agent as well as the semantics of a 3APL multi-agent system is derived directly from the transition relation $\rightarrow$. The meaning of individual agents and multi-agent systems consists of a set of so called computation runs.

DEFINITION 2.11 (COMPUTATION RUN) *Given a transition system, a computation run* $\mathtt{CR}(s_0)$ *is a finite or infinite sequence* $s_0, \ldots, s_n$ *or* $s_0, \ldots$ *where* s_i *are configurations, 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 semantics of individual 3APL agents and the semantics of 3APL multi-agent systems.

DEFINITION 2.12 (SEMANTICS OF 3APL MULTI-AGENT SYSTEMS) *The semantics of a 3APL multi-agent system* $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{E} \rangle$ *is the set of computation runs* $\mathtt{CR}(\langle \mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{E} \rangle)$ *of the transition system for 3APL multi-agent systems.*

Note that the computation runs of a 3APL multi-agent system consist of multi-agent transitions which can be derived by means of two multi-agent transition rules. The first is defined in definition 2.4 and the second is the synchronization rule specified in definition 2.7.

3APL Verification

We deem the verification of multi-agent systems very important (cf. [150]). At the moment we do not yet have verification tools for 3APL agents. We have done some theoretical work on agent verification in general [116, 108], and some work more focused on the language 3APL in particular [226]. However, this work is still too theoretical to be the basis of a practical tool. Following related work on the verification of AgentSpeak programs [19] we plan to employ model-checking techniques. At the moment we are investigating if we can check (LTL) temporal properties of agents programmed in a light version of 3APL, using PROMELA, the finite state model specification language for the SPIN LTL model checker [110].

2.2.3 Software Engineering Issues

The 3APL platform and 3APL programming language are designed to respect a number of software engineering and programming principles. Below we give an overview of these principles and how they can be used.

Separation of concerns

Development methodologies for multi-agent systems [234] differ from each other in many respects. Some of them focus on inter-agent aspects, while others also provide support for the design of internal components of an agent, such as mental attitudes and the deliberation process. Finally, some methodologies explicitly deal with the environment, while others do not. The tools to develop and implement multi-agent systems should therefore support each of these issues separately.

The 3APL programming language supports the implementation of inter-agent issues by providing the communication action *Send*, and the 3APL platform manages the transportation of the communicated messages. Moreover, the platform provides information about existing agents to other agents through the Agent Management System (AMS). The information provided by the AMS to agents is required for agents' interactions. The environment of 3APL multi-agent systems can be implemented directly and explicitly through external programs accessible to the agents through API's (application program interfaces).

Finally, the 3APL programming language respects the separation of concerns related to the distinction between an agent's data structures and an agent's operations. In particular, the data structures are mental attitudes such as beliefs, goals, and plans while operations concern manipulation of the mental attitudes such as updating of beliefs, plans and goals, and execution of plans. This distinction is made explicit by introducing two levels of programming: at the data level one can specify the mental attitudes of the agents and at the operation level one can implement the deliberation process of the agent.

Modularity

The implementation of an agent is modular in the sense that an agent can be implemented in terms of seven different modules. The first module is the capability base of the agent which implements the mental actions that an agent can perform to update its beliefs. The second module is the belief base of the agent which contains information the agent believes about the world as well as information that is internal to the agent. The initial beliefs of the agents can be distinguished in two kinds. The first kind of initial beliefs constitutes the background knowledge which can be used by different agents. The second kind of initial beliefs is specific to agents and cannot be used by other agents. Since the background knowledge can be used by different agents, we allow individual agents to load a separate file containing the background knowledge. In this way, one can implement the background knowledge once and allow different agents to load it as part of their initial beliefs. The third module is the goal base that denotes the situation the agent wants to realize. The fourth module is the plan base of the agent which contains the plans that the agent intends to perform. The fifth module is the goal planning rule base that contains the rules that can be used to generate a plan for the possible goals of an agent. The sixth module is the plan revision rule base that contains rules to revise existing agent's plans. Finally, the seventh module is the deliberation module that allows the implementation of an agent's deliberation process.

Abstraction

The abstraction mechanisms that can be exploited in the 3APL programming language are related to external actions and abstract plans. In particular, the external actions allow the 3APL programmers to use external programs through their corresponding API's without having any access to the internal data and operations of the programs. The second abstraction mechanism is related to abstract plans which allow users to abstract over certain parts of plans. The abstract plans can be instantiated with a plan through the application of plan revision rules. It is very important to note that an abstract plan should be introduced, not only because it occurs in different plans, but also because its specific instantiation depends on the conditions known only at run time. For example, going to work can be considered an abstract plan since its specific instantiations such as going to work by bus, by taxi, by train, or by own car depend on the conditions that hold when the plan is to be executed. For example, if the agent does not have enough money, then it may consider going by bus or train, otherwise it may consider using a taxi.

The introduction of abstract plans in 3APL implies the introduction of plan revision rules. In implementing 3APL agents, the programmers tend to conceive abstract plans as a kind of procedure calls and the plan revision rules as the corresponding procedure. It is important to note that this is not the optimal and principal use of abstract plans and their corresponding plan revision rules.

Reusability

Finally, the 3APL platform allows reusing multi-agent systems by providing a library of templates for individual agents and templates for multi-agent systems. Using the templates for individual agents, the 3APL programmer can use generic agents that have certain initial mental attitudes. The templates for multi-agent systems, also known as projects, allow the 3APL programmers to use a set of generic agents that, in addition to their initial mental attitudes, follow a specified interaction protocol. Such a template can include an environment with which the agents are supposed to interact. An example of a multi-agent template is a template for an auction. In order to implement such an auction, a 3APL programmer can load such a multi-agent template and implement both the details of the agents, such as their specific initial mental attitudes, as well as the details of their environment.

2.2.4 Language integration

The 3APL programming language together with its platform allows the integration of Prolog and Java. The Prolog programs can be integrated since they can be loaded in 3APL and used as background knowledge. Given a loaded Prolog program, the agent can pose queries in three different contexts: as the pre-condition of mental actions, as test actions in plans, and as the guard of the reasoning rules. The Prolog programs can thus be used to control the execution of mental actions, the execution of plans, and the application of reasoning rules. Note that the queries may yield substitutions that can bind other variables used in the post-conditions of the mental actions, in the rest of plans that follow a test action, and in the bodies of reasoning rules.

Moreover, the 3APL programming language allows Java programs to be used through external actions. The external actions can be used to call methods of Java classes. Using the arguments of these methods, it is possible to pass data from 3APL to Java and vice versa. In this way, data can be passed from Java to the plans of the agent to the Prolog part (belief base) of the agent and vice versa. Note that the integration of Java is also used to implement the multi-agent environment with which the agents interact.

2.3 Platform

2.3.1 Available tools and documentation

The 3APL platform is an experimental tool, designed to support the development, implementation, and execution of 3APL agents [54]. The detailed information about installation and deployment of the 3APL platform can be found in the 3APL user guide which is available online at the following URL:

`http://www.cs.uu.nl/3apl/download/java/userguide.pdf`

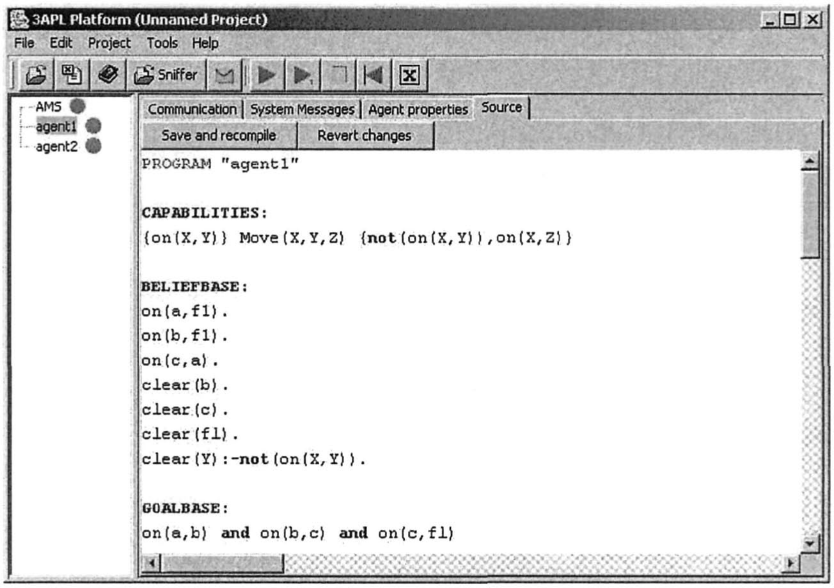
or in [217]. Moreover, we are developing a tutorial and training material which will be available soon from the 3APL web page:

`http://www.cs.uu.nl/3apl`

Also, various papers on 3APL can help to understand how to deploy the 3APL platform [107, 228, 54, 52, 227]. Finally, the implementation documentation of the platform can be found at:

`http://www.cs.uu.nl/3apl/docs/aplp-refman/index.html`

The 3APL platform provides a graphical interface, as shown in Figure 2.4, through which a user can develop and execute 3APL agents using several facilities, such as a syntax-colored editor and several debugging tools. The platform allows communication among agents and provides the Agent Management System (AMS) that is responsible for registration of the hosted agents. Multiple 3APL platforms can run on different machines connected in a network at the same time, such that agents hosted on these platforms can communicate with each other. When the 3APL platform is started, the user should select whether the multi-agent application is intended to act as a server or as a client. The server option must be selected the first time the 3APL platform is run. The client option can be selected only if the 3APL platform is running as a server already. When the user selects the client option, the IP of the server with which the (client) platform should connect, must be filled in.



```
PROGRAM "agent1"

CAPABILITIES:
{on(X,Y)} Move(X,Y,Z) {not(on(X,Y)),on(X,Z)}

BELIEFBASE:
on(a,fl).
on(b,fl).
on(c,a).
clear(b).
clear(c).
clear(fl).
clear(Y):-not(on(X,Y)).

GOALBASE:
on(a,b) and on(b,c) and on(c,fl)
```

Figure 2.4. An illustration of the graphical user interface of the 3APL platform.

The graphical interface shows in the left side window the names of the agents that are hosted and running on the platform in a tree-like structure. The tree includes also the AMS (Agent Management System) which is modelled as a non-programmable agent that provides information about hosted agents to each of the running agent. The information will be provided only on request. The same window of the graphical interface presents also the status of the hosted agents such as initial, running, stopped, final, and erroneous. Moreover, the `Communication` tab of the graphical interface provides a message window that displays the messages that are exchanged between agents. The `System Messages` tab is a window that shows the system messages such as parse errors or the errors that are generated during the execution. The `Agent properties` tab is a window that can be used to monitor the (mental) states of the agents during their execution. The `Source` tab provides an editor that allows programmers to modify the initial mental state of agents. In addition, the interface provides a sniffer button that displays the graphical representation of the message exchange.

2.3.2 Standards compliance, interoperability and portability

The 3APL platform has been tested on Windows 98, Windows NT and Windows XP, as well as on Linux, Unix (Solaris) and Mac OS X. 3APL is written in Java 2 SDK 1.4, and makes use of the Prolog engine of JIProlog, which is also implemented in Java. We have tested it for Java 2 SDK 1.4.0_02 and upwards. The downloadable 3APL package consists of a .jar file that contains all the .class files needed, as well as examples of 3APL programs. The package needs approximately 800 KB.

The 3APL platform adheres to the FIPA standard to the extent that it provides a simplified version of an Agent Management System which provides a combination of name service and yellow-page services. Moreover, the format of the messages that are communicated between 3APL agents are based on FIPA standards, consisting of the identifiers of the sender and receiver of the message, the performative or speech act, and the content of the message. The 3APL platform supports only the development, implementation, and execution of multi-agent systems that consist of 3APL agents. At this moment, the platform does not support open multi-agent systems, mobile agents, or heterogeneous agents.

The 3APL platform is still in a prototyping stage and can execute only a small number of agents. The performance of the platform decreases if the number of agents, which are loaded and executed concurrently on the platform, grows. One reason for the low performance is the complex and cognitive nature of agents and the fact that agents have the capability to reason with their mental attitudes. The platform can handle the messages that are exchanged by the agents, although the number of agents that can be run efficiently on the 3APL platform is small.

The platform provides distributed control such that the agents can be executed concurrently. This enables loading, executing, and stopping agents while other agents are running. The platform also provides the possibility to build a library of agents, multi-agent systems and agent templates. The templates can be loaded and extended to build multi-agent systems. Finally, based on the templates it is possible to have interaction protocols in the platform's library, since the protocols can be defined in terms of a set of agent templates in which only the actions prescribed by the protocols are specified.

2.4 Applications supported by the language and/or the platform

The applications that can be developed using the 3APL platform and the 3APL programming language are those that are best understood in terms of cognitive and social concepts like beliefs, goals, plans, actions, norms, organizational structures, resources and services that are part of the multi-agent environment. We have already implemented a number of toy problem applications such as block world logistics, Axelrod's tournament, English Auction, and Contract Net protocols. Also, 3APL is already applied to implement the high-level control of mobile robots. In this project, external actions of 3APL were defined and connected to some simple sensory and motor actions of the mobile robot. In this way, a programmer can implement a 3APL program that senses the position of the robot it is controlling and determine how to reach a goal position in a rectangular environment, a model of which is accessible to the 3APL program. Currently, 3APL is also being applied to control the behavior of SONY AIBO robots and to implement small device mobile applications.

2.5 Final Remarks

The 3APL platform can be employed to implement multi-agent systems where each individual agent is implemented through the 3APL programming language. Using the 3APL programming language, individual agents can directly be implemented in terms of cognitive concepts such as beliefs, goals, plans, actions, and reasoning rules. Experience from deploying the 3APL platform for educational purposes have proved it to provide appropriate programming constructs for direct and easy implementation of applications that are analyzed and designed by existing multi-agent system development methodologies such as Prometheus [163] and Gaia [242].

The programming language 3APL is subject to constant theoretical and practical improvements. For example, the definition of the 3APL language is extended with specific programming constructs to implement the agent's deliberation process, declarative goals, other types of reasoning rules such as goal planning rules, and external and communication actions. Also, the specification of belief is distinguished from the belief query expressions. The practical development consists of the implementation of the 3APL platform that allows the design, implementation, and testing of multi-agent applications. Facilities provided by the platform ease the task of developing multi-agent systems.

Currently, we are working to extend and refine the implementation of the 3APL platform by adding additional features needed to facilitate the development of multi-agent systems. One of the extensions is to provide programming constructs for adopting different types of goals such as achievement goals, perform goals and maintenance goals at run time. The extension will add basic actions dedicated for adopting different types of goals such that executing plans that include these types of basic actions generates goals. Another extensions is to provide programming constructs to allow explicit implementation of the organizational structures and the multi-agent environment. In particular, we are building on the existing coordination mechanisms designed for concurrent component-based systems and extend them with social and organizational concepts needed to specify multi-agent organizations. Moreover, we aim at using the existing web technologies such as XML and web services to define the environment of multi-agent systems. Our aim is that any introduced extension and refinement should have a theoretical foundation, being defined in terms of formal syntax and semantics.

Acknowledgments

Thanks to Frank de Boer and Frank Dignum for discussion on the issues raised in this paper.