# Formal Semantics for an Abstract Agent Programming Language

K.V. Hindriks, F.S. de Boer, W. van der Hoek, J.-J.Ch. Meyer

Universiteit Utrecht, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
tel. +31-30-2539267
{koenh,frankb,wiebe,jj}@cs.ruu.nl

**Abstract.** In this article we investigate agent-oriented programming both from a theoretical and a practical view. We propose an abstract agent programming language with a clear and formally defined semantics. The semantics of our language is defined in terms of a transition system. Our language combines features of both logic programming, i.e. it is rule-based, and imperative programming, i.e. it includes the full range of convential programming constructs. These features are well-understood and provide a solid basis for a structured agent programming language. On the more practical side we investigate the different properties of agents usually attributed to them, i.e. agents have a complex mental state, act pro-actively and reactively, and have reflective capabilites. We illustrate how these properties of agents are implemented in our programming language. In particular, we propose general rules, called practical reasoning rules, which are used to define the reflective capabilities of agents and provide a mechanism for goal revision. The combination of the theoretical and more practical perspective contributes, we hope, to filling the gap between theory and practice.

## 1 Introduction

Research on intelligent agents ranges from theoretical, logical investigations to more practical, implemented applications. The issues that are being solved on each side of the spectrum, in general, are very different. On the logical side, much research goes into deriving formal properties of logical systems, like completeness or decidability ([13,15,17]). On the other side, architectures, issues like scheduling, and other design and implementation issues are investigated ([8]).

Quite a number of researchers have made proposals for agent programming languages or agent design frameworks, e.g. [2,10–12,14]. Some of these languages, however, lack a clear and formally defined semantics, and therefore it is difficult to formalize the design, specification and verification of programs. Other types of languages are based directly on logic ([4,5,3,18]).

In contrast to these existing approaches, our approach is based on the combination of existing programming conceptss from various known paradigms in order to model agent-oriented features. The clear advantage of this approach is that these concepts are well-understood, both from a theoretical and practical perspective.

More precisely, the programming language we propose is based on a combination of features of logic programming and imperative programming, which allows for an elegant description of many agent-oriented features. In particular, concepts from imperative programming are used to describe the *execution* of the goals of an agent. Whereas the *reflective* capabilities of an agent, on the other hand, are described by a rule-based reasoning which is driven by the pattern-matching from logic programming. The operational semantics of the agent programming language is defined in terms of a transition system. This is a simple and lucid formalism for specifying the meaning of programs.

## 2 Programming BDI-agents

In the DAI community the concept of BDI-agents is nowadays a well-known notion. A lot of theory and practical investigation has gone into making this notion both better understood and applicable to real-world problems. The design and programming of BDI-agents, however, is still somewhat disconnected from theory. One of our aims is to construct a more unifying language which covers a relatively large part of the current research into programming agents. We thereby hope to clarify also part of the essence of AOP. The

programming language we propose captures the main features of both AGENT-0 ([12]) and AgentSpeak(L) ([11]). Although, at first sight, these languages seem very different in nature, they incorporate a few basic principles which are unified in the language we propose. The basic differences reside in the particular *control structure* chosen in each of these languages from which we abstract.

An important question which must be answered first is which features a general agent programming language should include. Most researchers seem to agree on the following characteristics of BDI-agents:

- agents have a complex internal mental state which is made up of beliefs, desires, plans, and intentions, and which may change over time;
- agents act pro-actively, i.e. goal-directed, and reactively, i.e. respond to changes in their environment in a timely manner;
- agents have reflective or meta-level reasoning capabilities.

From this list we derive that intelligent agents are *goal-directed*, *belief-transforming* entities, i.e. agents have a set of goals which guide their behaviour and keep up with the world by maintaining a belief base with information about the (current) situation they find themselves in. To realise their goals, agents need to find the means for their ends, i.e. agents try to realise their goals by *means-end reasoning* or *practical reasoning*. Plans are recipes for achieving the goals of the agent, and therefore play a crucial role in this reasoning. A reflective agent also needs to have the ability to *monitor* its success or failure, and the reasoning mechanisms for responding to failure.

A minimal requirement for an agent programming language, therefore, is that it has features for dealing with these different notions. At the least, this means such a language must have features for:

- *belief updating*, for dealing with newly observed, communicated data, and the derivation of data,
- *goal updating*, for goal revision in the light of (new) information and requests of other agents, and
- *practical reasoning*, for finding the means to achieve a goal.

A more practical concern is related to the choice of programming constructs. In our language we include all the familiar constructs from imperative programming like sequential composition, tests, parallel execution, etc. (cf. also [6]).

## 3 An Abstract Agent Programming Language

We now define an agent language corresponding to the three components outlined above: belief updating, goal updating, and practical reasoning. First, we define a first-order language in which to express the beliefs of agents. Next we define a goal language; goals are taken to be programs. The agent's practical reasoning capabilities are defined by practical reasoning rules. Thus, our language is a rule-based language.

### 3.1 Beliefs

We do not say much about the beliefs of agents here. The beliefs of an agent are simply first-order formulae from a language $\mathcal{L}$.

The basic elements of the language are given by a *signature* $\Sigma$, which is a quadruple $\langle \mathsf{P}, \mathsf{F}, \mathsf{C}, \mathsf{A} \rangle$ where $\mathsf{P}$ is a set of *predicate symbols*, $\mathsf{F}$ is a set of *function symbols*, $\mathsf{C}$ is a set of *constants*, and $\mathsf{A}$ is a set of *action symbols*. *Terms* and *first-order formulae* of a language $\mathcal{L}$ are defined as usual from a signature $\Sigma$ and an infinite set of variables $\mathsf{TVar}$. (The action symbols are not used in this definition.) The set of *atoms* is denoted by $\mathsf{At}$.

*Example 1. (robot Greedy)*
The running example in this paper concerns a robot called Greedy. Imagine robot Greedy wandering around in a two-dimensional grid world. In this fictitious world, diamonds pop up and disappear at random. The only obstacles in the world are rocks. Robot Greedy cannot move rocks and has to go around them. The basic predicates which describe this world are: `diam(d,x,y)` for 'diamond d is at coordinates x,y'; `robot(r,x,y)` for 'robot r is at coordinates x,y'; `rock(x,y)` for 'there is a rock at x,y'.

Greedy uses two functions to calculate the coordinates of one of the nearest diamonds in its neighbourhood: `xc(x,y)` for the x-coordinate and `yc(x,y)` for the y-coordinate of a diamond with minimal distance from x,y.

For this example, we assume that Greedy has perfect knowledge of its environment, i.e. it knows at any moment its own position, and the location of the rocks and diamonds.

### 3.2 Goals and actions

The goals of an agent set objectives the agent tries to achieve. We distinguish two kinds of goals: goals to do some action and goals to achieve some state of affairs. We also allow different kinds of composition of these types of goals.

**Definition 2.** *(basic actions, goal language)*
Let $\Sigma = \langle \mathsf{P}, \mathsf{F}, \mathsf{C}, \mathsf{A} \rangle$ be a signature, and $\mathsf{Gvar}$ an infinite set of goal variables ranging over goals.

- The set of *goals* $\mathcal{L}^g$ is inductively defined by:
  - [Syn-5] $\mathsf{A} \subseteq \mathcal{L}^g$, called *basic actions*,
  - [Syn-6] $\mathsf{At} \subseteq \mathcal{L}^g$,
  - [Syn-7] If $\varphi \in \mathcal{L}$, then $\varphi? \in \mathcal{L}^g$,
  - [Syn-8] $\mathsf{Gvar} \subseteq \mathcal{L}^g$,
  - [Syn-9] If $\pi_1, \pi_2 \in \mathcal{L}^g$, then $\pi_1; \pi_2, \pi_1 + \pi_2, \pi_1 \| \pi_2 \in \mathcal{L}^g$.

Basic actions $a \in \mathsf{A}$, achievement goals $P(\mathbf{t}) \in \mathsf{At}$, and test goals $\varphi?$ are the basic goals in our language. Basic actions in our language are *update actions* or *update operators* on the belief base of an agent. For example, an action `pickup(Greedy,d)` deletes from the belief base of Greedy the belief that there is a diamond at the current position of Greedy. Achievement goals $P(\mathbf{t})$ are goals to achieve a state where $P(\mathbf{t})$ holds. Test goals $\varphi?$ are checks performed on the belief base to see if a proposition $\varphi$ holds.

It is also possible to build more complex compositions of basic goals, by using the program constructs for sequential composition, nondeterministic choice, and parallel composition. This allows us to specify disjunctive goals $\pi_1 + \pi_2$ by means of nondeterministic choice, i.e. do either $\pi_1$ or $\pi_2$, and conjunctive goals $\pi_1 \| \pi_2$ by means of parallel composition, i.e. do both $\pi_1$ and $\pi_2$.

Furthermore, the language includes variables which range over goals. These variables serve several purposes. First of all, they can be used for reflective reasoning. This will become clear in the next paragraph, where these variables are also allowed in the head of practical reasoning rules. Secondly, they might be used in communication. For example, an agent might receive in a goal variable a request to establish some goal.

*Example 3.* The robot Greedy is capable of performing the following basic actions: `west` for moving west, `east` for moving east, `north` for moving north, `south` for moving south, and `pickup(r,d)` for 'robot r picks up diamond d'.

The *goal* of robot Greedy is to collect as much diamonds as possible. This goal is denoted by the predicate `max_diam`, which is a user-defined predicate and is in fact given below as a (usual) procedure definition.

*Remark 4.* A note on terminology:
We speak about goals, and use this term rather than the term 'intention', since we think that a goal in the goal base of an agent does reflect a choice an agent has made, but does not specify the level of commitment made to that goal. Intentions are sometimes defined in the literature as choice + commitment ([1]). Since the commitment made to a goal remains implicit in the practical reasoning rules, we use the more neutral term 'goal' for the proattitudes of agents. The practical reasoning rules determine how easily an agent will revise its goals, and thereby the level of commitment made to a goal.

### 3.3 Practical reasoning rules

To achieve its goals an agent has to find the means for achieving them, and sometimes may have to revise its goals. This kind of reasoning is called *practical reasoning*. To perform this type of reasoning an agent uses a set of practical reasoning rules:

**Definition 5.** *(rule language)*
The set of *practical reasoning rules* $\mathcal{L}^p$ is defined by:

[Syn-10] If $\varphi \in \mathcal{L}$, $\pi, \pi' \in \mathcal{L}^g$, then $\pi \leftarrow \varphi | \pi' \in \mathcal{L}^p$.

A practical reasoning rule has the following components:

**Definition 6.** *(head, body, guard, global and local variables)*
Let $\pi \leftarrow \varphi | \pi'$ be a practical reasoning rule.

- $\pi$ is called the *head* of the rule,
- $\pi'$ is called the *body* of the rule,
- $\varphi$ is called the *guard* of the rule,
- the free variables in the head of a rule are called the *global variables* of the rule,
- all variables in a rule which are not global are called *local variables*.

The guard of a rule serves two functions. First, the guard can be used to specify the context in which the rule might be applied. For example, Greedy might want to pick up a diamond in case there is a diamond at its feet. Secondly, the guard can be used to retrieve data from the set of beliefs. This is done by unifying some of the parameters in the guard and head with beliefs in the belief base (see below). For example, the current position of Greedy needs to be retrieved for deciding which way to go.

The distinction between global and local variables is made to separate the specific local data-processing in the body which uses a local set of variables from the global variables which are also used in other parts of the program (i.e. goal) and which can be used to 'communicate with the rest of the goal' (return values, etc.).

A practical reasoning rule also serves two functions.

First, a rule can specify the *means* to achieve a particular goal. In this case the head of the rule is a basic goal $P(\mathbf{t})$ and the body of the rule is a *plan* or *recipe* for achieving the goal. So, achievement goals $P(\mathbf{t})$ may be viewed as procedure calls to plans specifying the means to achieve the goal. A plan is a program which may contain parallel actions and nondeterministic choice. These type of rules are also called *plan rules*. A plan rule encodes the *procedural knowledge* of an agent.

*Example 7.* Greedy wants to maximize the number of diamonds collected. A plan rule which specifies how to achieve this goal is the following:

```
max_diam <- robot(Greedy,x0,y0)/\x=xc(x0,y0)/\y=yc(x0,y0)|
robot(Greedy,x,y);diam(z,x,y)?;pickup(Greedy,z);max_diam
```

This rule specifies a plan to achieve `max_diam`. The plan is to go to the nearest place where a diamond can be found, pick up that diamond, and then again maximize the number of diamonds collected. The guard retrieves the current position of Greedy.

A plan for getting Greedy at a specific position is the following:

```
robot(r,x,y) <- robot(r,x0,y0) | (x=x0/\y=y0)? +
[(x<x0)?;west +(x0<x)?;east +(y<y0)?;south +(y0<y)?;north];robot(r,x,y)
```

The plan (i.e. body) states that to get at position x,y one should repeat making a move in the right direction until the position is reached (x=x0,y=y0). The rule also illustrates how to program while or repeat-until loops.

This rule shows the two uses of predicates in our language. The predicate `robot(r,x,y)` in the head denotes a possible (sub)*goal* of the agent, while the predicate `robot(r,x0,y0)` in the guard denotes a test on the belief base. By using a predicate both in the head (as a goal) and in the guard (as a belief) in a practical reasoning rule an interface between the belief and goal bases is established. Thus, predicates in goals are not just procedurally defined, but can be related to their logical interpretation as beliefs via practical reasoning rules.

The second purpose practical reasoning rules can be used for is the *revision of goals*. In this case practical reasoning rules encode the *reflective capabilities* of an agent, and arbitrary programs (including goal variables) may be used as the head of a rule. Basically, there are two types of situations in which a rational agent might wish to revise one of his goals, namely, in case a more optimal strategy can be followed, or in case of failure. We show an instance of each of these cases of goal revision, also illustrating the use of goal variables.

*Example 8.* Assume that Greedy has a goal to go west to collect a diamond $d$, but suddenly a new diamond $d'$ pops up nearer to his current position. In this unpredictable fictitious world where diamonds may pop up and disappear at random a more optimal strategy for Greedy probably is to revise its current goal and go after the new diamond $d'$. The following rule makes this type of goal-revision possible:

```
X;robot(r,x,y) <- robot(r,x0,y0)/\not(x=xc(x0,y0)/\y=yc(x0,y0)) |
robot(r,xc(x0,y0),yc(x0,y0))
```

The rule applies when robot r has a goal of doing some (possibly empty) list of actions X, for example going west, and after that get at position x,y, but at the same time there also is a diamond nearer to robot r's current position than the target-position x,y. In that case the rule makes it possible to revise the old goal by replacing it with a new goal of going to the nearest diamond.

In another situation, where Greedy still has the goal of going west, Greedy might encounter another problem. Assume this time that a rock is in the way and Greedy cannot move west. So, if Greedy tries to move west, it fails. Since Greedy is assumed to have perfect knowledge Greedy immediately detects the failure and knows that it is still at its old position. To avoid such failure Greedy might use the following rule:

```
west;robot(r,x,y) <- robot(r,x0,y0)/\rock(x0-1,y0) |
[(y0<=y)?;north + (y0>=y)?;south];robot(r,x,y)
```

This rule detects the failure and makes it possible to revise the goal and employ a strategy to go around the rock. (Not a very adequate one.)

Summarizing, our approach distinguishes three levels of agent programming. At the most basic level, basic actions act upon the belief bases of agents. At the second level, goals are executed. Note that goals *are* somewhat generalized programs, i.e. goals are programs built from the regular programming constructs *and* goal variables. The goal base of an agent in fact represents the program an agent is executing at the moment. Agents, however, also have reflective capabilities for revising their goals. Therefore, agents are not just programs (goals), but are *self-modifying* programs. This concept of self-modification is built into the practical reasoning rules, the third level of our agent programming language. By means of practical reasoning the agent can transform a program to another program (as in the examples above), not just by execution or procedure calls, but by more general revision rules (i.e. the practical reasoning rules with general head). This is a distinguishing feature of agents.

*Substitutions and variables* We need a number of technical definitions in order to proceed. The *scope of a quantifier*, a *bound occurrence* of a variable $x$ in an expression $e$, i.e. a term, formula, or program, and a *free variable* are defined in the usual way. The set of free variables in an expression $e$ is denoted by $\mathsf{Free}(e)$. The notions of *variable substitution*, *unifier*, *most general unifier*, and *variant* are defined as usual (cf. [7]).

## 4 Agent programs

Agents are goal-directed, belief-transforming entities, which are capable of means-end reasoning and have reflective capabilities. We assume that beliefs are updated by basic actions, and goals are updated by execution and revision. This cleanly seperates the two different types of updating. The practical reasoning component is encoded in practical reasoning rules.

In this picture, the only components of an agent that change are its beliefs and goals (thus, we assume the set of practical reasoning rules and basic actions are fixed). Beliefs and goals make up the mental state of an agent. Therefore, we define:

**Definition 9.** *(mental state)*
A *mental state* is a pair $\langle \Pi, \sigma \rangle$, where

- $\Pi \subseteq \mathcal{L}^g$ is a *goal base*, i.e. a set of goals, and
- $\sigma \subseteq \mathcal{L}$ is a *belief base*, i.e. a set of beliefs.

We use $\Pi$ to denote a goal base, and $\sigma$ (possibly indexed) to denote a belief base. The set of belief bases is denoted by $\mathcal{B}$. We use $\Gamma$ to denote a PR-base, i.e. a set of practical reasoning rules.

The dynamics or behaviour of an agent, therefore, is fully specified if the semantics of basic actions is given and the mechanisms for executing goals and applying rules are defined. We assume that a transition function for basic actions is given. Basic actions define the basic skills of agents. The treatment of the mechanisms for executing goals is postponed to the next section.

Since basic actions only change belief bases, and have no effect on the goal base or PR-base, the effect of basic actions is fully specified by giving their effect on belief bases.

**Definition 10.** *(basic action transitions)*
The semantics of basic actions A is given by a transition function $\mathcal{T}$ of type : $\mathcal{B} \times \mathcal{B} \rightarrow \wp(\mathsf{A})$. Let $a \in \mathsf{A}$. We use the following notational convention, and write: $\langle \sigma, \sigma' \rangle a$ for $a \in \mathcal{T}(\sigma, \sigma')$.

*Example 11.* Robot Greedy is capable of performing five basic actions. The operational semantics of these actions is given by:

- $\langle \{\ldots, \texttt{robot(Greedy,n,m)}, \texttt{not(rock(n-1,m)}, \ldots\},$
  $\{\ldots, \texttt{robot(Greedy,n-1,m)}, \texttt{not(rock(n-1,m)}, \ldots\} \rangle \texttt{west};$
  similar definitions apply to actions `east`, `north`, and `south`.
- $\langle \{\ldots, \texttt{diam(d,n,m)}, \texttt{robot(Greedy,n,m)}, \ldots\},$
  $\{\ldots, \texttt{robot(Greedy,n,m)}, \texttt{not(diam(d,n,m)}, \ldots\} \rangle \texttt{pickup(Greedy,d)},$

The intended reading of these definitions is that, for example, action `west` can be performed if Greedy believes it is at coordinates n,m, and afterwards its beliefs have changed such that Greedy believes he is at n-1,m after performing the action `west`.

Note that in the case there is a rock blocking the way to the west but Greedy doesn't *believe* there is a rock to the west, although the `west` action will fail in the real world Greedy will still *believe* he has made a move to the west. Only by observing the environment Greedy will notice that his action of moving west has failed. Similar remarks apply to the other actions. (The issue of failure is not discussed any further in this paper, however, since we assumed perfect knowledge this type of failure cannot occur.)

Now, to program an agent is to specify its initial mental state, the semantics of the basic actions the agent can perform, and to write a set of practical reasoning rules. This is formally stated in the next definition.

**Definition 12.** *(agent program)*
An *agent program* is a quadruple $\langle \mathcal{T}, \Pi_0, \sigma_0, \Gamma \rangle$ where

- $\mathcal{T}$ is a basic action transition function, specifying the effect of basic actions,
- $\Pi_0$ is the initial goal base,
- $\sigma_0$ is the initial belief base, and
- $\Gamma$ is a PR-base.

*Example 13.* The agent program for Greedy is the following:

- $\mathcal{T}$ is defined for the basic actions in example 11,
- the initial goal base is given by: $\{\texttt{max_diam}\}$,
- the initial belief base is given by:
  $\{\texttt{robot(Greedy,0,0)}, \texttt{rock(1,5)}, \texttt{rock(3,3)}, \texttt{rock(2,1)}, \texttt{diam(d,2,2)}\}$.
- the PR-base contains the PR-rules as defined in examples 7 and 8. We add two other rules, to illustrate two other features:

- If at the target-position, Greedy should pick up a diamond. However, since diamonds may disappear at random, the diamond may not be there anymore. Greedy tests before picking up a diamond if there is one; in case there is none, however, Greedy will fail the test, and the goal becomes infeasible. A rule for dropping such a goal is an example of the first feature which is that we allow an *empty body* in rules: `diam(d,x,y)?;pickup(r,z) <- not(diam(d,x,y))`
- In case Greedy happens to be at a position of a diamond, Greedy should pick it up, regardless of what other goals Greedy has at that moment. This can be achieved by the following rule, an example of the second feature which is that we also allow *empty heads* in rules: `<- robot(r,x,y)/\diam(d,x,y) | pickup(r,d)`. We call this type of rules *data-directed*.

Data-directed rules create new goals on the basis of beliefs only. These rules are a distinguishing feature of agents, although they do resemble interrupts. Data-directed rules could be used to make agents reactive to their environment (like interrupts), however, the level of reactiveness accomplished also highly depends on the underlying control structure of the language.

## 5 Operational semantics

We define the operational semantics of our language in this section. First, we explain what a transition system is. Next we give the transition rules which define the semantics.

### 5.1 Transition system

Transition systems can be used to define the operational semantics of programming languages ([9]). For example, specifying the operational semantics of an imperative programming language means specifying how a program can transform the state of a system. This is done by specifying how an assignment changes the state, and by giving rules for all the other programming constructs.

Formally, a transition system is a deductive system which allows to *derive* the transitions of a program. A transition system consists of a set of *transition rules* that specify the meaning of each programming construct in the language. Transition rules transform *configurations*. In imperative programming a configuration is a pair $\langle \pi, \sigma \rangle$ with $\pi$ a program and $\sigma$ a variable assignment. In our framework a configuration is the same as a mental state, i.e. a pair $\langle \Pi, \sigma \rangle$ of goals and beliefs. Thus, a transition rule maps a configuration to another configuration by executing a goal or matching the head of a practical reasoning rule with a goal. For example, there is a rule which fixes the meaning of sequential composition, a rule for parallel composition, etc. A single transition models the execution of a simple instruction (assignment in the case of imperative programming, basic actions and tests in our case).

### 5.2 Practical reasoning rule

In the rest of this paragraph and the next, let $\sigma, \sigma'$ be belief bases, $\theta$ a variable substitution, and $V$ a set of variables. In the examples, we have already seen a number of practical reasoning rules. In the following definition the application of these rules is formally defined.

**Definition 14.** *(PR-rule application)*
Let $\theta$ be a most general unifier for $\pi$ and $\pi'$, and $\gamma$ a substitution.

$$\frac{\pi' \leftarrow \varphi | \pi'' \in' \Gamma \text{ and } \sigma \models \forall(\varphi\theta\gamma)}{\langle \pi, \sigma \rangle_V \longrightarrow_{\theta\gamma} \langle \pi''\theta\gamma, \sigma \rangle} \quad \text{where}$$

- $\pi' \leftarrow \varphi | \pi'' \in' \Gamma$ means that $\pi' \leftarrow \varphi | \pi''$ is a variant of a PR-rule in $\Gamma$ with fresh variables not occurring in $V$, and
- $\gamma$ is a substitution such that for no variable $x$: $\gamma(x) \in V$

Pattern-matching of logic programming is used to unify the head of a practical reasoning rule with a goal in the goal base. First, the substitution $\theta$ unifies the head of the practical reasoning rule with a goal in the goal base to get the parameter values which are part of the agent's current goal to instantiate (some of) the global variables. Next, $\gamma$ retrieves parameter values from the current belief base. This explains the order of application of the substitutions $\theta$ and $\gamma$. A variant of the plan is used to avoid interference between local and global variables. In the set $V$ the global variables of the goal base are recorded. Local variables are chosen such that no interference occurs. The additional information (the substitution $\theta\gamma$) attached to the transition-relation ($\longrightarrow$) is used below in the execution rules.

*Example 15.* Suppose that the goal base is {`west;robot(Greedy,3,4)`} and the belief base is the initial belief base from example 13. In this case, the revision rule of example 8 is applicable, since Greedy's current goal is not leading to any diamond at all.

The most general unifier $\theta = \{X/$`west`$,r/$`Greedy`$,x/3,y/4\}$. Using this substitution the guard becomes:
`robot(Greedy,x0,y0)/\not(3=xc(x0,y0)/\4=yc(x0,y0))`.
Matching this with the belief base yields a substitution $\gamma = \{x0/0,y0/0\}$. Now we can substitute the original goal with the new goal, using substitution $\theta\gamma$, which is:
`robot(Greedy,xc(0,0),yc(0,0))=robot(Greedy,2,2)`.

It is important that the local variables in the body of a rule are changed to new variables not yet occurring in the goal base (recorded in the set $V$) to avoid undesired communication between subgoals. For example, in a goal
`robot(r,n,m,2);robot(r,x0,y0)?;robot(r,x0+1,y0)`
the global variables $x0$ and $y0$ are the same as the local variables of the practical reasoning rule for `robot(r,x,y)` for getting at a specific position (cf. example 8). Without renaming the local variables the application of the practical reasoning rule leads to undesired parameter passing. This explains the conditions in the definition.

### 5.3 Execution rules

In this paragraph we define the meaning of the basic actions and conventional programming constructs. We use $E$ to denote termination, and identify $E;\pi$ with $\pi$, $E+\pi$ with $\pi$, etc. Specifying the meaning of a basic action $a$ is simple, since we have assumed that a transition function $\mathcal{T}$ is given for these actions. Executing a basic action means changing the state according to the transition function and after that stop execution. The identity substitution $\emptyset$ is attached to the transition for basic actions.

**Definition 16.** *(execution rule 1; basic actions)*

$$\frac{\langle \sigma, \sigma' \rangle a}{\langle a, \sigma \rangle_V \longrightarrow_\emptyset \langle E, \sigma' \rangle}$$

Tests are actions to check if some condition follows from the belief base. Free variables in tests may be used to retrieve data from the belief base. The values retrieved are recorded in a substitution $\theta$; this substitution is a parameter of the transition, which can be applied in other transitions again.

**Definition 17.** *(execution rule 2; first-order tests)*

$$\frac{\sigma \models \forall(\varphi\theta)}{\langle \varphi?, \sigma \rangle_V \longrightarrow_\theta \langle E, \sigma \rangle}$$

For example, the test `diam(z,x,y)?` executed in the initial state of example 13 yields the substitution $\theta = \{z/$`d`$,x/2,y/2\}$.

These values can be passed to the rest of the goal. For example, by executing a sequential goal `diam(z,x,y)?;pickup(Greedy,z` first, the test is executed, and then the values are passed onto the action `pickup(Greedy,z)` by application of the substitutions to the remaining goal. More formally, this is stated in the next rule for sequential composition:

**Definition 18.** *(execution rule 3; sequential composition)*

$$\frac{\langle \pi_1, \sigma \rangle_V \longrightarrow_\theta \langle \pi_1', \sigma' \rangle}{\langle \pi_1; \pi_2, \sigma \rangle_V \longrightarrow_\theta \langle \pi_1'; \pi_2\theta, \sigma' \rangle}$$

Applying the rule to the example yields a new mental state with action component `pickup(Greedy,z)`$\{z/\mathtt{d}, x/2, y/2\}$`=pickup(Greedy,d)`.

To execute a nondeterministic choice goal, just pick one of the subgoals that is enabled, i.e. can be executed, and execute this goal.

**Definition 19.** *(execution rule 4; non-determinimistic choice)*

$$\frac{\langle \pi_1, \sigma \rangle_V \longrightarrow_\theta \langle \pi_1', \sigma' \rangle}{\langle \pi_1 + \pi_2, \sigma \rangle_V \longrightarrow_\theta \langle \pi_1', \sigma' \rangle}$$

(A similar rule can be given for right choice.)

For example, `(2<0)?;west+ (0<2)?;east+ (2<0)?;south+ (0<2)?;north` can be executed by executing `(0<2)?;east` or `(0<2)?;north`.

Parallel execution is modelled by interleaving. The parallel subgoals may communicate with each other through shared variables: If one of the subgoals retrieves data from the belief base, the substitution so obtained is also applied to the other parallel subgoal. (Note that $\theta$ does not have to be applied to $\pi_1'$ below.)

**Definition 20.** *(execution rule 5; parallel composition)*

$$\frac{\langle \pi_1, \sigma \rangle_V \longrightarrow_\theta \langle \pi_1', \sigma' \rangle}{\langle \pi_1 || \pi_2, \sigma \rangle_V \longrightarrow_\theta \langle \pi_1' || \pi_2\theta, \sigma' \rangle}$$

(A similar rule can be given for the right branch.)

For example, Greedy might use the following somewhat more complicated plan for moving:
`(robot(Greedy,x0,y0)?;[(xc(x0,y1)<x0)?;west+(x0<xc(x0,y1))?;east])||`
`(robot(Greedy,x1,y1)?;[(yc(x0,y1)<y1)?;north+(y1<yc(x0,y1))?;south])`. The left parallel subgoal handles horizontal movement (west, east), the right parallel subgoal handles vertical movement. The left subgoal communicates horizontal coordinates to the right subgoal via variable x0; the right subgoal communicates through variable y1. (Note that the functions `xc` and `yc` cannot be computed without the values of its two arguments.)

Finally, we need one more rule for executing a goal from the goal base in a mental state. We call a mental state the *top level* of execution. At this level, the various goals of the agent are executed in a parallel fashion without communication. This means the top-level goals are independently executing from each other and do not communicate through shared variables (i.e. the substitution obtained is not applied to the other goals).

**Definition 21.** *(execution rule 6; goal execution)*
Let $\Pi = \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\} \subseteq \mathcal{L}^g$, and $V = \mathsf{Free}(\Pi)$. Then:

$$\frac{\langle \pi_i, \sigma \rangle_V \longrightarrow_\theta \langle \pi_i', \sigma' \rangle}{\langle \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\}, \sigma \rangle \longrightarrow \langle \{\pi_0, \ldots \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots\}, \sigma' \rangle}$$

Execution rule 6 defines the operational semantics of an agent's mental state. Recall that an agent program was defined as a tuple consisting of the definition of the skills of the agent (basic actions), a PR-base, and its initial belief and goal base. The latter two components determine the initial mental state of the agent which is updated during execution. The former two components define global and fixed properties of the agent.

We have to take one precaution at this level, i.e. we have to make sure the free, global variables at this level do not interact with the local variables of rule-bodies introduced by practical reasoning. To ensure this

all the free variables are put in a set $V$ for later reference; the only place this set is used is in the practical reasoning rule, defined in the previous paragraph, which may introduce new local variables.

Note that we distinguish two types of transitions corresponding to goal revision (application of PR-rules) and goal execution (execution rules). The interactions between these transitions are nicely integrated in one system.

*Computations and compositionality* The transition system defines in a natural way the possible computations of a program. A *computation* of an agent program is a finite or infinite sequence of configurations $\langle \Pi_0, \sigma_0 \rangle, \langle \Pi_1, \sigma_1 \rangle, \langle \Pi_2, \sigma_2 \rangle, \ldots$, i.e. a sequence of mental states, such that for all $i$ we have $\langle \Pi_i, \sigma_i \rangle \longrightarrow \langle \Pi_{i+1}, \sigma_{i+1} \rangle$.

Given the above definition of a computation we can define various notions of *observables*. For example, we might want to observe the sequence of belief bases extracted from a computation, or the sequence of basic action symbols corresponding to single transitions in a computation, for example for planning (cf. [4]). A compositional characterization of these notions does not fall in the scope of this paper, however, it is an important issue for future research. In general a compositional description of the semantics of a programming language allows us to understand a complex program in terms of its (simpler) constituents and thus provides a basis for a better control over the design process.

## 6 Comparison and conclusions

We will not be able to discuss all the work on agent programming languages here. But we will discuss the two languages closest to our own, AGENT-0 and AgentSpeak(L), and comment on ConGolog.

Both AGENT-0 and AgentSpeak(L) are, like our own language, rule-based languages. The basic features of AGENT-0 are commitment-rules and a number of update mechanisms. The programs AGENT-0 agents can execute are restricted to the basic, primitive actions or skills of the agent. In our language the agent can handle much more general programs. Also, the commitment (goal, in our terminology) revision mechanism of AGENT-0 is restricted to removing infeasible commitments and uses a built-in mechanism for this update. In our language, we allow much more general revision rules which are provided by the programmer in the form of practical reasoning rules. AGENT-0 includes primitives for communcation, which our language does not yet provide. However, communication primitives are already extensively investigated in concurrency theory and therefore it should not be to difficult to incorporate communication in our framework in the near future ([16]). The update mechanisms of AGENT-0 are all rule-based and we believe that for all these features we can provide rules in our language which perform the same function.

AgentSpeak(L) is a rule-based language which is quite similar to the language proposed in this paper. Similar remarks as for AGENT-0 apply here. We provide more general and high-level programming constructs, and features for goal revision which AgentSpeak(L) lacks. AgentSpeak(L) incorporates a notion of events which we do not have in our language. It is not quite clear, however, whether or not this is a crucial notion in the language AgentSpeak(L). Actually, we can formally prove that we can simulate every program defined in AgentSpeak(L) in our language (which is discussed more formally in the extended version of this paper).

Although at first sight AGENT-0 and AgentSpeak(L) may seem quite different, we think the main difference resides in the control structure of these languages. This difference is reflected in the different order in which rules in these two languages are applied (cf. the discussion of the interpreters of the languages in resp. [12] and [11]).

Finally, we make a remark about ConGolog. In [4] a transition system is incorporated into the formalism of the situation calculus. Although the definitions are similar to our execution rules, there remain important differences between ConGolog and our language. First of all, (Con)Golog is a logic-programming language. This differs from our framework since we propose a genuine combination of features of logic programming and imperative programming. ConGolog extracts from a given high-level program a deterministic sequence of primitive actions. This provides for a kind of planning as defined in [4]. This focus also differs from ours since we are more concerned with the execution of programs (goals and plans) and the revision of these goals and plans. In our view we provide a more dynamic perspective contrasting the rather static planning of ConGolog.

In our paper we showed that a transition system is a suitable formalism for specifying the operational semantics of agent programming languages. Transition systems are well-understood and very lucid. A programming language combining features of logic programming and imperative programming was defined. This language includes all the regular programming constructs from imperative programming and the pattern-matching mechanism of logic programming. A number of distinguishing features of agents were identified. In particular, the general heads of practical reasoning rules for goal revision are a unique property of intelligent agents.

In future research we want to investigate extensions of our framework to multiagent systems with communication and we want to study mechanisms for failure recovery. For example, what type of operators for failure recovery might usefully be incorporated in the language? We want to exploit the notions of standard concurrency theory (CCS, $\pi$-calculus,etc.) as far as possible in modelling agent-oriented features. It remains to be investigated how to incorporate and finetune these notions for agent programming languages. We also want to study more closely the general practical reasoning rule we introduced in our paper, and would like to investigate the role of control structure in the decision mechanisms of agents. On the more theoretical side, we already mentioned as future research the important issue of compositionality of the semantics, and corresponding proof theories and refinement calculi, in order to obtain a method for the design of agents.

## References


1. P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
2. B. Dunin-Keplicz and J. Treur. Compositional Formal Specification of Multi-Agent Systems. In M.J. Wooldridge and N.R. Jennings, editors, *Intelligent Agents*, pages 102–117. Springer-Verlag, 1995.
3. M. Fisher. A Survey of Concurrent MetateM - The Language and its Applications. In *Temporal Logic*, pages 480–505. Springer, 1994.
4. G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. accepted for IJCAI-97.
5. Y. Lespérance, H.J. Levesque, F. Lin, and D. Marcu. Foundations of a Logical Approach to Agent Programming. In M.J. Wooldridge, J.P. Müller, and M. Tambe, editors, *Intelligent Agents II*, pages 331–346. Springer, 1996.
6. R. Levinson. A general programming language for unified planning and control. *Artificial Intelligence*, 76:319–375, 1995.
7. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1987.
8. J.P. Müller. *The Design of Intelligent Agents*. Springer, 1996.
9. G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, Computer Science Department, 1981.
10. A. Poggi. DAISY: An object-oriented system for distributed artificial intelligence. In M.J. Wooldridge and N.R. Jennings, editors, *Intelligent Agents*, pages 341–354. Springer, 1995.
11. Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. van der Velde and J.W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer, 1996.
12. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
13. M.P. Singh. *Multiagent systems*, volume 799 of *LNAI*. Springer, 1994.
14. Sarah Rebecca Thomas. *PLACA, An Agent Oriented Programming Language*. PhD thesis, Department of Computer Science, Stanford University, 1993.
15. W. van der Hoek, B. van Linder, and J.-J. Ch. Meyer. A logic of capabilities. In A. Nerode and Y.V. Matiyasevich, editors, *Proc. of the third int. symposium on the logical foundations of computer science*, pages 366–378. Springer, 1994.
16. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Concurrent programming languages for multi-agent systems. Technical report, Department of Computer Science, University Utrecht, 1997.
17. B. van Linder, W. van der Hoek, and J.-J.Ch. Meyer. Formalising motivational attitudes of agents: On preferences, goals, and commitments. In M.J. Wooldridge, J.P. Müller, and M. Tambe, editors, *Intelligent agents II*, pages 17–32. Springer, 1996.
18. M.J. Wooldridge. A Knowledge-Theoretic Semantics for Concurrent MetateM. In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III*, pages 357–374. Springer, 1997.