Agent Programming in 3APL

Koen V. Hindriks, Frank S. de Boer,
Wiebe van der Hoek, and John-Jules Ch. Meyer

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

Abstract

An intriguing and relatively new metaphor in the programming community is that of an *intelligent agent*. The idea is to view programs as intelligent agents acting on our behalf. By using the metaphor of intelligent agents the programmer views programs as entities which have a mental state consisting of beliefs and goals. The computational behaviour of an agent is explained in terms of the decisions the agent makes on the basis of its mental state. It is assumed that this way of looking at programs may enhance the design and development of complex computational systems.

To support this new style of programming, we propose the agent programming language 3APL. 3APL has a clear and formally defined semantics. The operational semantics of the language is defined by means of transition systems. 3APL is a combination of imperative and logic programming. From imperative programming the language inherits the full range of regular programming constructs, including recursive procedures, and a notion of state-based computation. States of agents, however, are belief or knowledge bases, which are different from the usual variable assignments of imperative programming. From logic programming, the language inherits the proof as computation model as a basic means of computation for querying the belief base of an agent. These features are well-understood and provide a solid basis for a structured agent programming language. Moreover, on top of that 3APL agents use so-called *practical reasoning rules* which extend the familiar recursive rules of imperative programming in several ways. Practical reasoning rules can be used to monitor and revise the goals of an agent, and provide an agent with reflective capabilities.

Applying the metaphor of intelligent agents means taking a design stance. From this perspective, a program is taken as an entity with a

mental state, which acts pro-actively and reactively, and has reflective capabilities. We illustrate how the metaphor of intelligent agents is supported by the programming language. We also discuss the design of control structures for rule-based agent languages. A control structure provides a solution to the problem of which goals and which rules an agent should select. We provide a concrete and intuitive ordering on the practical reasoning rules on which such a selection mechanism can be based. The ordering is based on the metaphor of intelligent agents. Furthermore, we provide a language with a formal semantics for programming control structures. The main idea is not to integrate this language into the agent language itself, but to provide the facilities for programming control structures at a meta level. The operational semantics is accordingly specified at the meta level, by means of a meta transition system.

Keywords: intelligent agent, agent-oriented programming, practical reasoning rule, comparison of agent programming languages, control structure, selection mechanism, formal semantics, meta transition system.

1 Introduction

An intriguing and relatively new metaphor in the programming community is that of an *intelligent agent*. The idea is to view programs as intelligent agents acting on our behalf. By using the metaphor of intelligent agents the programmer views programs as entities which have a mental state consisting of beliefs and goals. The computational behaviour of an agent is explained in terms of the decisions it makes on the basis of its beliefs and goals. It is assumed that this way of looking at programs may enhance the design and development of complex computational systems.

In this paper, we present our approach to agent oriented programming. To support this style of programming, we propose the agent programming language 3APL (pronounced "triple-a-p-l"). 3APL is a new programming language which incorporates features from both imperative and logic programming and also includes some additional new features, which allow for an elegant description of many agent oriented features. In particular, the proof as computation model of logic programming is used to implement the querying of the beliefs an agent. Whereas concepts from imperative programming are used to describe the execution of the goals of an agent. On top of that, the programming language supports the programming of agents which have reflective capabilities related to their goals or plans. These re-

flective capabilities are provided by so-called *practical reasoning rules*.

We present a detailed account of most aspects of the agent language in this paper. The language is introduced both informally as well as formally. We believe it is particularly important to specify the formal semantics of an agent language. The advantage provided by a formal semantics is that it facilitates the specification of and reasoning about agents and moreover allows a detailed comparison with other approaches. The semantics of 3APL we present in this paper is an *operational semantics*. The operational semantics is defined in terms of transition systems ([18]). This is a simple and lucid formalism for specifying the meaning of programs.

In the second part of the paper, we extend our framework to deal with the problem of programming *control structures* for agents. The issues that arise at this level concern the *selection of goals* or plans and the *selection of practical reasoning rules*. We distinguish between an object level which concerns the programming of agents in the agent language 3APL and a meta level which concerns the programming of control structures for agents in a meta level programming language. This meta language is defined by means of transition systems again, and we illustrate its use by implementing a new control structure in this language. Finally, we compare our approach with a number of related approaches.

2 The Metaphor of Intelligent Agents

The current research on intelligent agents is very broad and ranges from more theoretical and logical investigations to more practical, implemented applications. As far as we know, however, there is still no unified account of agents which bridges the gap between the theoretical and more practical work on agents. First, formal logics, like BDI-logic or the KARO-framework ([21, 23, 25, 26], for example, are presented as a means for specifying intelligent agents. Nevertheless, so far no methodology has been provided for refining such specifications to known programming languages. Secondly, on the more practical side, a lot of agent architectures have been proposed. Although many useful insights have been provided by work on agent architectures, agent architectures can result in complicated and specialised solutions for building agents. A third approach consists in the design of agent programming languages or agent design frameworks, e.g. [4, 19, 20, 22, 24]. Some of these languages, however, lack a clear and formally defined semantics, and therefore it is difficult to formalise the design, specification and verification of programs. Other types of languages are based directly on

logic ([7, 16, 6, 27]).

In this paper, we consider the third approach of desinging a programming language for building programs based on the metaphor of intelligent agents. To promote this new style of programming, we think it is important to show at least three things. First of all, we believe it is important to show that the metaphor of intelligent agents enhances the programming and design task in particular domains by providing *convincing programming examples* which illustrate the power of using this metaphor. Some preliminary and suggestive work has been done to show this. Secondly, an *agent programming language* which supports the programming of agents should be provided. In this respect, it is particularly important to make clear how the agent language supports agent oriented programming and to be clear about which aspects associated with agents are supported. For this purpose, an account of intelligent agents which explicates these notions in a precise and computationally useful way is required. Finally, we believe it is important to provide a *logic of agents* to reason about the agents written in the agent programming language. In this paper, our concern is mainly with the second issue, that is, with providing a proposal for an agent programming language.

Before we proceed, and introduce the agent programming language 3APL, we give a basic outline of intelligent agents which identifies their main features. This outline is intended as a partial answer as to what type of features an agent programming language should support. We think that most researchers would agree that (at least most of) the following characteristics are constitutive of intelligent 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, and
- agents have *reflective* or meta-level reasoning *capabilities*.

Intelligent agents are *goal-directed* entities, which means that they have a set of goals and associated plans which result in the execution of primitive or basic actions. To realise their goals, agents need to find suitable plans which involves a particular type of reasoning known as *practical reasoning*. Moreover, an agent needs to have the ability to *monitor* its success or failure, which requires reflective capabilities with respect to its goals or plans. Finally, beliefs represent the environment from the agent's point of view, and to keep its beliefs up to date an agent should be able to revise its beliefs.

In short, we think an agent programming language should support these concepts and associated mechanisms of updating. Therefore, an agent programming language should include features for:

- representing and querying the agent's *beliefs*,
- *belief updating*, for incorporating new and removing existing information in the agent's belief base,
- *goal updating*, to facilitate *practical reasoning*, that is, for planning and the reconsideration of adopted plans.

3 The Agent Programming Language 3APL

In this section, we make the concepts introduced informally so far more precise and introduce the programming language 3APL which is based on these concepts (cf. also [11]). The syntax of the language is introduced and the meaning of the language constructs is discussed informally.

3.1 Beliefs

The beliefs of an agent are formulas from some logical language $\mathcal{L}$. This language is used for knowledge represention purposes. In principle, the choice of the knowledge representation language is not fixed by the programming language, and could be a modal language, first order language, or any other knowledge representation language. The programmer is free to choose the logical language which fits his purposes best. However, in this paper we choose to fix the knowledge representation language to be a first order language, both for technical reasons and ease of exposition. The consequences of choosing some other language thus are not explored in this paper.

Definition 3.1 *(beliefs)*
The set of first order formulas $\mathcal{L}$ is built from a signature $\Sigma = \langle \mathsf{Pred}, \mathsf{Func} \rangle$ of predicate symbols Pred and function symbols Func, and a set of countably infinite variables Var with typical elements $x, y, z, \ldots, x_1, \ldots$. We use Term to denote the set of terms built from Func and Var.

- if $p \in \mathsf{Pred}$ and $t_1, \ldots, t_n \in \mathsf{Term}$, then $p(t_1, \ldots, t_n) \in \mathcal{L}$,
- if $t_1, t_2 \in \mathsf{Term}$, then $t_1 = t_2 \in \mathcal{L}$,
- if $\varphi \in \mathcal{L}$, then $\neg\varphi \in \mathcal{L}$,

- if $\varphi \in \mathcal{L}$, then $\varphi \wedge \psi \in \mathcal{L}$,
- if $\varphi \in \mathcal{L}$, then $\forall x(\varphi) \in \mathcal{L}$.

A number of special syntactic classes and concepts are associated with this language. First, a formula of the form $p(\vec{t})$ is called an *atom*; the set of atoms is denoted by Atom. The notions of *free* and *bound variable* are defined as usual (cf. [17]). A variable x is bound if it occurs in the scope of some quantifier $\forall x$, otherwise it is free. The set of free variables in an expression e is denoted by $\mathsf{Free}(e)$. A *ground atom* is an atom without occurrences of free variables; a *ground term* is a term without occurrences of free variables. A formula without free variables is also called a *sentence*. The usual *entailment relation* for first order logic is denoted by $\models$. Informally, $\Gamma \models \varphi$ if φ is implied by the set of sentences Γ.

Finally, we need to define the notion of a *substitution*. Substitutions play an important role in the operational semantics of the language. We use $\theta, \theta', \ldots, \gamma, \eta$ to denote substitutions.

Definition 3.2 *(substitution)*

- A *substitution* θ is a *finite* set of *pairs* (also called *bindings*) of the form $x_i = t_i$, where $x_i \in \mathsf{Var}$ and $t_i \in \mathsf{Term}$, and $x_i \neq x_j$ for every $i \neq j$, and $x_i \notin \mathsf{Free}(t_j)$, for any i and j,
- A *ground* substitution θ is a substitution such that for every pair $x = t \in \theta$ the term t is ground, i.e. $\mathsf{Free}(t) = \varnothing$,
- The domain of θ, denoted by $dom(\theta)$, is the set of variables x for which θ contains a pair $x = t$.

We define the application of a substitution to a syntactic expression only informally, to avoid the need to repeat similar definitions for every syntactic category introduced below (a formal definition of the application of substitutions to formulas can be found in [17]).

Definition 3.3 *(application of substitution)*
Let e be any syntactic expression (including new ones to be introduced in the sequel), and θ be a substitution. Then $e\theta$ denotes the expression where all *free* variables x in e for which $x = t \in \theta$ are *simultaneously* replaced by t.

Now we are in a position to formally define the concept of a *belief base* of an agent. Note that a belief base is closed, i.e. does not contain free variables. The means to query the belief base are provided by a test goal which is introduced below.

Definition 3.4 *(belief base)*
A *belief base* σ is a sentence from $\mathcal{L}$.

Example 3.5 *(a personal assistant)*
The running example in this paper concerns a personal assistant for maintaining the agenda of its particular user. This type of personal assistants support their users in scheduling their activities by automating certain routine actions. Moreover, they may provide their users with relevant information related to these activities, like, for example, information about transportation.

In the examples, we use the Prolog-style convention that strings starting with capital letters are variables. The beliefs of a personal assistant concern the agenda of its user. A predicate *agenda* with four arguments *Act*, *Time*, *Duration*, and *Location*, respectively, is used to keep track of the items in the user's agenda. For example, $agenda(teach, 11:00, 2:00, c009)$ states that at 11am the user has to teach a class for 2 hours in room $c009$. An integrity constraint associated with the predicate *agenda* is that items in the agenda are not allowed to overlap in time. Formally, this is represented by [1]:

$$(agenda(A1, T1, D1, L1) \wedge agenda(A2, T2, D2, L2) \wedge T1 \leq T2 < T1 + D1) \\ \rightarrow A1 = A2 \wedge T1 = T2 \wedge D1 = D2 \wedge L1 = L2$$

The predicate $location(Loc, Time)$ is used to keep track of the user's location at a particular time. The predicate is assumed to be persistent. That is, if there is no information to the contrary the location of the user at a time t' such that $t < t'$ is by default assumed to be the same as that of time t. Since the information concerning the location of the user is derived from the activities and associated locations in the user's agenda, a closed world assumption associated with the *agenda* predicate is needed to implement this. In that case, we can specify the relation between the two predicates as follows:

$$(agenda(A1, T1, D1, Loc) \wedge T1 \leq Time \wedge \\ (T1 < T2 < Time \rightarrow \neg agenda(A2, T2, D2, L2))) \rightarrow \\ location(Loc, Time)$$

[1] Throughout the paper, free first order variables in examples concerning belief bases are implicitly universally quantified.

3.2 Goals and actions

In common sense, and in most logics of agents, the goals of an agent describe the state of affairs an agent would like to be realised. These goals are *declarative* in nature, and are also called *goals-to-be*. Although much research has gone into clarifying the notion of a declarative goal, it is still not very clear how to integrate such goals, apart from the most simple cases, into either agent programming languages or agent architectures.

There is, however, a second, more *procedural* notion of goal. These goals are also called *goals-to-do*, because they specify a plan of action the agent is intending to execute. Also, the motivational concept of intention has been analysed in terms of plans ([1]). In Artificial Intelligence, moreover, the concept of a plan has since long been recognised as similar to that of an imperative program.

From a computational perspective, it is most natural to focus on this second type of goal. Goals in this sense are a kind of imperative program, which given a set of simple goals can be composed into more complex goals by means of the usual operators from imperative programming. It is this notion of goal that is incorporated in the agent language 3APL.

First, we introduce *basic actions*, which constitute one of the simple goals in the language. Basic actions specify the capabilities which an agent has to achieve a particular state of affairs. It is important in this context to emphasise that an agent is viewed as a mental entity. Although it may well be that an agent changes its environment through some interface which depends on the execution of basic actions in the agent language, there is nothing in the agent language which requires such an interface with an environment external to the agent. The use of intelligent agents to control robots is an example where such an interface is required. Personal assistants, which are agents to maintain an agenda of their user, however, do not require any interface to control some external environment. However, in both cases the beliefs of an agent represent something external to that agent itself. Blocks and other objects in the first case, activities, locations and so on, in the second case.

Because a 3APL agent essentially is a mental entity, from the point of view of the programming language basic actions are actions which affect the mental state of the agent (although these actions, as we noted in the previous paragraph, in practice may be connected with some interface). Basic actions are actions which update or change the beliefs of an agent. This is most natural, since these updates change the representation of the environment the agent is supposed to control (or represent, in the case of

personal assistants) by means of performing actions. For example, the action $\mathsf{insagenda}(Act, Time, Duration, Location)$ inserts in the agenda in the belief base the activity *Act* at time *Time* if it does not violate any integrity constraints associated with the agenda.

Definition 3.6 *(basic actions)*
Let Asym be a set of action symbols with typical elements a, a' each with a given arity. Then $\mathsf{a}(t_1, \ldots, t_n)$ is a *basic action* for any action symbol $\mathsf{a} \in \mathsf{Asym}$ and terms $t_1, \ldots, t_n \in \mathsf{Term}$ where n is the arity of a. The set of basic actions is denoted by Bact.

There are two other simple goals, i.e. *achievement goals* and *test goals*. *Achievement goals* are atomic propositions from the logical language $\mathcal{L}$. The use of atoms as achievement goals, however, is very different from the use of atoms as beliefs. Whereas in the latter case atoms are used to represent and therefore are of a declarative nature, in the former case they serve as an abstraction mechanism like procedures in imperative programming and have a procedural meaning. Apart from the procedural reading of these goals, however, an assertional reading is also possible. An achievement goal $p(\vec{t})$ would then be interpreted as specifying a goal to achieve a state of affairs such that $p(\vec{t})$. We think such a reading is valid in case the plans for achieving an achievement goal $p(\vec{t})$ actually do establish $p(\vec{t})$. Plans to achieve an achievement goal are provided by practical reasoning rules which are introduced below.

The last type of basic goal, a *test goal* $\varphi?$, allows an agent to introspect its beliefs. A test is evaluated relative to the current beliefs of an agent. Their main use, however, is not just to check whether or not the agent believes a particular proposition, but to compute values or bindings for the free variables which occur in the test. In this respect a test is similar to an assignment in imperative programming. The main difference is that a test is evaluated by means of logical proof (as in logic programming), and a test can only be used to *initialise* a variable to some value, not to update the value assigned to a variable.

Together, the *basic actions* $\mathsf{a}(\vec{t})$, *achievement goals* $p(\vec{t})$, and the test goals $\varphi?$ are the *basic goals* in the agent language. The complex goals are composed from basic goals by using the programming constructs for *sequential composition* and *nondeterministic choice* which are the well-known programming constructs from imperative programming. This allows the specification of a sequence of goals $\pi_1; \pi_2$ and disjunctive goals $\pi_1 + \pi_2$ by means of nondeterministic choice, i.e. do either π_1 or π_2. An important

aspect of the parameter mechanism of 3APL is how different occurrences of the same variable relate to each other. The rule is that the first occurrence of a free variable in a sequential composition *implicitly* binds all later occurrences of that same variable. This will be discussed in more detail below. An agent also may have goals which are executed in parallel, which is not explicitly represented by some programming operator, but is a result of the fact that an agent may have more than one goal at the same time.

Definition 3.7 *(goals)*
The set of *goals* Goal is inductively defined by:

1 Bact $\subseteq$ Goal,

2 Atom $\subseteq$ Goal,

3 If $\varphi \in \mathcal{L}$, then $\varphi? \in$ Goal,

4 If $\pi_1, \pi_2 \in$ Goal, then $(\pi_1;\ \pi_2), (\pi_1 + \pi_2) \in$ Goal.

Note that by using test goals and nondeterministic choice we can define a new construct IF ... THEN ... ELSE ... as follows:

$$\mathsf{IF}\ \varphi\ \mathsf{THEN}\ \pi_1\ \mathsf{ELSE}\ \pi_2 \stackrel{df}{=} (\varphi?;\ \pi_1 + \neg\varphi?;\ \pi_2)$$

An important difference with the same construct from imperative programming, however, is that IF φ THEN π_1 ELSE π_2 cannot always be executed. This is due to the possibility that neither φ nor $\neg\varphi$ is entailed by the current beliefs of an agent. Moreover, we did not include an iteration operator, which could be used to define a construct WHILE ... DO .... Such operators, however, can be defined by means of recursive rules which are introduced below.

Example 3.8 The personal assistant of our running example is capable of performing a number of basic actions related to maintaining an agenda and informing the user about appointments etc. Among others, the agent is capable of performing an action

$$\mathsf{inform\_user}(\mathit{String}, \mathit{Act}, \mathit{Time}, \mathit{Duration}, \mathit{Location})$$

to inform the user about an activity where *String* denotes a string which is prefixed to the output of the parameters concerning the activity, and

$$\mathsf{insagenda}(\mathit{Act}, \mathit{Time}, \mathit{Duration}, \mathit{Location})$$

to insert an item into the agenda and keep the agenda up to date.

The test $location(Location, Time)?$ can be used to find out about the location of the user at a given time (or, in case a location already has been assigned to the variable *Location*, the times the user was or is going to be at that particular location according to its agenda).

An example of an achievement goal is $schedule(Act, Time, Dur, Loc)$ which may be used for scheduling an activity. A plan to implement this goal is given below by means of a practical reasoning rule.

Finally, we give an example of a sequential goal consisting of a test followed by a basic action. The goal

$$location(Location, 12:00)?;\ \mathsf{insagenda}(lunch, 12:00, 1:00, Location)$$

by first executing the test derives from the current beliefs the location of the user at 12am and computes a binding for the variable *Location*. Let's assume that the location retrieved is the office of the user. Since the second occurrence of the variable *Location* is implicitly bound by the first occurrence, the value computed by the test for this variable is then used to instantiate the second occurrence and a lunch at 12am in the office of the user is inserted in the agenda.

As is clear from the discussion so far, goals operate on the belief base of an agent. They both change and derive information from the agent's beliefs. Their main purpose is in manipulating the beliefs of the agent. Goals, therefore, can be viewed as *belief update operators*.

Remark 3.9 *(note on terminology)*
We use the notion of a *goal* rather than that of *intention*. The reason is that the notion of a goal is a more general notion than that of intention. Intentions are usually viewed as some kind of *choice* with an associated level of *commitment* made to that choice ([2]). The commitment made to a choice determines when an agent will reconsider or drop its intention. An agent may adopt several commitment strategies towards its intentions. In the programming language, a goal does reflect a choice the agent has made. However, there is no explicit level of commitment associated with each of the goals of an agent. The commitment strategies or revision strategies of an agent are more or less implicit in the practical reasoning rules of an agent. The practical reasoning rules of an agent thus determine whether or not a goal can be considered as an intention.

3.3 Practical Reasoning Rules

In this section, we discuss the notion of a practical reasoning rule. The main purpose of practical reasoning rules is that they supply the agent with a facility to manipulate its goals. Whereas goals operate on the beliefs of an agent, rules operate on the goals. These rules both facilitate planning and monitoring of the goals of an agent. Practical reasoning rules can be used to build a plan library from which an agent can retrieve plans for achieving a achievement goal $p(\vec{t})$. They also can be used to provide the means to revise and monitor goals of the agent.

The name of these rules derives from the role they play in the operation of intelligent agents. They supply agents with reasoning capabilities to reflect on their goals. The type of reasoning involved is similar to a certain extend to the practical reasoning used by human agents to achieve their goals. Informally, this type of common sense reasoning can be paraphrased as follows. From a goal to achieve φ and the belief that the plan π is sufficient to achieve φ, the agent concludes that it is reasonable to adopt the plan π. This type of reasoning is also called *means-end reasoning*. The goals considered in this type of reasoning are declarative, and only simple achievement goals correspond to this type of goals in 3APL. Note that in our explanation of practical reasoning the conclusion of this type of reasoning is the adoption of a new goal (and not the performance of some action, as some philosophers claim should be the result of practical reasoning).

Apart from means-end reasoning, an agent may also have to reconsider the plans it adopted. The reasoning involved in reconsidering one's goals follows a similar pattern as the means-end reasoning in the previous paragraph. Informally, a commitment to a plan π and a belief that the situation requires replacing this plan with an alternative course of action π', provide the agent with reasons to conclude that it should adopt π'. We will classify both types of reasoning under the heading of *practical reasoning*, and illustrate below how rules are used to implement this type of reasoning.

In the example patterns of practical reasoning we presented, it is not too difficult to recognise a type of conditional rule. The practical reasoning rules we are about to introduce capture the essence of this pattern of reasoning. Presented with a plan and a belief concerning the current situation, a practical reasoning rule allows the agent to adopt a new plan which replaces its original plan.

To support these reflective capabilities, the language is extended with a new type of variables, called *goal variables*. As the name already suggests, these variables range over goals. We extend the notion of goals and intro-

duce a new concept of *semi-goals* which are constructed from basic actions, achievement goals, tests, sequential composition and nondeterministic choice like regular goals, but may also include goal variables. The goal variables in semi-goals should be thought of as *place-holders* for goals.

Definition 3.10 *(semi-goals)*
Let Gvar be a countably infinite set of goal variables (ranging over goals) with typical elements X, X' such that $\mathsf{Gvar} \cap \mathsf{Var} = \varnothing$.
The set of *semi-goals* SGoal is inductively defined by the syntactic rules **1-4** from definition 3.7, where Goal is replaced by SGoal, and the rule

5 $\mathsf{Gvar} \subseteq \mathsf{SGoal}$.

Substitutions are generalised to be able to deal with these new variables and now may also include bindings of the form X/π where X is a goal variable and π is a goal.

Now we are in a position to define practical reasoning rules. The set of practical reasoning rules is built from semi-goals and first order formulas. We introduce three different types of practical reasoning rules.

Definition 3.11 *(practical reasoning rules)*
Let $\pi_h, \pi_b \in \mathsf{SGoal}$ be semi-goals, and $\varphi \in \mathcal{L}$ be a first order formula. Then the set of *practical reasoning rules* Rule is defined by:

- $\pi_h \leftarrow \varphi \mid \pi_b \in \mathsf{Rule}$ such that any goal variable X occurring in π_b also occurs in π_h,
- $\leftarrow \varphi \mid \pi_b \in \mathsf{Rule}$ such that no goal variables occur in π_b, i.e. $\pi_b \in \mathsf{Goal}$, and
- $\pi_h \leftarrow \varphi \in \mathsf{Rule}$.

A practical reasoning rule has the following components:

Definition 3.12 *(head, body, guard, global and local variables)*
Let $\pi \leftarrow \varphi \mid \pi'$ be a practical reasoning rule.

- π is called the *head* of the rule,
- π' is called the *body* of the rule,
- φ is called the *guard* of the rule,

- the free first order variables in the head of a rule are called the *global variables* of the rule,
- all first order variables in the body of a rule which are not global are called *local variables*.

Informally, one can read a practical reasoning rule $\pi \leftarrow \varphi \mid \pi'$ as stating that if the agent has adopted some goal or plan π and believes that φ is the case, then it may consider adopting goal π' as a new goal. The application of a rule is based on pattern-matching: A rule $\pi_h \leftarrow \varphi \mid \pi_b$ applies to a current (sub)goal π of an agent if π matches with π_h. The guard of the rule (more precisely, an instance of the guard) must also be entailed by the agent's current beliefs for a rule to be applicable. A current (sub)goal of an agent that matches with the head of a rule π_h is *replaced* by the body of that rule if the rule is applied.

By convention, we write $\pi_h \leftarrow \pi_b$ for $\pi_h \leftarrow \mathsf{true} \mid \pi_b$. Rules of the form $\leftarrow \varphi \mid \pi$ are said to have an *empty head*, whereas rules of the form $\pi \leftarrow \varphi$ are said to have an *empty body*. Rules with an empty body are used to *drop goals*. Rules with an empty head are used to *create new goals* independent of the current goals of an agent.

Although practical reasoning rules are superficially akin to (guarded) horn clauses known from (concurrent) logic programming, in fact they are very different from such rules. The best way to think about these rules is as extending recursive procedures from imperative programming. Practical reasoning rules extend recursive rules because they can also be used to modify plans in arbitrary ways, as we illustrate below, which is not a feature of recursive procedures.

The distinction between global and local variables is made to separate the local data-processing in the body by means of the local variables which are only used in the body, from the global variables which may be used both as input variables and output variables. Global variables can be used as input variables to supply data to the body of the rule, and as output variables to return computed results. All local variables in a body are renamed to variables which do not occur anywhere else in the goals of agents, which provides for an *implicit scoping mechanism*, similar to that in logic programming.

Example 3.13 We first give an example of a rule which provides a plan for the achievement goal $schedule(Act, Time, Dur, Loc)$ of example 3.8 and

illustrate the application of this rule to a particular instance of this goal.

$$\begin{array}{l} \mathit{schedule}(\mathit{Act}, \mathit{Time}, \mathit{Dur}, \mathit{Loc}) \leftarrow \mathit{location}(\mathit{FromLoc}, \mathit{Time}) \ \mid \\ \quad \mathit{transport}(\mathit{Means}, \mathit{FromLoc}, \mathit{Loc}, \mathit{DurTrans})?; \\ \quad \mathsf{insagenda}(\mathit{Means}, \mathit{Time} - \mathit{DurTrans}, \mathit{DurTrans}, \mathit{FromLoc}); \\ \quad \mathsf{insagenda}(\mathit{Act}, \mathit{Time}, \mathit{Dur}, \mathit{Loc}) \end{array}$$

The plan in the body of this rule is to schedule the activity *Act* by inserting it in the agenda at the appropriate time. But the agent also supports the user by automatically calculating which means of transportation are sufficient to get to the location of the activity in time and reserving time for travelling in the agenda by means of the first insagenda action in the plan. This is achieved by retrieving this information by means of the test $\mathit{transport}(\mathit{Means}, \mathit{Fromloc}, \mathit{Loc}, \mathit{DurTrans})?$ where *Means* returns the type of transportation required to get from *Fromloc* to the destination *Loc* and *DurTrans* denotes the time it takes to get to the destination.

Now suppose that the agent has a goal of scheduling a meeting in Utrecht of one hour at 11am:

$$\mathit{schedule}(\mathit{meeting}, 11:00, 1:00, \mathit{utrecht})$$

Moreover, also assume that the agent believes that:

$$\begin{array}{l} (\mathit{agenda}(A, T, D, L) \leftrightarrow \\ \quad (A = \mathit{meeting} \wedge T = 9:00 \wedge D = 1:00 \wedge L = \mathit{amsterdam})) \wedge \\ \mathit{transport}(\mathit{train}, \mathit{amsterdam}, \mathit{utrecht}, 0:45) \wedge \\ ((\mathit{agenda}(A1, T1, D1, \mathit{Loc}) \wedge T1 \leq \mathit{Time} \wedge \\ (T1 < T2 < \mathit{Time} \rightarrow \neg \mathit{agenda}(A2, T2, D2, L2))) \rightarrow \\ \qquad \mathit{location}(\mathit{Loc}, \mathit{Time})) \end{array}$$

In that case, the rule for scheduling an activity is applicable. First, the head of the rule matches with the current achievement goal of the agent; the substitution $\{\mathit{Act} = \mathit{meeting}, \mathit{Time} = 11:00, \mathit{Dur} = 1:00, \mathit{Loc} = \mathit{utrecht}\}$ is a (most general) unifier of the head of the rule and the current goal of the agent. Second, because of the closed world assumption associated with the predicate *agenda*, the agent can derive $\mathit{location}(\mathit{amsterdam}, 11:00)$ from its beliefs, which is an instance of the guard of the rule.

All the global variables in the head of the rule are used as input variables in this example. These input values are used to instantiate the body of the rule. By replacing its current achievement goal with the new plan the agent

ends up with the new goal:

$transport(Means, Fromloc, utrecht, DurTrans)?;$
$\mathsf{insagenda}(Means, 11:00 - DurTrans, DurTrans, Fromloc);$
$\mathsf{insagenda}(meeting, 11:00, 1:00, utrecht)$

Note that only the global input variables are instantiated in the body. The binding for the variable *Fromloc* is recorded in a substitution which is similar to a variable assignment in imperative programming (see also the operational semantics below).

The example illustrates the use of a simple rule of the form $p(\vec{t}) \leftarrow \phi \mid \pi$ where $p(\vec{t})$ is an achievement goal. This type of rule is similar to a (recursive) procedure in imperative programming. These rules are useful for specifying a *plan* to achieve an achievement goal. We also call rules of this form *plan rules*. A plan rule encodes the *procedural knowledge* of an agent. A set of such rules constitutes a plan library which an agent can consult to find plans to achieve its goals. The retrieval of these plans occurs during the operation of the agent, which is quite different from static planning systems like STRIPS (cf. [5]).

Example 3.14 We provide two more rules to illustrate the use of recursion. The purpose of these rules is simply to provide the user with a pre-calculated list of items in its agenda. Each of these items consists of the activity involved, and the time, the duration and the location of the activity.

$inform\_list(ActList) \leftarrow ActList = [[Act, Time, Dur, Loc], L] \mid$
$\quad \mathsf{inform\_user}(\mathtt{scheduled}, Act, Time, Dur, Loc);\ inform\_list(L)$
$inform\_list(ActList) \leftarrow ActList = [] \mid$

The rules recursively inform the user of the consecutive items in the list *ActList*. Note how the second rule which has an empty body is used to deal with the termination of the recursion. The two rules also illustrate how recursion can be used to program iteration.

Apart from the plan rules illustrated so far, there are a number of other types of rules. As we already mentioned, rules with an empty head $\leftarrow \phi \mid p(\vec{t})$ can be used for *goal creation*. Such rules are useful for implementing reactive behaviour of an agent; moreover, rules of the form $\pi \leftarrow \phi$ can be used to *drop* goals.

In particular, however, the use of goal variables in rules allows for very general types of goal modification. For example, the rule $X;\ Y \leftarrow Y;\ X$

changes the order of the goals in a sequential goal. Although this rule may not be very useful, it illustrates the quite general type of modification allowed for by practical reasoning rules. As we illustrated elsewhere, rules with goal variables are particularly useful for monitoring goals, since they can be used to implement interrupts ([8]). Note that the modification of goals by means of rules cannot simply be replaced, for example, by the IF ... THEN ... ELSE ... construct. For example, it is not possible to simulate the rule $X;\ Y \leftarrow Y;\ X$ in this way.

This kind of modification of goals has a number of consequences. One of the more important ones is that sequential composition is no longer associative. That is, $(\pi_1;\ \pi_2);\ \pi_3$ is not equivalent to $\pi_1;\ (\pi_2;\ \pi_3)$, as is easily verified by applying the rule $X;\ Y \leftarrow Y;\ X$ to both goals and comparing the results. Therefore, by convention, we stipulate that sequential composition ; is *left associative*.

The use of the goal variables resides in the way they support very general modification of goals. The use of these variables in practical reasoning rules allows for all kinds of revision and monitoring facilities. Practical reasoning rules, therefore, provide an agent with *reflective capabilities* concerning its goals. In particular, they may be used to deal with failure of a current plan of an agent.

Example 3.15 In this example we provide a very simple illustration of the use of other practical reasoning rules than plan rules and of goal variables. A more extensive discussion of the use of practical reasoning rules and goal variables can be found in [8].

Recall the plan from example 3.13 for scheduling an activity A. This plan first attempts to compute the appropriate means of transportation to get to the location of the activity A which the agent tries to schedule, then attempts to insert the activity of travelling and the time it takes in the agenda in the appropriate place, and finally attempts to schedule the activity A itself.

Of course, this plan will not always succeed. In particular, one case in which it will fail is when there is not enough time to travel to get to the location of activity A because of other scheduled activities. Now, in 3APL there are two ways of dealing with this failure. On the one hand, there is the traditional approach which insists that the plan is *incorrect* and has to be *rewritten* to correct the errors in it. On the other hand, in 3APL, there is a second option to deal with this failure by using practical reasoning rules. In that case the original plan does not have to be rewritten, and can be viewed as the correct plan *in the normal case*. Rules to deal with the failure are

added to deal with the *abnormal cases* in which the plan would fail.

A practical reasoning rule to deal with the case where not enough time is available to travel to the location of the activity is presented next. The guard of the rule implements the condition that there is not enough time to travel. The crucial parameter here which needs to be retrieved from the scheduling plan is the time at which the travelling is supposed to begin. This is retrieved from the scheduling plan by matching the head of the rule which consists of an insagenda action and a goal variable with the scheduling plan. The variable *Time* then retrieves the time the travelling is supposed to begin. In the guard, it is checked whether this time conflicts with another activity which has already been scheduled. In that case, the rule applies and the original scheduling plan is replaced by an action to inform the user of an unsuccessful attempt to reserve time for travelling. Finally, the goal variable is used to match with the second part of the scheduling plan consisting of a second insagenda action which also needs to be replaced in case of failure.

$$\begin{array}{l}\mathsf{insagenda}(Act, Time, Dur, Loc);\ X \leftarrow \\ agenda(A1, T1, D1, L1) \wedge \\ ((T1 \leq Time < T1 + D1) \vee (Time < T1 < Time + Dur)) \\ \qquad \mathsf{inform\_user}(\mathtt{unsuccessful}, Act, Time, Dur, Loc)\end{array}$$

In section 6.4, we will give a detailed account of the application of this rule to a scheduling plan.

4 Classification of rules

If we look more closely to the different purposes that practical reasoning rules can be used for, we can classify rules and impose an intuitive order on these classes. The classification which we propose both illustrates the expressive power of practical reasoning rules as well as that it may be useful for designing agents. Moreover, because the classes distinguished in our classification are motivated by common sense considerations, this classification both supports and extends the use of the metaphor of intelligent agents for designing agents.

We distinguish four types of rules:

- the class $\mathcal{F}$ of *failure rules*,
- the class $\mathcal{R}$ of *reactive rules*,
- the class $\mathcal{M}$ of *plan rules*, and

- the class $\mathcal{O}$ of *optimisation rules.*

According to their purpose, we can impose an intuitive and logical order on these classes by: $\mathcal{F} > \mathcal{R}$, $\mathcal{R} > \mathcal{M}$, $\mathcal{M} > \mathcal{O}$. Note that this order is total.

The intuition behind this classification and ordering is explained by the priority we associate with each of the tasks involved: failure handling, showing responsive behaviour, planning and optimising current plans. We believe that the classification and the priorities that we assigned to the different classes of rules is a natural one, although, of course, alternative classifications and orderings are possible.

The highest priority is assigned to the class of failure rules. Failure rules are of the form $\pi_h \leftarrow \varphi \mid \pi_b$ where π_h is any semi-goal except for an achievement goal. The priority assigned to failure rules is based on the purpose of avoiding failure, or cleaning up after failure. The idea to use rules for failure handling is one of the main motivations for extending the usual plan rules to our practical reasoning rules. Failure rules revise the means to achieve a goal of an agent. The reason for this revision typically is that the means to achieve a current goal of the agent are not suitable in the current situation the agent is in.

In case a failure has occurred or will occur, there basically exist two strategies to deal with the failure. The agent may drop its current goal because of the failure, since it is no longer feasible or the costs to achieve it in some other way are too high. Or, alternatively, the agent may substitute some alternative means to deal with the failure situation and try to achieve the goal in some other way. The high priority assigned to this type of rule is reasonable given that an agent needs to avoid any behaviour which could potentially lead to catastrophic consequences.

The class of reactive rules is assigned second-highest priority. Reactive rules are rules with an empty head of the form $\leftarrow \varphi \mid \pi$. The application of reactive rules does not depend on the current goals of an agent but only on the current beliefs of an agent. Reactive rules specify a plan of action in the body of the rule which is required to respond appropriately to a type of situation that is represented by the guard of the rule. For example, a reactive rule could be used to remind the user of an activity which is scheduled in the near future by triggering an inform_user action just before the time the activity is scheduled. Typically, these responses are time critical in the sense that delays in the response makes it redundant and even may lead to failure. The requirement of time-critical response explains the high priority we have assigned to this type of rule.

After avoiding behaviour which leads to failure and reacting to urgent

situations, the agent should use its time to find the appropriate means to achieve its achievement goals. Agents use plan rules of the form $p(\vec{t}) \leftarrow \varphi \mid \pi$ for this purpose. The lower priority we have assigned to plan rules reflects our assumption that it is very important to avoid failure and be reactive to particular events, and that only after this has been accomplished the agent should worry about how to achieve its achievement goals. Nevertheless, planning is a natural and necessary mode of functioning of an agent. Plan rules provide a plan in the body of the rule to achieve the achievement goal in the head of the rule.

The lowest priority is assigned to optimisation rules. These rules are of the form $\pi_h \leftarrow \varphi \mid \pi_b$ where π_h is a semi-goal except for an achievement goal. In this case the *means* to achieve a goal are found to be suboptimal and a more optimal way of achieving the same goal is possible. The purpose of these rules is to achieve a better performance of an agent. Without these types of rules presumably nothing would go wrong, but more costs are induced by the agent's current plan to achieve a goal than necessary. If the agent has enough time, these rules serve their purpose and the agent could try to use them. Because optimisation rules have the same form as failure rules, obviously, they cannot be distinguished syntactically.

The classification and ordering we proposed is a natural one, we believe, and highlights the close correspondence between the syntactic structure of rules and the use of these rules. However, the classification is just one classification among possible others, and is not imposed by the programming language itself. Moreover, although it is very natural to provide a classification which is closely related to the syntactic structure of rules, there does not seem to be a strict one-to-one correspondence of the form of a rule and its purpose. However, we believe that in practice our classification is useful.

5 Intelligent Agents

Intelligent agents in 3APL are entities which represent their environment by means of their beliefs, control this environment by means of actions which are part of their goals or plans, and manipulate their goals using practical reasoning rules. Goals keep the representation of the environment up to date by performing belief updates. The dynamic components of an agent are its beliefs and goals. During the operation of an agent these are the only components which can change. The set of practical reasoning rules associated with an agent does not change during the operation of an agent

[2]. The expertise of an agent provided by a set of basic actions also remains fixed during the lifetime of an agent. This informal discussion motivates a separation of the static and dynamic part of the agent, which is provided by the following definition of the *mental state* of an agent.

Definition 5.1 *(mental state)*
A *mental state* is a pair $\langle \Pi, \sigma \rangle$, where

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

Note that we do not allow any goal variables in the mental state of an agent, but only allow goals from Goal which do not contain occurrences of goal variables.

Convention 5.1 We use Π to denote a goal base, and σ to denote a belief base. We use Γ to denote a set of practical reasoning rules, also called a *PR-base*.

To program an agent means to specify its mental state, and to write a set of practical reasoning rules. An agent may have several different goals at the same time. These goals are executed in parallel. The concept of an intelligent agent is formalised next.

Definition 5.2 *(intelligent agent)*
An *intelligent agent* is a triple $\langle \Pi, \sigma, \Gamma \rangle$ where

- Π is a goal base,
- σ is a belief base, and
- Γ is a PR-base.

Summarising, 3APL is a combination of imperative and logic programming. Whereas imperative programming constructs are used to program

[2]Although this may seem restrictive, dynamically changing the practical reasoning abilities of an agent in a sensible way seems to require facilities which are not easily provided by introducing new constructs into the agent programming language; however, we will make a suggestion related to this issue concerning the introduction of new practical reasoning rules by means of a *plan* action in the meta language for programming control structures which is introduced below.

the usual flow of control from imperative programming [3] and update the current beliefs of the agent by executing basic actions, logic programming implements the querying of the belief base of the agent and the parameter mechanism of the language based on computing bindings for variables.

In the agent programming language 3APL three conceptual levels can be distinguished: beliefs, goals, and practical reasoning rules. At the most basic level, the beliefs of an agent represent the current situation from the agent's point of view. At the second level, the execution of goals operate on the belief base of an agent by adding and deleting information. At the third level, practical reasoning rules supply the agent with reflective capabilities to modify its goals. This cleanly separates the different types of updating. From a more traditional perspective, the beliefs of the agent correspond to the state of the system and the goal base of an agent represents the program which is being executed. Because of the reflective capabilities of agents to modify their goals in arbitrary ways by means of rules, however, agents are not just programs in the traditional sense, but are *self-modifying* programs. This is a distinguishing feature of intelligent agents in the agent language 3APL.

6 Operational Semantics

The dynamics of an agent corresponds to changes in the mental state of that agent. In this section, we provide a formal semantics which formalises these dynamical aspects of an agent. The semantics specifies how the operation of an agent affects the mental state of that agent. The semantics we use in this paper is an *operational semantics* defined by means of transition systems. Operational semantics provides a constructive approach to semantics, in contrast with a denotational semantics which provides a more abstract, mathematical type of semantics.

6.1 Transition Systems

Transition systems are a means to define the *operational semantics* of a programming language ([18]). A *transition system* consists of a set of derivation rules for deriving transitions which are associated with an agent. A transition corresponds to a *single computation step*. Such derivation rules are also

[3] Although this is not implied by the operational semantics given below, the execution model from imperative programming does not involve backtracking. The execution model of 3APL agents, therefore, also does not involve backtracking.

called *transition rules*. The axioms, premises and conclusions in a transition system are transitions or computation steps. A set of transition rules can be viewed as an inductive definition of a *transition relation* $\longrightarrow$. The relation $\longrightarrow$ defined by a transition system is the smallest relation which contains all the axioms of the system and all conclusions which are derivable by using these axioms. This transition relation specifies the possible computation steps of an agent.

A transition relation is a relation on so-called *configurations*. A 3APL configuration consists of three components. The first two components correspond to the current mental state of an agent. The first component consists of the the *goal base* of the agent and the second component consists of the *belief base* of the agent. The third component is a *substitution* which is used to store values or bindings associated with first order variables. This component is updated during a computation by tests which are used to compute these bindings. A 3APL configuration thus is a triple $\langle \Pi, \sigma, \theta \rangle$ where the Π and σ make up the current mental state of the agent, and θ is a substitution which contains a record of the computed bindings for variables. Because practical reasoning rules remain fixed during computation we do not include these rules in the configuration.

Now, the general format of a transition rule consists of a set of premises which are transitions and a conclusion derivable from these premises which also is a transition; moreover, auxiliary conditions may be used to fix a set of transitions. Two distinct types of transitions are defined below. The first type of transitions defines what it means to execute a single goal given the current belief base of an agent and the current bindings for variables. Execution defined by these transitions is called *execution at the goal level*. The second type of transitions is defined in terms of the first type and defines what it means to execute an agent. Execution defined by these transitions is called *execution at the agent level*.

6.2 Execution at the Agent Level

Although agent execution is defined in terms of goal execution, we first define a transition rule for agent execution, which shows how to reduce a computation step of an agent to a computation step of a single goal. At the agent level, the set of goals in the goal base of an agent are executed in parallel. The parallel execution of multiple goals is modeled by the interleaving of the computation steps of the different goals [4]. Because an agent

[4] An interleaving semantics for the parallel execution of goals provides a useful abstraction for modeling parallel execution of goals in practice. An interleaving semantics

executes multiple goals in parallel, an agent is a multi-threaded system. The goals can communicate through shared variables. Moreover, because they operate on a single belief base the beliefs of an agent can also be used to communicate information between goals.

Convention 6.1 We use $V \subseteq \mathsf{Var}$ to denote an arbitrary set of variables.

Definition 6.1 *(agent level execution)*
Let $\Pi = \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\} \subseteq \mathsf{Goal}$, θ, θ' be ground substitutions, and $V = \mathsf{Free}(\Pi)$. Then:

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

The semantics of an agent is defined by this transition rule in terms of the semantics of a single goal π_i, the goal which is selected for execution. The effects of the computation step of this goal on the beliefs and the substitution are transferred to the conclusion and moreover the goal π_i in the old goal base is updated to π_i', which is the remaining part of π_i after the computation step.

The presence of the set V of first order variables in the premise of the transition rule is explained as follows. Because the application of practical reasoning rules may introduce new occurrences of variables into the goal base, as we will see below, we have to take care that no new (implicit) bindings are created between variables. For this reason, the set of all free variables in the goal base is a parameter of the transition in the premise, which is used to prevent the introduction of any new implicit bindings in the transition rule for the application of practical reasoning rules.

6.3 Execution at the Goal Level

In this section, we define execution at the goal level. The transition rules below define possible computation steps for a single goal given a current belief base and current set of bindings (substitution). Transition rules which specify the semantics of basic actions, test goals, sequential composition, and nondeterministic choice are provided.

requires that an implementation of parallelism in the language by true concurrent execution of goals also provides a strategy for ruling out any possible conflicts which might occur if two or more goals simultaneously try to access the same data structure (update a particular belief or variable, in our case). However, an interleaving semantics abstracts from such implementation issues.

Convention 6.2 We use E to denote successful termination, and identify $E;\ \pi$ with π. Moreover, $\Pi \cup \{E\}$ is identified with Π.

The basic action repertoire of an agent defines the skills or expertise of that agent. As discussed above, basic actions are updates on the belief base of an agent. The semantics for these actions is a parameter of the transition semantics. We did not specify a fixed set of actions as part of the language 3APL, but allow the programmer to select and define any set of actions which are most suitable to him. Like the knowledge representation language, the set of basic actions thus is not fixed by the programming language itself. The semantics of basic actions is given by a *transition function*. A transition function for basic actions is a mapping from a basic action and a belief base (a sentence from $\mathcal{L}$) to a new belief base.

Definition 6.2 *(semantics of basic actions)*
A *transition function* $\mathcal{T}$ is a (partial) function of type : $\mathsf{Bact} \times \mathcal{L} \rightarrow \mathcal{L}$.

For example, the semantics of the basic action $\mathsf{insagenda}(Act, Time, Dur, Loc)$ for a belief base σ (with a closed world assumption associated with *agenda*) of the form

$$\begin{array}{l}\forall\, A, T, D, L(agenda(A, T, D, L) \leftrightarrow \\ ((A = a1 \wedge T = t1 \wedge D = d1 \wedge L = l1) \vee \ldots \vee \\ (A = aN \wedge T = tN \wedge D = dN \wedge L = lN))) \wedge \ldots\end{array}$$

can be defined by $\mathcal{T}(\mathsf{insagenda}(Act, Time, Dur, Loc), \sigma) = \sigma'$ where

$$\begin{array}{l}\sigma' = \\ \forall\, A, T, D, L(agenda(A, T, D, L) \leftrightarrow \\ ((A = a1 \wedge T = t1 \wedge D = d1 \wedge L = l1) \vee \ldots \vee \\ (A = aN \wedge T = tN \wedge D = dN \wedge L = lN) \vee \\ (A = Act \wedge T = Time \wedge D = Dur \wedge L = Loc))) \wedge \ldots\end{array}$$

for all ground instances of the variables $Act, Time, Dur, Loc$ such that σ' is consistent; for all belief bases σ'' equivalent to σ, define

$$\mathcal{T}(\mathsf{insagenda}(Act, Time, Dur, Loc), \sigma'') = \sigma'$$

; otherwise, $\mathcal{T}$ is undefined. Informally, this definition specifies that the insagenda action expands the belief base with a new belief concerning the agenda if this is consistent and otherwise the action is not enabled.

A number of constraints are imposed on transition functions. First of all, the update executed by a basic action on logically equivalent belief bases

should result in logically equivalent belief bases again. That is, if σ and σ' are logically equivalent, then $\mathcal{T}(a, \sigma)$ and $\mathcal{T}(a, \sigma')$ should also be logically equivalent for any basic action a (if $\mathcal{T}$ is defined for a, σ and σ'). Second, we assume that the transition function is only defined for ground basic actions. The reason for this is that it is not clear what the meaning of non-ground basic actions is supposed to be. For example, what does it mean to execute $\mathsf{insagenda}(Act, Time, 1:00, utrecht)$? And third, we do not allow updates on functions. For example, if $f(a) = b$ is a definition of a function f on argument a in the belief base, then no basic action is allowed to change this definition, into, for example, $f(a) = c$ where $c \neq b$. This constraint can be relaxed, but we do not provide the details here.

Given a transition function $\mathcal{T}$ which specifies the update performed by a basic action, we can now define the semantics of basic actions. The execution of a basic action $\mathsf{a}(\vec{t})$ amounts to updating the beliefs in accordance with the transition function. The substitution θ which contains the current record of bindings for variables computed so far is used to instantiate free variables in the basic action $\mathsf{a}(\vec{t})$. The execution of a basic action does not compute any new bindings for variables and therefore does not change the substitution θ.

Definition 6.3 *(transition rule for basic actions)*

$$\frac{\mathcal{T}(\mathsf{a}(\vec{t})\theta, \sigma) = \sigma'}{\langle \mathsf{a}(\vec{t}), \sigma, \theta \rangle_V \longrightarrow \langle E, \sigma', \theta \rangle}$$

A test goal $\varphi?$ computes bindings for the free variables in the condition φ which is being tested. It is enabled, i.e. can be executed, only if some instance of the condition φ is entailed by the current beliefs (otherwise, nothing happens). The bindings computed are stored for later reference in a substitution γ and are combined with the bindings θ which already have been computed. A test goal $\varphi?$ can *initialise* some of the free variables in the condition φ *for which no bindings have been computed yet.* The bindings θ which already have been computed are used to instantiate free variables in the test and therefore cannot be updated by the test. Moreover, the bindings $x = t$ computed are required to be ground, i.e. t is required to be ground. A test is evaluated relative to the current belief base. From a logic programming perspective, the belief base can be viewed as a logic program which is used to compute the bindings. A test does not change the beliefs of the agent.

Definition 6.4 *(transition rule for tests)*
Let γ be a ground substitution such that $dom(\gamma) \subseteq \mathsf{Free}(\varphi\theta)$.

$$\frac{\sigma \models \varphi\theta\gamma}{\langle \varphi?, \sigma, \theta \rangle_V \longrightarrow \langle E, \sigma, \theta\gamma \rangle}$$

For example, in case the current belief base contains the proposition $agenda(meeting, 10:00, 1:00, utrecht)$ and no binding has been established yet for the variable $Time$ the test $agenda(meeting, Time, 1:00, utrecht)?$ could return the binding $Time = 10:00$. It would return this binding if there is only one meeting of 1:00 in Utrecht in the agenda, otherwise also alternative bindings could be returned. (That is, there may be $\gamma \neq \gamma'$ such that both $\sigma \models \varphi\theta\gamma$ and $\sigma \models \varphi\theta\gamma'$; in that case, one of the the possible substitutions is nondeterministically chosen.)

The bindings computed by a test are used in the remaining computation. We illustrate this by an example. Consider the sequential goal $location(Loc, 10:00)?;\ agenda(Act, 10:00, Dur, Loc)?$ and assume the location in the belief base at 10:00 is Utrecht since a meeting of 1:00 is scheduled at that time and no bindings have been computed yet. Then the first step is to execute the test $location(Loc, 10:00)?$ which returns a binding $Loc = utrecht$. The remaining goal consists of the test $agenda(Act, 10:00, Dur, Loc)?$. Because a binding for the variable Loc has already been established, this binding is used to instantiate the variable Loc resulting in $agenda(Act, 10:00, Dur, utrecht)?$. Evaluating this test with respect to the current belief base then results in bindings $Act = meeting$ and $Dur = 1:00$. Note that the second test does not compute a (new) binding for the variable Loc.

Essentially, this example explains the execution of a sequential goal. Simply execute the first part of the goal, record any changes to either the belief base or the current store of bindings, and continue executing the remainder of the goal. More formally, the transition rule below defines the semantics of sequential composition.

Definition 6.5 *(execution rule for sequential composition)*

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

The execution of a nondeterministic choice goal amounts to selecting one of the subgoals that is enabled, i.e. can be executed, execute this goal, and drop the other goal. The semantics of nondeterministic choice is defined by

two transition rules, one for choosing the left branch of the choice goal and one for choosing the right branch of the choice goal.

Definition 6.6 *(execution rules for non-deterministic choice)*

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

6.4 The Application of Practical Reasoning Rules

Practical reasoning rules operate on the goals of the agent. A practical reasoning rule $\pi_h \leftarrow \varphi \mid \pi_b$ is applicable if the head of the rule unifies with a (subgoal of a) current goal of the agent and the guard is entailed by the current beliefs. Two transition rules are required to define the application of rules due to the different nature of rules with empty head and rules which do not have an empty head. The semantics of rules with empty body can be viewed as a special case of the semantics of rules with nonempty bodies, where the goal which unifies with the head of such a rule simply is dropped and this is interpreted as successful termination.

The notion of a *variant* plays an important role in the semantics of rules. An expression e is a variant of another expression e' in case e can be obtained from e' by renaming of variables (for a formal definition cf. [17]).

Definition 6.7 *(PR-rule application)*
Let η be a most general unifier for π and π_h such that $\pi\theta = \pi_h\eta$, and θ, γ be ground substitutions such that $dom(\gamma) \subseteq \mathsf{Free}(\varphi\eta)$.

$$\frac{\sigma \models \varphi\eta\gamma}{\langle \pi, \sigma, \theta \rangle_V \longrightarrow \langle \pi_b\eta, \sigma, \theta\gamma \rangle}$$

where $\pi_h \leftarrow \varphi \mid \pi_b$ is a *variant* of a PR-rule in the PR-base Γ of the agent such that no free variables in the rule occur in either V or $dom(\theta)$.

The application of a PR-rule defined by the transition rule is explained as follows. The first step is to check whether or not the current (sub)goal π instantiated with the current bindings θ is an instance of the head π_h of the rule. If this is the case, there is a most general unifier η with bindings for *all* goal variables in π_h (because π does not contain goal variables) and possibly some bindings for first order variables. The bindings for the first order variables may represent input values supplied to the body of the rule but may also rename variables. The computation of the substitution η is

based on pattern-matching. The second step is to check whether the guard is entailed by the current beliefs. The evaluation of a guard is analogous to the evaluation of a test, but in the case of a guard the input values which are stored in the substitution η are used to instantiate the guard. The evaluation of the guard may compute new bindings for free variables in $\varphi\eta$. Finally, the original goal π is replaced by the body of the rule which has been instantiated with the bindings from η. Since η can only contain bindings for first order variables which do not occur in the goal base of the agent (the rule is a variant), we do not add these bindings to the current set of bindings θ; the substitution γ resulting from the evaluation of the guard, however, may contain bindings for variables in the goal base since η may have renamed variables and therefore γ is added to the set of bindings θ.

We remark here that the restriction on goal variables (cf. definition 3.11) such that all goal variables in the body of a rule must also occur in the head of the rule implies that no goal variables are introduced into the goal base of an agent, because the most general unifier η binds all goal variables to goals.

The transition rule for PR-rule application only specifies *how* to apply a rule to a goal. It does not specify, however, *which* applicable rule - in case there is more than one - should be applied. It also does not determine which γ is applied (in case there are $\gamma \neq \gamma'$ such that $\sigma \models \phi\eta\gamma$ and $\sigma \models \phi\eta\gamma'$).

Special care needs to be taken to avoid introducing any new implicit bindings between variables by applying a rule and replacing a goal by the body of a rule. The set V is used to prevent this. V consists of all the free variables of the goal base. Because in the transition rule a variant of a practical reasoning rule is used in which new variables have been substituted for the free variables, which do not occur in V nor in the current set of bindings θ, no new implicit bindings can be created.

Example 6.8 We illustrate what can go wrong if variables in a rule are not renamed appropriately. Recall the rule for scheduling an activity from example 3.13:

$$\begin{aligned} &\mathit{schedule}(\mathit{Act}, \mathit{Time}, \mathit{Dur}, \mathit{Loc}) \leftarrow \mathit{location}(\mathit{FromLoc}, \mathit{Time}) \ \mid \\ &\quad \mathit{transport}(\mathit{Means}, \mathit{FromLoc}, \mathit{Loc}, \mathit{DurTrans})?; \\ &\quad \mathsf{insagenda}(\mathit{Means}, \mathit{Time} - \mathit{DurTrans}, \mathit{DurTrans}, \mathit{FromLoc}); \\ &\quad \mathsf{insagenda}(\mathit{Act}, \mathit{Time}, \mathit{Dur}, \mathit{Loc}) \end{aligned}$$

Now suppose one of the current goals of the agent is:

$$\begin{aligned} &\mathit{schedule}(\mathit{meeting}, 10:00, 1:00, \mathit{utrecht}); \\ &\mathit{transport}(\mathit{Means}, \mathit{utrecht}, \mathit{amsterdam}, \mathit{DurTrans})? \end{aligned}$$

and among other things the agent believes: $location(utrecht, 10:00)$.

In that case, if we do not rename the variables in the plan rule for scheduling an activity, then by applying this rule to the goal we would end up with the following new plan, in which the variable *Fromloc* has been instantiated with the computed binding $Fromloc = utrecht$:

$transport(Means, utrecht, utrecht, DurTrans)?;$
$\mathsf{insagenda}(Means, 10:00 - DurTrans, DurTrans, utrecht);$
$\mathsf{insagenda}(meeting, 10:00, 1:00, utrecht);$
$transport(Means, utrecht, amsterdam, DurTrans)?$

In the resulting plan, an implicit binding is introduced between the first and second occurrence of the variable *Means* and the first and second occurrence of the variable *DurTrans*. However, the type of transportation required to get from Utrecht to Utrecht (none) and the time it takes (none) are obviously different from the type of transportation required to get from Utrecht to Amsterdam (train, bus, etc.) and the time it takes. No link between these different entities was intended, and would not have been introduced if the variables in the rule would have been renamed appropriately.

Example 6.9 We illustrate the application of the practical reasoning rule from example 3.15 to deal with a failure of the scheduling of an activity. For ease of exposition, we repeat the practical reasoning rule to deal with a failure to schedule the time needed for traveling first.

$\mathsf{insagenda}(Act, Time, Dur, Loc);\ X \leftarrow$
$agenda(A1, T1, D1, L1) \wedge$
$((T1 \leq Time < T1 + D1) \vee (Time < T1 < Time + Dur)) \mid$
$\quad \mathsf{inform\_user}(\mathtt{unsuccessful}, Act, Time, Dur, Loc)$

To illustrate the application of this failure rule, suppose that the agent has applied the plan rule for scheduling an activity, already has executed the test in that plan, and ended up with the remaining goal [5]:

$\mathsf{insagenda}(train, 10:15, 0:45, utrecht));$
$\mathsf{insagenda}(meeting, 11:00, 1:00, amsterdam)$

To create a situation where a conflict arises in scheduling the time needed for travelling, also suppose that the agent believes that:

$agenda(meeting, 10:00, 0:30, amsterdam)$

[5] Actually, this goal cannot be a result of applying the scheduling plan, but is an instance of such a goal where variables have been instantiated by the current set of bindings.

It is clear that the agent cannot insert the travelling time in the agenda because of the meeting already scheduled at 10:00 which takes half an hour. To see how the failure rule deals with this, first note that the head of the failure rule can be unified with the (remaining part of the) scheduling plan. The most general unifier η is $\{Act = train, Time = 10:15, Dur = 0:45, Loc = utrecht, X/\mathsf{insagenda}(meeting, 11:00, 1:00, amsterdam)\}$. Note that this unifier also contains a binding for the goal variable X. The second step is to check whether an instance of the guard is entailed by the agent's beliefs. It is not difficult to see that the agent's beliefs entail

$$\begin{array}{l} agenda(meeting, 10:00, 0:30, amsterdam) \wedge \\ ((10:00 \leq 10:15 < 10:00 + 0:30) \vee \\ \quad (10:15 < 10:00 < 10:15 + 0:45)) \end{array}$$

which is an instance of the guard (note that variable *Time* and *Dur* are bound by the unifier η). The guard thus returns a substitution γ which binds the variables $A1, T1, D1, L1$ in the guard, where

$$\gamma = \{A1 = meeting, T1 = 10:00, D1 = 0:30, L1 = amsterdam\}$$

Therefore, the failure rule is applicable and in the third and last step the original goal consisting of the two insagenda actions is replaced by the body of the rule, where free variables are instantiated by η and γ is added to the current set of bindings in the configuration. The resulting goal is:

$$\mathsf{inform\_user}(\mathtt{unsuccessful}, train, 10:15, 0:45, utrecht)$$

which informs the user that the agent was unable to reserve the travelling time in the agenda. (Note that in this example it is unnecessary to rename any variables in the failure rule.)

The last transition rule specifies the semantics of reactive rules, i.e. rules with empty head. The details are analogous to that of the transition rule for rules with nonempty head. The main difference is that no unification with a current (sub)goal of the agent is involved and reactive rules simply add new goals to the current goal base. The semantics of this rule therefore needs to be specified at the agent level (with respect to a goal base) and not at the goal level (with respect to a single goal). Also, a slight difference arises because the bindings γ retrieved by the guard of a reactive rule are bindings for variables which only occur in the reactive rule itself, and therefore γ does not have to be added to the current bindings θ but can be used to instantiate the variables in the body of the rule instead.

Definition 6.10 *(transition rule for the application of reactive rules)*
Let Π be a goal base and θ, γ be ground substitutions such that $dom(\gamma) \subseteq \mathsf{Free}(\varphi)$.

$$\frac{\sigma \models \varphi\gamma}{\langle \Pi, \sigma, \theta \rangle \longrightarrow \langle \Pi \cup \{\pi\gamma\}, \sigma, \theta \rangle}$$

where $\leftarrow \varphi \mid \pi_b$ is a *variant* of a PR-rule in the PR-base Γ of the agent such that no free variables in the rule occur in the goal base Π or $dom(\theta)$.

6.5 Auxiliary Notions

In the sequel, it will be convenient to have some notation available to distinguish between computation steps or transitions resulting from the application of a practical reasoning rule and transitions resulting from the execution of either a basic action or test goal. This distinction can be made by distinguishing two types of derivations in the transition system. Transitions which are derivable by means of the transition rules for rule application (both rules for the application of PR-rules with empty and nonempty head) and all other transition rules except for the transition rules for basic actions and tests correspond to transitions involving the application of a PR-rule. On the other hand, transitions which are derivable by means of the transition rules for basic actions and tests and all other rules except for the transition rules for rule application correspond to transitions involving either the execution of a basic action or a test. We introduce some convenient notation to capture this distinction.

Notation 6.11 Let σ, σ' be belief bases, θ, θ' be substitutions, and $\Pi = \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_n\}$, and $\Pi' = \{\pi_0, \ldots, \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots, \pi_n\}$ be goal bases. Note that the goal base Π' results from substituting π_i' for π_i in the goal base Π.
Now we introduce the following notation.

- $\langle \Pi, \sigma, \theta \rangle \stackrel{\pi_i, \pi_i'}{\longrightarrow} \langle \Pi', \sigma', \theta' \rangle$ is used to express that the transition $\langle \Pi, \sigma, \theta \rangle \longrightarrow \langle \Pi', \sigma', \theta' \rangle$ can be derived in the transition system without using the transition rules for PR-rule application,
- $\langle \Pi, \sigma \rangle \stackrel{\pi, \pi'}{\longrightarrow}$ is used to express that there are Π', σ', θ' such that $\langle \Pi, \sigma, \theta \rangle \stackrel{\pi, \pi'}{\longrightarrow} \langle \Pi', \sigma', \theta' \rangle$,

- $\langle \Pi, \sigma, \theta \rangle \stackrel{\pi_i, \rho, \pi_i'}{\longrightarrow} \langle \Pi', \sigma, \theta' \rangle$ is used to express that the transition $\langle \Pi, \sigma, \theta \rangle \longrightarrow \langle \Pi', \sigma, \theta' \rangle$ can be derived in the transition system without using the transition rules for basic actions or tests,

- $\langle \Pi, \sigma, \theta \rangle \stackrel{\pi, \rho, \pi'}{\longrightarrow}$ is used to express the fact that there are Π', θ' such that $\langle \Pi, \sigma \rangle \stackrel{\pi, \rho, \pi'}{\longrightarrow} \langle \Pi', \sigma, \theta' \rangle$.

The fact that a transition $\langle \Pi, \sigma, \theta \rangle \longrightarrow \langle \Pi', \sigma', \theta' \rangle$ can be derived without using the transition rules for rule application indicates that there is a goal π in Π which can be executed by either executing a basic action or a test. The primed goal π_i' in the notation introduced corresponds to the goal which results from executing π_i. The fact that a transition $\langle \Pi, \sigma, \theta \rangle \longrightarrow \langle \Pi', \sigma, \theta' \rangle$ can be derived without using the transition rules for basic actions or tests indicates that there is a practical reasoning rule ρ which can be applied to a goal in the goal base Π. The notation we have introduced explicitly mentions which goal the rule is applied to (π_i), the practical reasoning rule that is being applied (ρ) and the goal which results from applying the rule (π_i'). One final remark is in place. We use the notation $C \stackrel{\pi, \rho, \pi'}{\longrightarrow}$ also for reactive rules where no goal π from the current goal base is involved, and in that case we could also write $C \stackrel{\rho, \pi'}{\longrightarrow}$. The discussion motivates the following definitions.

Definition 6.12 *(executable, applicable)*
Let $\pi \in \mathsf{Goal}, \rho \in \mathsf{Rule}$, and C be a 3APL configuration.

- π is *executable (or enabled) in* C if there is a π' such that $C \stackrel{\pi, \pi'}{\longrightarrow}$,

- a rule ρ is *applicable to goal* π *in* C if there is a π' such that $C \stackrel{\pi, \rho, \pi'}{\longrightarrow}$.

6.6 Computations and Observables

The transition system defines in a natural way the possible computations of an agent. A *computation* of an agent is a finite or infinite sequence of mental states $\langle \Pi_0, \sigma_0, \theta_0 \rangle, \langle \Pi_1, \sigma_1, \theta_1 \rangle, \langle \Pi_2, \sigma_2, \theta_2 \rangle, \ldots$, such that for all i we have $\langle \Pi_i, \sigma_i, \theta_i \rangle \longrightarrow \langle \Pi_{i+1}, \sigma_{i+1}, \theta_{i+1} \rangle$ and such that $\langle \Pi_0, \sigma_0 \rangle$ is the initial mental state of the agent.

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 actions corresponding to single transitions in a computation, for example for planning (cf. [7]).

7 Selection Mechanisms and Control Structures

A multi-threaded, rule-based language such as 3APL induces two issues, namely which goal should be executed and which rule should be applied. Any implementation of the language has to deal with how to reduce the inherent non-determinism in the semantics of the language. For example, the transition rule for PR-rule application does not tell us which applicable rule to select for application. The main purpose of a control structure for an agent language is to determine which goals to deal with first and which rules to use during the lifetime of an agent.

There are two basic assumptions that can be made about a control structure. On the one hand one could argue that the specifics of a control structure are of no concern during the design of an agent and therefore should be hidden from the programmer. This is called the *black-box assumption*. On the other hand, one could argue that the control structure is relevant for the design of programs, and at least some aspects of the control structure should be made explicit and readily available to the programmer. This assumption is called the *glass-box assumption* (cf. discussions of a similar issue concerning constraint programming languages in [15]).

Although a black-box approach is suitable for most languages, we argue that for agent languages a glass-box approach is more appropriate. We think that in the case of agent programming the problem is better seen as a semantic problem. The reason lies in the aim of agent programming. In our view, the aim of agent languages is to provide practical tools and support for programming agents using the metaphor of intelligent agents. In order for the programming language and its control structure to support the use of this metaphor, it is important that the formal semantics fits this metaphor as close as possible if the metaphor is to be of any use. For this reason, we think it is important to clarify and explicate the selection mechanisms agents use. The specification of these mechanisms is relevant for the design and correctness of agents if agents are designed using the metaphor of intelligent agents.

As a consequence, we need to specify explicitly how an agent deals with the selection of goals and rules. As part of a solution to this problem, we introduced a classification and ordering on types of rules earlier which can be used in a selection mechanism for rules. However, neither this classification nor the ordering are part of the specification of the operational semantics of the agent language. To incorporate the concept of priority into our framework, in the next section we introduce a separate semantics for control structures of agent languages.

8 A Language for Programming Control Structures

There seem to be two ways to formally achieve our aim of specifying selection mechanisms. First, we could incorporate the basic features of agents and the selection mechanisms controlling these features into the same semantic framework. This would require us to extend the operational semantics of the agent language. Secondly, we could separate the basic features of agents and the selection mechanisms and specify them in two different semantic systems. This approach supposes that it is possible in the semantic system which defines the control structure to refer to the semantic objects in the semantics of the agent language and introduce a distinction between an object and meta level semantics. We have chosen for the second option (cf. also [13]). The main reason for this choice is that the specification of the selection mechanisms at a meta level gives us two independent systems. This gives us the usual advantages associated with the modularity of two different systems. For instance, by separating the two semantic systems *any* suitable agent language specified at the object level can be plugged in in *any* control structure specified at the meta level. This choice also allows for more freedom in specifying the selection mechanisms of agent languages. As a consequence, it becomes possible to specify the control structures of different agent languages and to compare their respective control mechanisms.

The main idea, thus, is to separate the semantic specification of the agent language and its control structure. This approach calls for a distinction between an object and a meta level. At the object level, the transition system which was introduced earlier defines the semantics of the agent language. At the meta level, a second transition system is introduced to define the semantics of the meta language for programming control structures. The latter is called a *meta transition system* and includes features for referring to the object agent language. The meta language will include a number of expressive operators for programming control structures for the object (agent) language.

The design of the meta language is guided by the needs of a programmer of a control structure for agent languages. There are a number of issues a control structure must be able to deal with. First of all, we need the regular imperative constructs (or some other means for programming in general) to program control flow. Imperative constructs seem most suitable for the purpose of building control structures since these structures usually specify some sequential order of execution. Secondly, we need a number of *basic*

actions to select, apply rules, and execute goals. For this purpose, four actions are introduced: (1) an action for selecting an applicable rule, (2) an action for the application of a number of rules, (3) an action for selecting an enabled goal, and (4) an action for the execution of a set of goals. And thirdly, we need constructs to express a preference order over goals and rules. Also, a control structure needs to have the means to access relevant features of an agent, like its PR-base, and the current mental state of the agent.

Since the main issue that a control structure for agent languages has to deal with is the selection of goals and rules from *sets* of goals and rules, a *set-based* language seems to provide the right abstraction level to discuss the control structure for an agent language. Summarising, the meta language is an imperative, set-based language. It has terms for referring to the goals and rules of an agent. Furthermore, the meta-language includes assignment, four task-specific basic actions, and the usual operators from imperative programming.

8.1 Syntax of the Meta Language

The goal and rule terms of the meta language are used to access the goal base and rule base of an agent in the meta language. The terms refer to sets of goals or sets of rules. Therefore, the operators of the meta language for building complex terms are the usual set operators.

Definition 8.1 *(terms)*
Let Var_g with typical elements G, G' and Var_r with typical elements R, R' be countably infinite sets of variables.

- The set of goal terms $\mathfrak{T}_g$:
  - $\mathsf{Var}_g \subseteq \mathfrak{T}_g$,
  - $\varnothing, \Pi \in \mathfrak{T}_g$,
  - if $g_0, g_1 \in \mathfrak{T}_g$, then $g_0 \cap g_1, g_0 \cup g_1, g_0 - g_1 \in \mathfrak{T}_g$,
  - if $g \in \mathfrak{T}_g$, then $max_g(g) \in \mathfrak{T}_g$.
- The set of *rule terms* $\mathfrak{T}_r$:
  - $\mathsf{Var}_r \subseteq \mathfrak{T}_r$,
  - $\varnothing, \Gamma \in \mathfrak{T}_r$,
  - if $r_0, r_1 \in \mathfrak{T}_r$, then $r_0 \cap r_1, r_0 \cup r_1, r_0 - r_1 \in \mathfrak{T}_r$,
  - if $r \in \mathfrak{T}_r$, then $max_r(r) \in \mathfrak{T}_r$.

The goal and rule terms are the usual set terms, constructed from the set operators $\cap$, etc. $\varnothing$ is a constant denoting the empty set. We use capital letters G, G', R, R' to denote meta variables and lower case g, g', r, r' to denote arbitrary meta terms. The denotation of the rule term Γ is the non-dynamic set of rules of the agent. The denotation of the goal term Π is the dynamic goal base of the *current* mental state of the agent during execution. The use of Π and Γ in the meta language introduces some overloading of these symbols, because they are also used in the transition semantics. However, it should be clear from the context which use is intended. The set operators are extended with one more operator, the *max* function. The goal and rule term functions *max* return the maximal goals respectively rules in a given set under the goal and rule orderings.

Definition 8.2 *(meta statements)*
The set of statements $\mathfrak{S}$ is defined by:

- if $G \in \mathsf{Var}_g, g \in \mathfrak{T}_g$, then $G := g \in \mathfrak{S}$,
- if $R \in \mathsf{Var}_r, r \in \mathfrak{T}_r$, then $R := r \in \mathfrak{S}$,
- if $g, g' \in \mathfrak{T}_g$, then $g = g', g \neq g' \in \mathfrak{S}$,
- if $r, r' \in \mathfrak{T}_r$, then $r = r', r \neq r' \in \mathfrak{S}$,
- if $g \in \mathfrak{T}_g, G \in \mathsf{Var}_g$, then $selex(g, G) \in \mathfrak{S}$,
- if $G, G' \in \mathsf{Var}_g$, then $ex(G, G') \in \mathfrak{S}$,
- if $r \in \mathfrak{T}_r, g \in \mathfrak{T}_g, R \in \mathsf{Var}_r, G \in \mathsf{Var}_g$, then $selap(r, g, R, G) \in \mathfrak{S}$,
- if $R \in \mathsf{Var}_r$ and $G, G' \in \mathsf{Var}_g$, then $apply(R, G, G') \in \mathfrak{S}$,
- if $\beta, \beta' \in \mathfrak{S}$, then $\beta;\ \beta', \beta + \beta', \beta^* \in \mathfrak{S}$.

The meta language includes assignment of sets of goals or rules to goal or rule variables, tests for equality on the goal and rule terms, and the regular programming constructs for sequential composition, nondeterministic choice, and iteration. Specific for the programming of control structures, the meta language includes two actions *selex* and *selap* for selection and two actions *ex* and *apply* for respectively execution of goals and application of rules to goals. These four actions are calls to the object agent system to test and select, or to perform object transition steps corresponding to execution of goals or application of rules. The former two only perform tests

on the object level and do not change the mental state, the latter two also modify the object configuration, that is they actually execute the changes at the object level as specified in the object transition system. The first argument position of the actions *selex* and *ex* should be filled with an input term denoting the set of goals from which to select or execute. The second argument position should be substituted with an output variable. The first argument position of *selap* and *apply* needs to be filled with an input term denoting the set of rules from which to select a rule, in the first case, or to apply a rule, in the second case. The second argument position of *selap* and *apply* should be substituted with input terms denoting a set of goals to which the rules provided in the first argument should be applied. The third argument position (and fourth argument position of *selap*) should be instantiated with an output variable. This will be explained in more detail below.

Remark 8.3 If in some context a *planning system* is defined, then we could also introduce an action $plan(g, r, G, R)$ into the language for adding new plan rules to the rule base of an agent. (g, r are input terms denoting respectively the set of goals for which a plan must be constructed and the set of plan rules already available; in G the set of goals for which a plan has been found could be stored, and in R the set of plans found.) Note that in this case, the rule term Γ would no longer be a constant.

8.2 Semantics of the Meta Language

In this section the operational semantics of the meta language is defined. The transition relation of the meta transition system is denoted by $\Longrightarrow$.

8.2.1 Preliminaries

The transition relation $\Longrightarrow$ is a relation on meta configurations, which are pairs consisting of a meta statement and a meta state. Meta level states include the information about object level features which an agent control structure needs to be able to access. Among these features are the current object configuration $\langle \Pi, \sigma, \theta \rangle$, the rule base of the agent and the orderings on the goals and rules. The meta state also keeps track of the values of variables used in the meta program. This is formalised in the following definition.

Definition 8.4 *(meta state)*
A *meta-state* τ is a tuple $\langle \langle \Pi, \sigma, \theta \rangle, \Gamma, <_g, <_r, I \rangle$, where

- $\langle \Pi, \sigma, \theta \rangle$ is a *3APL configuration*,
- $\Gamma \subseteq \mathsf{Rule}$ is a set of *object rules*,
- $<_g$ is an ordering on the set of goals Goal of the agent language,
- $<_r$ is an ordering on the set of rules Rule of the agent language, and
- I is a *variable valuation* of type : $(\mathsf{Var}_g \rightarrow \wp(\mathsf{Goal})) \cup (\mathsf{Var}_r \rightarrow \wp(\mathsf{Rule}))$.

The orderings on the set of goals and rules in a meta-state are used to define the selection mechanisms for goals and rules of an agent. We assume these orderings do not change. Since the orderings are features of an agent, we could also incorporate them into the definition of agent. Note that there is nothing in the operational semantics of the meta language which determines that the classification and ordering we introduced earlier has to be used. Of course, we can use this classification, and instantiate the ordering on rules correspondingly.

Definition 8.5 *(meta configuration)*
A *meta-configuration* is a pair $\langle \beta, \tau \rangle$ where β is a meta statement and τ is an meta-state.

We also write a meta-configuration as a seven-tuple, by listing all the elements of a meta-state as elements of a meta-configuration. Note that a meta-configuration is the analogue of the 3APL configuration in the operational semantics for the meta language.

8.3 Semantics of Terms

Some means to use the orderings on goals and rules should be available to the programmer. The two function terms max_g and max_r provide these facilities.

Definition 8.6 *(max)*
Let Π be a set of goals, and Γ be a set of rules. Furthermore, let $<_g$ be an ordering on the set of *all* goals $\mathcal{L}^g$ and $<_r$ be an ordering on the set of *all* rules. Then:

$$\begin{aligned} max_g(\Pi) &= \{\pi \in \Pi \mid \text{there is } no\ \pi' \in \Pi \text{ such that } \pi' >_g \pi\}, \\ max_r(\Gamma) &= \{\rho \in \Gamma \mid \text{there is } no\ \rho' \in \Gamma \text{ such that } \rho' >_r \rho\}. \end{aligned}$$

Since the orderings on goals and rules and the set of rules itself are global properties of an agent which do not change, we will not always list them explicitly in a configuration or state.

A semantic function $[\![\cdot]\!]_\tau$ is defined for assigning meanings to terms. Since terms evaluate to sets, and there are two types of terms, the function has type : $(\mathfrak{T}_g \to \wp(\mathsf{Goal})) \cup (\mathfrak{T}_r \to \wp(\mathsf{Rule}))$.

Definition 8.7 *(semantics of terms)*
Let $\tau = \langle \Pi, \sigma, \theta, \Gamma, I \rangle$ be a meta-state. Then the interpretation function $[\![\cdot]\!]_\tau$ for goal and rule terms is defined by:

- On goal terms:
  - $[\![G]\!]_\tau = I(G)$, for $G \in \mathsf{Var}_g$,
  - $[\![\Pi]\!]_\tau = \Pi$,
  - $[\![\varnothing]\!]_\tau = \varnothing$,
  - $[\![g_0 \oplus g_1]\!]_\tau = [\![g_0]\!]_\tau \oplus [\![g_1]\!]_\tau$, for $\oplus \in \{\cap, \cup, -\}$,
  - $[\![max(g)]\!]_\tau = max([\![g]\!]_\tau)$.
- On rule terms:
  - $[\![R]\!]_\tau = I(R)$, for $R \in \mathsf{Var}_r$,
  - $[\![\Gamma]\!]_\tau = \Gamma$,
  - $[\![\varnothing]\!]_\tau = \varnothing$,
  - $[\![r_0 \oplus r_1]\!]_\tau = [\![r_0]\!]_\tau \oplus [\![r_1]\!]_\tau$, for $\oplus \in \{\cap, \cup, -\}$,
  - $[\![max(r)]\!]_\tau = max([\![r]\!]_\tau)$.

8.3.1 Execution Rules for Selection Actions

The selection action $selex(g, G)$ selects a goal which is executable in the current mental state from a given set of goals $[\![g]\!]$. The goal selected must be maximal with respect to the ordering on the set of all goals, i.e. a goal is not chosen if there is a goal in $[\![g]\!]$ which is executable and has higher priority with respect to the ordering on goals. The selection action returns the singleton set with the selected goal in the output variable G.

Definition 8.8 *(execution rule for selex)*

$$\frac{\langle \Pi, \sigma, \theta \rangle \xrightarrow{\pi, \pi'}}{\langle selex(g, G), \langle \Pi, \sigma, \theta \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma, \theta \rangle, I\{\{\pi\}/G\} \rangle}$$

such that

1. $\pi \in [\![g]\!]$,

2. there is *no* $\delta \in [\![g]\!]$ such that $\delta >_g \pi$ and $\langle \Pi, \sigma, \theta \rangle \xrightarrow{\delta, \delta'}$ for some $\delta' \in \mathsf{Goal}$.

If there is no executable goal in the set of goals denoted by g, then the selection action returns the empty set in G.

$$\frac{\langle \Pi, \sigma, \theta \rangle \not\xrightarrow{\pi, \pi'} \text{ for all } \pi \in [\![g]\!], \pi' \in \mathsf{Goal}}{\langle \mathit{selex}(g, G), \langle \Pi, \sigma, \theta \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma, \theta \rangle, I\{\varnothing / G\} \rangle}$$

The selection action $\mathit{selap}(r, g, R, G)$ selects a rule from $[\![r]\!]$ that is applicable to a goal from $[\![g]\!]$ in the current mental state. The action first selects a maximal goal from $[\![g]\!]$ to which a rule from $[\![r]\!]$ can be applied, and then selects a maximal rule from $[\![r]\!]$ which is applicable to that goal. This order of maximisation is explained by the fact that an agent is *goal-driven*, not *rule-driven*. An agent tries to achieve its most important goals, not to apply its best rules. The selected rule is returned as a singleton set in the output variable R and the goal with highest priority to which the rule can be applied is returned in G.

Definition 8.9 *(execution rule for selap)*

$$\frac{\langle \Pi, \sigma, \theta \rangle \xrightarrow{\pi, \rho, \pi'}}{\langle \mathit{selap}(r, g, R, G), \langle \Pi, \sigma, \theta \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma, \theta \rangle, I\{\{\rho\}/R, \{\pi\}/G\} \rangle}$$

provided that

1. $\rho \in [\![r]\!]$, $\pi \in [\![g]\!]$,

2. there is *no* $\delta >_g \pi$, $\delta \in [\![g]\!]$ such that:
   there is a $\rho' \in [\![r]\!]$ such that: $\langle \Pi, \sigma, \theta \rangle \xrightarrow{\delta, \rho', \delta'}$ for some $\delta' \in \mathsf{Goal}$,

3. there are *no* $\rho' >_r \rho$, $\rho' \in [\![r]\!]$ and goal π' such that $\langle \Pi, \sigma \rangle \xrightarrow{\pi, \rho', \pi'}$.

If there is no rule applicable to a goal in the denotation of r and g, then the empty set is returned in both R and G.

$$\frac{\langle \Pi, \sigma, \theta \rangle \not\xrightarrow{\pi, \rho, \pi'} \text{ for all } \pi \in [\![g]\!], \rho \in [\![r]\!], \pi' \in \mathsf{Goal}}{\langle \mathit{selap}(r, g, R, G), \langle \Pi, \sigma, \theta \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma, \theta \rangle, I\{\varnothing / R, \varnothing / G\} \rangle}$$

8.3.2 Execution Rules for *ex*

Informally, the meaning of the action $ex(G, G')$ is to (partly) execute as many goals from $[\![G]\!]$ from the current state as possible. The choice to execute *all* executable goals from $[\![G]\!]$ introduces a duality with the selection of *some* goal by the action *selex*. A maximal subset of $[\![G]\!]$ is chosen in such a way that each goal can be (partly) executed in sequence when the goals of the subset are ordered appropriately. Each of these goals is only partly executed, i.e. only one computation step as defined by the object agent transition system is made. The reason for partly executing a goal is that in general it is undecidable to check whether or not a program will halt, while it is reasonable to assume that it is possible to decide whether a basic action or test can be performed. Furthermore, the goals G are transformed by using basic action or test transition rules, and no rules are applied to a goal in G.

The semantics of the action $ex(G, G')$ is defined by iteration. $ex(G, G')$ is executed by arbitrarily choosing an executable goal from $[\![G]\!]$, deleting this goal from $[\![G]\!]$, executing the goal at object level, and returning the new goal in output variable G', until no goals from $[\![G]\!]$ can be executed including the case $[\![G]\!] = \varnothing$.

Definition 8.10 *(execution rules for ex)*

$$\frac{\langle \Pi, \sigma, \theta \rangle \xrightarrow{\pi, \pi'} \langle \Pi', \sigma', \theta' \rangle, \pi \in [\![G]\!]}{\begin{array}{c}\langle ex(G, G'), \langle \Pi, \sigma, \theta \rangle, I \rangle \Longrightarrow \\ \langle ex(G, G'), \langle \Pi', \sigma', \theta' \rangle, I\{[\![G]\!] - \{\pi\}/G, [\![G']\!] \cup \{\pi'\}/G'\} \rangle\end{array}}$$

$$\frac{\langle \Pi, \sigma, \theta \rangle \xrightarrow{\pi, \pi'} \text{ for all } \pi \in [\![G]\!], \pi' \in \mathsf{Goal}}{\langle ex(G, G'), \langle \Pi, \sigma, \theta \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma, \theta \rangle, I \rangle}$$

The meta basic action $ex(G, G')$ is a call to the object agent language system to execute a maximal subset of goals from the set $[\![G]\!]$ and recording the result of executing those goals in G' (resulting in a change to the variable valuation I). The variable G is also changed and contains the remaining goals which are not executable in the current object state. The second transition rule specifies the termination condition of the iteration.

The iterative definition of *ex* implicitly defines an executable sequence of goals from $[\![G]\!]$ that is maximal, i.e. which cannot be extended with other executable goals from $[\![G]\!]$. The execution rules for *ex* do not demand that

a sequence from $[\![G]\!]$ is chosen that is the longest sequence relative to all other executable sequences, i.e. it is a local maximum. The computation of a global maximum would require a lookahead facility that checks whether or not a sequence of goals from $[\![G]\!]$ can be extended or not. In case a longest sequence is asked for such a facility is needed to be able to compare different sequences in this respect.

8.3.3 Execution Rules for *apply*

Much along the same lines as for the action *ex* we can define the semantics of the action *apply*. As before, an iterative definition of the semantics of *apply* is given. Informally, the action $apply(R, G, G')$ applies as many rules from $[\![R]\!]$ to as many goals from $[\![G]\!]$ as possible, recording the change to the goal base as a result of this action in G'. We have to distinguish two cases. The first case concerns the application of a reactive rule. In this case, a reactive rule from $[\![R]\!]$ is applied, the rule is removed from $[\![R]\!]$, and the result is stored in the variable G'. The rule has to be removed from $[\![R]\!]$ to guarantee termination of *apply*. The second case concerns the case that a rule modifies a goal from G. In this case an arbitrary goal from G is chosen and an arbitrary rule from R is chosen that is applicable to the chosen goal; the goal is deleted from $[\![G]\!]$, and the result of applying the rule to the goal is stored in the output variable G'. In case π' is the empty goal, by convention, $\{\pi'\}$ denotes the empty set.

Definition 8.11 *(execution rules for apply)*

$$\frac{\langle \Pi, \sigma, \theta\rangle \xrightarrow{\rho,\pi'} \langle \Pi', \sigma, \theta'\rangle, \rho \in [\![R]\!]}{\begin{array}{c}\langle apply(R, G, G'), \langle \Pi, \sigma, \theta\rangle, I\rangle \Longrightarrow \\ \langle apply(R, G, G'), \langle \Pi', \sigma, \theta'\rangle, I\{[\![R]\!] - \{\rho\}/R, [\![G']\!] \cup \{\pi'\}/G'\}\rangle\end{array}}$$

$$\frac{\langle \Pi, \sigma, \theta\rangle \xrightarrow{\pi,\rho,\pi'} \langle \Pi', \sigma, \theta'\rangle, \pi \in [\![G]\!], \rho \in [\![R]\!]}{\begin{array}{c}\langle apply(R, G, G'), \langle \Pi, \sigma, \theta\rangle, I\rangle \Longrightarrow \\ \langle apply(R, G, G'), \langle \Pi', \sigma, \theta'\rangle, I\{[\![G]\!] - \{\pi\}/G, [\![G']\!] \cup \{\pi'\}/G'\}\rangle\end{array}}$$

$$\frac{\langle \Pi, \sigma, \theta\rangle \not\xrightarrow{\pi,\rho,\pi'} \text{ for all } \pi \in [\![G]\!], \rho \in [\![R]\!], \pi' \in \mathsf{Goal}}{\langle apply(R, G, G'), \langle \Pi, \sigma, \theta\rangle, I\rangle \Longrightarrow \langle E, \langle \Pi, \sigma, \theta\rangle, I\rangle}$$

The meta action $apply(R, G, G')$ is a call to the object agent language system to apply as many rules from $[\![R]\!]$ to as many goals from $[\![G]\!]$ as possible. Note, as before, that the recursive definition implicitly defines a sequence of rule-goal pairs. This sequence is maximal in the sense that it cannot be extended with another rule-goal pair from $[\![R]\!] \times [\![G]\!]$ such that the rule is applicable to the goal, i.e. it is a local maximum. As before, however, such a sequence does not necessarily have to be a global maximum of all the sequences of rule-goal pairs from $[\![R]\!] \times [\![G]\!]$, since this would require a lookahead facility.

8.4 Execution Rules for Regular Programming Constructs

The transition rules for the regular programming constructs define the usual semantics for these constructs. For an explanation, we refer back to section 2 (sequential composition, nondeterministic choice) and the literature (cf. [18]).

Definition 8.12 *(assignment for program variables)*

$$\frac{}{\langle G := g, \langle \Pi, \sigma \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma \rangle, I\{[\![g]\!]/G\} \rangle}$$

Definition 8.13 *(assignment for rule variables)*

$$\frac{}{\langle R := r, \langle \Pi, \sigma \rangle, I \rangle \Longrightarrow \langle E, \langle \Pi, \sigma \rangle, I\{[\![r]\!]/R\} \rangle}$$

Definition 8.14 *(test for goal terms)*

$$\frac{[\![g]\!]_\tau = [\![g']\!]_\tau}{\langle g = g', \tau \rangle \Longrightarrow \langle E, \tau \rangle} \qquad \frac{[\![g]\!]_\tau \neq [\![g']\!]_\tau}{\langle g \neq g', \tau \rangle \Longrightarrow \langle E, \tau \rangle}$$

Definition 8.15 *(test for rule terms)*

$$\frac{[\![r]\!]_\tau = [\![r']\!]_\tau}{\langle r = r', \tau \rangle \Longrightarrow \langle E, \tau \rangle} \qquad \frac{[\![r]\!]_\tau \neq [\![r']\!]_\tau}{\langle r \neq r', \tau \rangle \Longrightarrow \langle E, \tau \rangle}$$

Definition 8.16 *(sequential composition)*

$$\frac{\langle \beta_1, \tau \rangle \Longrightarrow \langle \beta_1', \tau' \rangle}{\langle \beta_1;\ \beta_2, \tau \rangle \Longrightarrow \langle \beta_1';\ \beta_2, \tau' \rangle}$$

Definition 8.17 *(nondeterministic left and right choice)*

$$\frac{\langle \beta_1, \tau \rangle \Longrightarrow \langle \beta_1', \tau' \rangle}{\langle \beta_1 + \beta_2, \tau \rangle \Longrightarrow \langle \beta_1', \tau' \rangle} \qquad \frac{\langle \beta_2, \tau \rangle \Longrightarrow \langle \beta_2', \tau' \rangle}{\langle \beta_1 + \beta_2, \tau \rangle \Longrightarrow \langle \beta_2', \tau' \rangle}$$

Definition 8.18 *(nondeterministic repetition)*

$$\frac{}{\langle \beta_1^*, \tau \rangle \Longrightarrow \langle E, \tau \rangle} \qquad \frac{\langle \beta_1, \tau \rangle \Longrightarrow \langle \beta_1', \tau' \rangle}{\langle \beta_1^*, \tau \rangle \Longrightarrow \langle \beta_1'; \ \beta_1^*, \tau' \rangle}$$

9 A Control Structure for 3APL

We are now able to design and compare different control structures by using the meta programming language of the previous section. The control structure that we propose for 3APL is a specialisation of the well-known update-act cycle. The update-act cycle is used in a number of agent languages (cf. [14], [20], [22]). The control structure for 3APL uses the classification and ordering of the rules as a selection mechanism (that is, the ordering on rules in the semantics of the meta language is instantiated with the ordering of rules discussed earlier). The basic outline of this control structure is given by the schema:

Update-Act Cycle

1 Select a rule R to fire

2 Update goal base by firing R

3 Select a goal G

4 Execute (part of) G

5 Goto 1

In step (1) the agent selects a rule which is applied in step (2). Together, steps (1) and (2) constitute the *application phase*. In step (1) a rule is selected by using the classification of rules. In step (2) the goal base of the agent is modified by applying the selected rule. Steps (1) and (2) are also called the planning phase in the literature. However, because 3APL allows for different rules than plan rules, we call this phase the application phase. This is reflected in the proposed control structure for 3APL, where

this application phase is split up into two separate parts and a so-called filter phase is added to deal with failure.

Steps (3) and (4) constitute the *execution phase*. At step (4), we need to decide what part of a goal is to be executed. Several reasons suggest that the selected goal should not be completely executed. First of all, there is no way of deciding whether or not it is possible to execute a goal completely. Secondly, failure rules operate on goals in the goal base; if a complex goal would be executed completely there is no purpose anymore in having failure rules for preventing or dealing with failure while executing the goal. There are several other interesting features of the execution phase which are left for future research. For example, we would like to have a natural ordering which could be used for goal selection. Note that in the proposed control structure at most one rule is fired and at most one goal is (partly) executed *in one cycle*.

A program in the meta language which implements this control structure for 3APL is given next. We will feel free to use constructs like `WHILE`, etc. which may be taken as abbreviations for other, defining expressions of the meta language. We use '_' at argument positions for output variables in the actions *ex* and *apply* if the results returned by these actions are not cared for (i.e. don't have to be recorded in any variable). For example, $ex(G, \_)$ is a call to the object agent system without recording in a variable what the result goals of the execution are.

```
WHILE (Π ≠ ∅) DO
BEGIN
/*/ Planning Phase /*/
selap(R ∪ P ∪ O, Π, R, G);
apply(R, G, _);
/*/ Filter Phase /*/
REPEAT
    selex(Π, G);
    selap(F, G, R, _);
    apply(R, G, _)
UNTIL R = ∅;
/*/ Execution Phase /*/
ex(G, _);
END ;
```

In the program text, the constants $\mathcal{F}, \mathcal{R}, \mathcal{M}, \mathcal{O}$ are used to denote the set of failure, reactive, plan and optimisation rules, respectively, and we just

assume here that some means of indicating to which class a rule belongs is provided for. The application phase in the Update-Act Cycle is refined in the control structure of 3APL, and consists of two separate parts. Steps (1) and (2) of the Update-Act Cycle are quite straightforwardly implemented by the part of the program called *planning phase*. This phase also allows for optimisation and reactive rules to be fired, and thus consists of more than just planning. The second part of the control structure for 3APL called *filter phase* is used to filter out possible failures, and implemented by applying failure rules *until* no rule can be applied anymore to the enabled goal with highest priority. Finally, the execution phase in the control structure again implements quite straightforwardly the same steps in the Update-Act Cycle. The selection of a goal to execute, however, is unnecessary because the goal which is selected in the filter phase can be used.

10 Comparison With Other Languages

In this section, we will discuss two other agent languages, AGENT-0 ([22]) and AgentSpeak(L) ([20]), and comment on ConGolog ([7],[16]). AGENT-0 and AgentSpeak(L) are also rule-based languages. For these languages, we can use the meta programming language to specify the control structures associated with these languages. ConGolog is a logic-based language based on the situation calculus, but is nevertheless closely related to the other procedural languages.

10.1 AGENT-0

An agent in the agent programming language AGENT-0 is an entity consisting of beliefs, commitments, and commitment rules. The agents are also able to communicate with each other. The commitments of AGENT-0 agents correspond to the goals of 3APL agents. AGENT-0 lacks constructs like those of imperative programming to program control flow, but uses explicit representation of time and a global clock to impose some order on the execution of commitments. Commitments are basic actions with a time stamp. The commitment rules correspond to the practical reasoning rules of 3APL. However, commitment rules cannot be used to revise commitments, but can only be used to add new commitments.

In [22] the main features of AGENT-0 are outlined, but the discussion of the language is informal. No formal semantics is given in [22] for AGENT-0. In [12], we specified a formal semantics for AGENT-0 similar to the operational semantics for 3APL. By abstracting from a number of features,

like communication, we can also specify a control structure for AGENT-0 in the meta language.

We first give a short outline of the control structure used in AGENT-0. The strategy for executing commitments and applying rules in AGENT-0, is to execute all executable commitments and apply all applicable rules in every cycle of the control structure. AGENT-0 has a simple mechanism for revision of the agent's commitments statically built into the control structure. Each cycle of the control structure the feasibility of each commitment is checked. If a commitment is considered to be infeasible, then the control structure removes that commitment.

The AGENT-0 control structure can be programmed in the meta language with the actions *ex* and *apply*. A built in feature of these actions is that they try to execute as many goals respectively apply as many rules as possible. By supplying these meta actions with the current set of commitments and all the commitment rules of the agent, these actions update the mental state of the agent according to the strategy used in AGENT-0. Note that in the update-act cycle of AGENT-0 no selection is made, and the agent does not focus on a particular commitment.

```
REPEAT
 /*/ step 1: fire as many rules as possible /*/
G := Π;
apply(Γ, G, _);
/*/ step 2: delete infeasible goals /*/
G := Π;
apply(Δ, G, _);
/*/ step 3: fire as many goals as possible /*/
ex(Π, _)
UNTIL TRUE;
```

In Step (1), the commitments are updated by applying all applicable commitment rules. In Step (2), the removal of infeasible goals is programmed by means of a set Δ of a specific set of practical reasoning rules. The set of rules Δ consists of rules with empty bodies and a guard that specifies the condition under which a goal (the head of the rule) is considered infeasible. Since the bodies of these rules are empty, if one of these rules is applied to a goal that goal is dropped. In Step (3), the actions the agent is committed to are all executed.

One of the main differences between 3APL and AGENT-0 is the range of rules of 3APL and AGENT-0. The rules of 3APL can be used to modify or revise goals, but those of AGENT-0 do not allow such revision but

can only be used to add new commitments. Moreover, 3APL has a clear and formal operational semantics which AGENT-0 lacks. However, we have shown how to specify a semantics for AGENT-0 in [12]. 3APL uses the regular programming constructs for programming control flow while AGENT-0 uses time stamps for this purpose. The control structures of AGENT-0 and 3APL are also on different sides of the extreme. While AGENT-0 each cycle executes all its commitments, 3APL agents focus on one specific goal. Finally, AGENT-0 includes primitives for communication, which we do not comment on here. 3APL has been extended to a multi-agent language with communication primitives (cf. [9]), and a formal semantics for these primitives has been provided.

10.2 AgentSpeak(L)

AgentSpeak(L) ([20]) is a rule-based language which is quite similar to 3APL. An AgentSpeak(L) agent consists of beliefs, intentions, a set of recorded events, and plan rules. In another paper [10], we showed that the notions of event and intention can be reduced to the notion of goal in 3APL. Again, the range of rules in 3APL is much larger than that of AgentSpeak(L) which only provides for plan rules. This means that AgentSpeak(L) lacks the means to modify or revise goals.

In contrast with AGENT-0, AgentSpeak(L) has a formal operational semantics. The formal semantics provides a basis for a detailed comparison. In [10], we show that AgentSpeak(L) can be simulated in 3APL by using a technique called bisimulation. Using these results, we can define a control structure for the regimented version of AgentSpeak(L) where both the notion of events and that of intention are reduced to goals.

The strategy of the control structure of AgentSpeak(L) is similar to that of 3APL. Each cycle of the control structure AgentSpeak(L) applies one applicable plan rule (if possible) and then selects one goal of the agent to execute. The following control structure quite straightforwardly implements each of these steps.

```
WHILE (Π ≠ ∅) DO
BEGIN
/*/ step 1: select an applicable rule /*/
selap(Γ, Π, R, G);
/*/ step 2: fire selected rule /*/
apply(R, G, _);
/*/ step 3: select an executable goal /*/
selex(Π, G);
/*/ step 4: execute selected goal /*/
ex(G, _)
END;
```

The main difference between the control structure of AgentSpeak(L) and 3APL is the filter phase in the control structure of 3APL. This reflects the main difference between the two programming languages, which resides in the types of rules both languages allow for. Because 3APL allows rules for handling failure, it makes sense to use these rules as a filter in an additional phase to prevent as much failure as possible. The lack of such rules in AgentSpeak(L) explains the absence of such a phase.

In many other respects, AgentSpeak(L) and 3APL are quite similar as is formally shown in [10]. AgentSpeak(L) can be viewed as an imperative programming language with belief bases as its domain of computation. Because of the obvious correspondence of a belief base and a database, it is an interesting question what connections there are between agent languages like AgentSpeak(L) and database programming.

10.3 ConGolog

ConGolog ([7],[16]) is a logic-based language based on the situation calculus. An agent in ConGolog is a high-level kind of imperative program. The situations of the situation calculus can be viewed as the belief states of the agent. The high-level program is the analogue of a goal. The semantics of the language is specified in the situation calculus. In [7] a transition system is incorporated into the formalism of the situation calculus. We conjecture that it is possible to prove a simulation result similar to that of AgentSpeak(L) by using the bisimulation technique. This would show there is a closer relation between the different agent languages we have discussed than one might expect at first sight. The main difference between ConGolog and 3APL, however, resides in the execution model. The ConGolog interpreter extracts from a given high-level program a deterministic

sequence of primitive actions by executing a logic program. This provides for a kind of planning as defined in [7]. This focus differs from ours since we are more concerned with the dynamics of the agent's mental life. In our view we provide a more dynamic perspective contrasting the static planning of ConGolog. However, the proposal of a so-called incremental interpreter for ConGolog also provides a more dynamic perspective ([3]).

11 Conclusion

In this paper, we defined the agent programming language 3APL which combines features of logic programming and imperative programming. The language includes all the regular programming constructs from imperative programming and the proof as computation model of logic programming. On top of that, practical reasoning rules are provided which allow for reflective capabilities concerning the goals of an agent. The semantics of 3APL is specified using a transition-style semantics. Transition systems are a well-known formalism to define the operational semantics of a programming language. We discussed the concepts associated with agents and the metaphor of intelligent agents which forms the basis of agent-oriented programming. Some distinguishing features of agents were identified. In particular, we identified the practical reasoning rules of 3APL as a unique feature of intelligent agents. These rules provide for the modification and revision of goals and provide agents with a reflective capability.

In the second part of this paper, we developed a meta programming language for programming control structures for agents. The particular control structure for 3APL is a special case of the update-act cycle. A control structure is a solution to the problem of which goals to select for execution and which rules to select for application. We argued that the selection strategies of agents which are part of a control structure are important for the design of agents. Therefore, we prefer a glass-box approach over a black-box approach which hides these strategies from the programmer. The formal semantics of the meta language is defined by means of a (meta) transition system. Since the semantics of the meta language is specified by means of a second transition system, the total agent system is a two level system. The so-called object level corresponds to the agent language, while the meta level corresponds to the meta language for programming control structures. This allows for a greater flexibility concerning the choice of agent language and type of control structure.

The selection mechanism for rules is based on an intuitive classification of

rules. The classification we proposed is integrated in a natural way into the metaphor of intelligent agents. The classification consists of four distinct classes of rules, each with its own purpose: failure rules for dealing with failure, reactive rules for reactive behaviour, plan rules for planning, and optimisation rules for reducing costs. The classification is used in the control structure for 3APL.

The formal semantics allows for a detailed comparison with some other agent languages. Also, we showed that the meta language is suitable to program control structures and how it can be used as a means of comparing different agent languages. In particular, we discussed the agent languages AgentSpeak(L), AGENT-0, and ConGolog. Although there are many differences, one interesting conclusion is that these languages are more similar than one might at first sight expect.

We list some of the issues that remain for future research. One line of research concerns the extension of the agent language with communication and possibly with other multi-agent features. We have proposed such an extension in [9]. Another important area for future research is the construction of a denotational semantics, and corresponding proof theories, in order to obtain a method for the design of agents. Also, more research is needed to obtain a better understanding of practical reasoning rules and their use. So far, only an intuitive classification and static ordering of rules on which to base priorities has been provided. We would like to extend the ordering to a *dynamic* ordering. Finally, a similar intuitive way of assigning priorities to goals remains for future research.

References

[1] M.E. Bratman. *Intentions, Plans, and Practical Reasoning.* Cambridge: Harvard University Press, 1987.

[2] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[3] Guiseppe de Giacomo and Hector J. Levesque. An incremental interpreter for high-level programs with sensing. Technical report, Department of Computer Science, Univ. of Toronto, 1998.

[4] Barbara Dunin-Keplicz and Jan Treur. Compositional Formal Specification of Multi-Agent Systems. In M.J. Wooldridge and N.R. Jennings, editors, *Intelligent Agents (LNAI 890)*, pages 102–117. Springer-Verlag, 1995.

[5] R.E. Fikes and N.J. Nilson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[6] Michael Fisher. A Survey of Concurrent MetateM - The Language and its Applications. In *Proceedings of First International Conference on Temporal Logic (LNCS 827)*, pages 480–505. Springer-Verlag, 1994.

[7] Guiseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In M. E. Pollack, editor, *Proceedings of the fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226. Morgan Kaufman, 1997.

[8] Koen Hindriks, Frank de Boer, Wiebe van der Hoek, and John-Jules Meyer. Failure, Monitoring and Recovery in the Agent Language 3APL. In Giuseppe De Giacomo, editor, *AAAI 1998 Fall Symposium on Cognitive Robotics*, pages 68–75, 1998.

[9] Koen Hindriks, Wiebe van der Hoek, and John-Jules Meyer. Semantics of Communicating Agents Based on Deduction and Abduction. Technical Report UU-CS-1999-09, Department of Computer Science, University Utrecht, 1999.

[10] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. A Formal Embedding of AgentSpeak(L) in 3APL.

In G. Antoniou and J. Slaney, editors, *Advanced Topics in Artificial Intelligence (LNAI 1502)*, pages 155–166. Springer-Verlag, 1998.

[11] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Formal Semantics for an Abstract Agent Programming Language. In Munindar P. Singh, Anand Rao, and Michael J. Wooldridge, editors, *Intelligent Agents IV (LNAI 1365)*, pages 215–229. Springer-Verlag, 1998.

[12] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Formal Semantics of the Core of AGENT-0. *ECAI'98 Workshop on Practical Reasoning and Rationality*, pages 20–29, 1998.

[13] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Control Structures of Rule-Based Agent Languages. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI 1555)*, pages 381–396. Springer-Verlag, 1999.

[14] Robert Kowalski and Fariba Sadri. Towards a unified agent architecture that combines rationality with reactivity. *Proc. International Workshop on Logic in Databases (LNCS 1154)*, pages 137–149, 1996.

[15] Robert Kowalski, F. Toni, and G. Wetzel. Towards a declarative and efficient glass-box clp language. In N.E. Fuchs and G. Gottlob, editors, *Proc. of the 10th Logic Programming Workshop*. University of Zurich (ifi-Report Nr. 94.10), 1994.

[16] 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 (LNAI 1037)*, pages 331–346. Springer-Verlag, 1996.

[17] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[18] G. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, Computer Science Department, 1981.

[19] A. Poggi. DAISY: An object-oriented system for distributed artificial intelligence. In M.J. Wooldridge and N.R. Jennings, editors, *Intelligent Agents (LNAI 890)*, pages 341–354. Springer-Verlag, 1995.

[20] 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 (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.

[21] Anand S. Rao. Decision procedures for propositional linear-time belief-desire-intention logics. In M.J. Wooldridge, J.P. Müller, and M. Tambe, editors, *Intelligent Agents II*, volume 1037 of *LNAI*, pages 33–48. Springer, 1996.

[22] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[23] Munindar .P. Singh. *Multiagent systems (LNAI 799)*. Springer-Verlag, 1994.

[24] Sarah Rebecca Thomas. *PLACA, An Agent Oriented Programming Language*. PhD thesis, Department of Computer Science, Stanford University, 1993.

[25] Wiebe van der Hoek, Bernd van Linder, and John-Jules 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 (LNCS 813)*, pages 366–378. Springer-Verlag, 1994.

[26] Bernd van Linder, Wiebe van der Hoek, and John-Jules 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 (LNAI 1037)*, pages 17–32. Springer-Verlag, 1996.

[27] 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 (LNAI 1193)*, pages 357–374. Springer, 1997.