A Formal Embedding of AgentSpeak(L) in 3APL

Koen 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
{koenh,frankb,wiebe,jj}@cs.ruu.nl

Abstract

Agent-based computing in Artificial Intelligence has given rise to a number of diverse and competing proposals for agent programming languages. Agents, in the sense we are using it, are complex mental entities consisting of beliefs, goals, and intentions. For several reasons it has been difficult to evaluate and compare the different proposals for agent languages. One of the main reasons, in our opinion, is the lack of a general semantic framework which provides a suitable basis for language comparison. Our aim is to make as much use as possible of formal methods from the area of programming semantics. In this paper, we give a formal embedding of the agent language AgentSpeak(L) in our own agent language 3APL. To this end we define a notion of simulation based on the formal operational semantics of the languages. The main result of the paper is a proof that 3APL can simulate AgentSpeak(L). As a consequence, 3APL has at least the same expressive power as AgentSpeak(L). The comparison yields some new insights into the features of the agent languages. One of the results is that AgentSpeak(L) can be substantially simplified.

1 Introduction

Agent-based computing in Artificial Intelligence has given rise to a number of diverse and competing proposals for agent programming languages. Agents, in the sense we are using it, are complex mental entities comprising the beliefs, goals, and intentions of agents. For several reasons it has been difficult to evaluate and compare the different proposals for agent languages.

One of the reasons for this, we think, is the lack of a general semantic framework which provides a suitable basis for language comparison. Our aim is to make as much use as possible of formal methods from more traditional computing science to deal with these issues. In [4] we introduced the agent programming language 3APL (*triple-a-p-l*) and formally defined its semantics. In that paper, we informally compared our programming language with several other programming languages for agents proposed in the literature. The most important characteristic of our language is that it is rule-based. This is a characteristic which it shares with two other well-known agent languages, AgentSpeak(L) ([8]) and AGENT-0 ([11]).

A more detailed comparison of (rule-based) agent programming languages yields a better understanding of these languages. It yields new insights into the features that are characteristic of agent languages, and into the agent concept itself. In this paper, we give a formal embedding of AgentSpeak(L) in 3APL based on the formal operational semantics of the languages. An embedding of a language in another language is a translation of agents of the source language, in our case AgentSpeak(L), into agents of the target language, 3APL, and a proof that the latter can simulate the former. Since the semantics of AGENT-0 is only given informally in [11], we cannot construct an embedding for AGENT-0 agents. However, the language 3APL, which has a formal semantics, might be used to define the formal semantics of AGENT-0 or an abstraction of AGENT-0.

The structure of this paper is as follows. In section 2 we give an informal overview of 3APL and AgentSpeak(L). In section 3 we will give an outline of the proof that 3APL has at least the same expressive power as AgentSpeak(L), i.e. that 3APL can simulate AgentSpeak(L). We introduce the notion of translation bisimulation which is the formal concept that makes our claims precise. In the next two sections we prove that 3APL can simulate AgentSpeak(L). We proceed in two steps, and first define a translation from AgentSpeak(L) to a language called AgentSpeak(1). In the second step, a translation from AgentSpeak(L) to a subset of 3APL is defined. In section 6 we end with some conclusions.

2 Overview of AgentSpeak(L) and 3APL

The languages AgentSpeak(L) and 3APL have many similarities. The most important of these is that both languages are *rule-based.* In AgentSpeak(L) the rules embody the know-how of the agent, and are used for planning. The informal reading of these rules reads: If the agent wants to achieve goal G and believes that situation S is the case, then plan P might be the right thing to do. The same type of rules for planning are available in 3APL. To avoid confusion, we will call these rules *plan rules* and we will call the body P of such a rule a *plan.* Other types of rules are conceivable, and are also included in 3APL. These second type of rules are used for other purposes than planning what to do, for example, for goal revision (cf. [4, 3]).

An agent of the programming language AgentSpeak(L) consists of *beliefs*, *goals*, *plan rules*, *intentions*, and *events* which make up its mental state. An agent also has an associated set of *actions* for changing its environment. To begin with the latter, *actions* are executed as a result of executing adopted plans. Adopted plans are called *intentions* in AgentSpeak(L), and actions are one of the constituents of intentions.

The *beliefs* represent the situation the agent thinks it is in. In AgentSpeak(L) two types of goals are distinguished: *achievement goals* and *test goals.* A test goal is a check on the set of beliefs of the agent to find out if something is believed to be the case or not. An achievement goal is a state of affairs desired by the agent. The means to achieve that state are provided by *plan rules.* An attempt to accomplish an achievement goal is initiated by a search in the plan library for a plan rule that provides the appropriate means to succeed. If such a plan rule is found, the agent will begin acting upon the plan as specified by the rule.

Plans are hierarchically structured, i.e. a plan may include new subgoals for achieving a higher-level goal. For these subgoals the agent also needs to find suitable plans. The agent keeps track of these plans which were adopted to achieve its (sub)goals by creating a stack of plans which is called an *intention.* Each entry of such a stack is a plan for the first (sub)goal in the plan of the next entry on the stack, except for the last entry.

An agent executes the plans which occur in its intentions. One of the plans is selected for execution, and may specify either that the agent should execute an action or that the agent should accomplish a subgoal. In the case that a an achievement goal is the next thing to accomplish, a so-called *event* is triggered to signal that a suitable plan for the subgoal must be found. Events might also be triggered on other occasions. For example, an event might be triggered to record that a change in the belief base of the agent has occurred. However, the semantics of the latter type of events is not formally defined in AgentSpeak(L).

An agent of the programming language 3APL consists of *beliefs*, *goals*, and *practical reasoning rules* which make up its mental state and has a set of associated *basic actions* which provide for the capabilities of the agent for changing its environment. Basic *actions* are one of the constituents of plans as in AgentSpeak(L), and are formally defined as update operators on the beliefs of the agent. A set of *beliefs* called a belief base is used by the agent to represent the situation the agent thinks he is in.

Goals have more structure than in AgentSpeak(L). Goals in 3APL incorporate achievement goals as well as the adopted plans to achieve goals of the agent. A goal is composed of achievement goals, to achieve a desired state of affairs, basic actions, tests. These constituents of goals are composed by means of sequential, alternative, and parallel composition. Because 3APL goals have more structure than AgentSpeak(L) goals and are the adopted plans of the agent, they are more like the intentions of AgentSpeak(L). In this paper, we formally show that AgentSpeak(L) intentions correspond to goals in 3APL.

Practical reasoning rules serve several purposes. They provide the means to achieve (sub)goals of the agent. This is similar to AgentSpeak(L). However, practical reasoning rules have a more complex structure than the plan rules of AgentSpeak(L). A practical reasoning rule also provides the means for modifying complex goals of the agent, instead of just providing the means for achievement goals. The latter type of rules can be used to express things like: If the agent has adopted a plan P (to achieve a goal G), and believes S is the case, then the agent should (consider) revising plan P and substitute it with a new plan P' or goal G'.

This concludes our short summary of AgentSpeak(L) and 3APL. For the details, the reader is referred to [8] for AgentSpeak(L) and to [4, 3] for 3APL, and the discussion in the rest of this paper. The operational semantics of AgentSpeak(L) as well as that of 3APL formally specify the exact meaning of the informal concepts which were outlined in the previous paragraphs.

3 Comparing the Expressive Power of Programming Languages

The main contribution of this paper consists in a formal proof that the computational behaviour of AgentSpeak(L) agents can be simulated by 3APL agents in a natural way. It follows from this result that 3APL has at least the same expressive power as AgentSpeak(L).

The proof of this claim is based on two concepts: that of a computation, and that of observation. The concept of a computation is defined by the operational semantics of a programming language. The concept of observation is a state-based concept, derived from that of an agent. Agents are the entities of a programming language which are executed, and give rise to computations.

An agent is able to simulate another agent if every legal (finite or infinite) computation of the latter agent is matched by a legal computation of the first agent. The notion of matching is derived from the notion of observation. The concept of simulation is used to compare the expressive power of AgentSpeak(L) relative to 3APL. To compare the expressive power of AgentSpeak(L) and 3APL we thus have to do two things: (i) we have to find a corresponding 3APL agent for each AgentSpeak(L) agent, and (ii) we have to show that the computations of the AgentSpeak(L) agent are simulated by that of the 3APL agent.

The operational semantics of all agent programming languages other than AgentSpeak(L) in this paper are specified by means of Plotkin-style transition systems ([7]). We have tried to stay as close as possible to the original definition of the semantics of AgentSpeak(L) as given in [8]. The semantics of AgentSpeak(L) is defined by a slightly different and somewhat weaker formalism. It is quite easy, however, to transform the semantics of AgentSpeak(L) into a transition system, as we will show. Both semantic formalisms define a transition relation which formalises the computational behaviour of an agent. A transition relation $\longrightarrow$ is a relation on *agents* that defines the legal computation steps of agents.

Definition 3.1 *(computation)*
Let $\longrightarrow$ be a transition relation on agents. $A \longrightarrow A'$ is called a *computation step* (of agent A). A finite or infinite sequence of agents $A_1, A_2, \ldots$ such that $A_i \longrightarrow A_{i+1}$ for all i is called a *computation.*

The comparison of two agents we will use in this paper is based on a comparison of the set of *possible* computation steps of the agents. The basic idea is that each computation step of one of the agents is matched or *simulated* by a computation step of the other agent, and vice versa. The comparison based on this idea is called *(strong) bisimulation* in the literature (cf. [6, 5]). A bisimulation is a binary relation between agents, based on a transition relation which defines the legal computation steps of agents.

To be able to compare computation steps of agents we need to make explicit when computation steps match with each other. In action-based semantics, transitions are labelled by actions, and it is easy to state such a condition: two computation steps match if they have the same action labels (cf. [5]). In case a state-based, unlabelled transition semantics is used, as is the case for the agent programming languages in this paper, the matching needs to be based on a state-based concept. We use the state-based concept of an *observable.* For now, we will assume that a function $\mathcal{O} : \mathcal{A} \to \Omega$, where $\mathcal{A}$ is a set of agents and Ω is the set of observables, has been given and defines the notion of observable. The function $\mathcal{O}$ allows us to *observe changes* in an agent during its execution.

Definition 3.2 *(strong bisimulation)*
Let $\mathcal{A}$ and $\mathcal{B}$ be two sets of agents. A binary relation $R \subseteq \mathcal{A} \times \mathcal{B}$ over agents is a *bisimulation* if $(A, B) \in R$ implies,

(i) Whenever $A \longrightarrow A'$ then, for some B', $B \longrightarrow B'$, and $(A', B') \in R$,

(ii) Whenever $B \longrightarrow B'$ then, for some A', $A \longrightarrow A'$, and $(A', B') \in R$, and

(iii) $\mathcal{O}(A) = \mathcal{O}(B)$.

The notion of strong bisimulation is the basic tool we use for comparing agents. However, we will make two changes to obtain a somewhat more suitable notion for our purposes. First of all, the notion of *strong* bisimulation requires that each computation step of one agent is matched by a computation step of the other agent. However, we might argue that computation steps $A \longrightarrow A'$ such that $\mathcal{O}(A) = \mathcal{O}(A')$ are not observable, and do not have to be simulated. We call such steps *internal* or *stuttering steps.* The notion that we obtain by allowing that internal steps of one agent are matched by zero or more internal steps of the other agent is called *weak bisimulation.*

Definition 3.3 *(derived transition relation $\Rightarrow$)*
Let $\longrightarrow$ be a transition relation on agents from $\mathcal{A}$, and $\longrightarrow^*$ denote the reflexive, transitive closure of $\longrightarrow$.

- the *internal step transition relation* $\xrightarrow{i}$ is obtained by restricting $\longrightarrow$: $A \longrightarrow_i A'$ iff $A \longrightarrow A'$ and $\mathcal{O}(A) = \mathcal{O}(A')$,
- the *derived transition relation* $\Rightarrow$ is defined by: $A \Rightarrow A'$ iff there are agents $X, X' \in \mathcal{A}$ such that $A \longrightarrow_i^* X$, $X \longrightarrow X'$ *or* $X = X'$, and $X' \longrightarrow_i^* A'$.

Note that $\longrightarrow_i^* \subseteq \Rightarrow$. Given that $\Rightarrow$ is the derived transition relation of $\longrightarrow$, we can define weak bisimulation. The definition of strong bisimulation 3.2 and that of weak bisimulation 3.4 are very similar. The difference is that a weak bisimulation allows that an internal step is simulated by zero or more internal steps and allows that a non-internal step is simulated by a number of internal steps and *one* non-internal step. Note that every strong bisimulation is a weak bisimulation.

Definition 3.4 *(weak bisimulation)*
Let $\mathcal{A}$ and $\mathcal{B}$ be two sets of agents. A binary relation $R \subseteq \mathcal{A} \times \mathcal{B}$ over agents is a *weak bisimulation* if $(A, B) \in R$ implies,

(i) Whenever $A \longrightarrow A'$ then, for some B', $B \Rightarrow B'$, and $(A', B') \in R$,

(ii) Whenever $B \longrightarrow B'$ then, for some A', $A \Rightarrow A'$, and $(A', B') \in R$, and

(iii) $\mathcal{O}(A) = \mathcal{O}(B)$.

For our purposes, we make a second change to the definition of bisimulation to obtain a specialised variant of (weak) bisimulation which relates two agents of (possibly) different languages. We assume that a method for mapping an agent from language $\mathcal{L}'$ to an agent from a language $\mathcal{L}$ is given. Such a method is called a *translation function.* If a translation function τ defines a (weak) bisimulation R, i.e. $\tau = R$, we obtain a special case of (weak) bisimulation also called a p-morphism in the literature (cf. [10]). A p-morphism is a bisimulation such that the bisimulation relation is a function. This specialised notion of bisimulation yields a concept suitable to compare the expressive power of agent programming languages. In general, however, the sets of observables of two different languages are different. Therefore, we introduce a mapping called a *decoder.* A decoder maps observables from language $\mathcal{L}'$ back onto observables of $\mathcal{L}$ (we are assuming that $\mathcal{L}'$ simulates $\mathcal{L}$, but not necessarily vice versa).

Definition 3.5 *(translation bisimulation)*
Let $\longrightarrow_{\mathcal{A}}$, $\longrightarrow_{\mathcal{B}}$ be two transition relations defined on the sets of agents $\mathcal{A}$ and $\mathcal{B}$, respectively. Let $\tau : \mathcal{A} \rightarrow \mathcal{B}$ be a (total) mapping from $\mathcal{A}$ to $\mathcal{B}$. Furthermore, $\mathcal{O}_{\mathcal{A}} : \mathcal{A} \rightarrow \Omega_{\mathcal{A}}$ and $\mathcal{O}_{\mathcal{B}} : \mathcal{B} \rightarrow \Omega_{\mathcal{B}}$ are two functions defining the observables of agents from the $\mathcal{A}$ and $\mathcal{B}$.
Then: τ is a *translation bisimulation* if $B = \tau(A)$ implies,

- Whenever $A \longrightarrow_{\mathcal{A}} A'$, then $B \Rightarrow_{\mathcal{B}} B'$, where $B' = \tau(A')$,
- Whenever $B \longrightarrow_{\mathcal{B}} B'$, then for some A', $A \Rightarrow_{\mathcal{A}} A'$ such that $B' = \tau(A')$, and
- There is a *decoder* $\delta : \Omega_{\mathcal{B}} \rightarrow \Omega_{\mathcal{A}}$ such that $\delta(\mathcal{O}_{\mathcal{B}}(B)) = \mathcal{O}_{\mathcal{A}}(A)$.

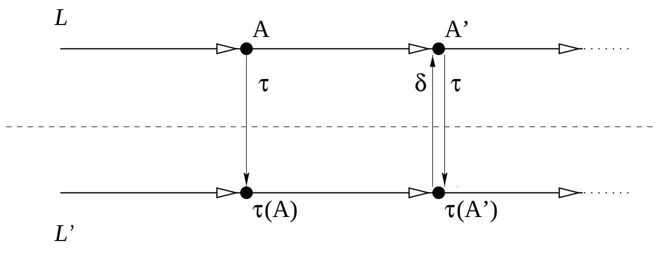

Figure 1: Translation Bisimulation

Figure 1 illustrates the concept of a translation bisimulation as defined in 3.5. Note that, although not depicted in the figure, the simulating agent may execute more than one step on condition that these steps are internal steps. A translation function τ maps an agent $A \in \mathcal{A}$ to an agent $\tau(A) \in \mathcal{B}$. For any agent A and a computation step $A \longrightarrow_{\mathcal{A}} A'$ there must be a corresponding computation $\tau(A) \Rightarrow_{\mathcal{B}} \tau(A')$ which has the same observable effects. This is defined formally in the first and third conditions in the definition of translation bisimulation, and assures that the behaviour produced by an agent from $\mathcal{A}$ can be simulated by the translated agent $\tau(A)$. The third condition requires a decoder for proving that the observables produced by an agent from $\mathcal{A}$ can be retrieved from the observable behaviour of the corresponding agent from $\mathcal{B}$. In the other direction, we have to make sure that *only* the behaviour produced by the agent from $\mathcal{A}$ is generated and no other observable behaviour is produced by the agent $\tau(A)$ from $\mathcal{B}$. This is formally captured by the second condition in definition 3.5. It states that it must be possible to simulate any computation step of the translated agent from $\mathcal{B}$ by the agent from $\mathcal{A}$, with the same observable effects. Again, we need the decoder to map the observables corresponding to the agents from $\mathcal{B}$ back onto observables corresponding to agents from $\mathcal{A}$. Note that in case the decoder δ is the identity function translation bisimulation just is a p-morphism.

The notion of translation bisimulation specifies a number of requirements which must hold for a language to have at least the same expressive power as another language. To compare the expressive power of two programming languages $\mathcal{L}$ and $\mathcal{L}'$, we need to find a translation bisimulation relating *all* agents in the source language $\mathcal{L}$ to *some* suitable set of agents from the target language $\mathcal{L}'$. However, this notion is still not strong enough to *define* expressive power. The reason is that any reasonable programming language is Turing-complete, and for this reason any programming language can simulate another programming language. Therefore, we will impose one more constraint on the translation function τ: A translation function should also preserve the global structure of an agent (cf. [2]). Such a constraint seems both intuitive and reasonable.

In general this constraint can be formalised by the requirement that the translation function τ and the decoder δ are *compositional*: Every operator *op* of the source language is translated into a *context* $C[X_1, \ldots, X_n]$ (assuming that n is the arity of *op*) of the target language such that a program $op(A_1, \ldots, A_n)$ is translated into $C[\tau(A_1), \ldots, \tau(A_n)]$. In order to account for the complex structure of an agent (that is, its various components comprising its beliefs, goals, intentions, etc.) we will assume that an agent is a tuple $A = \langle E_1, \ldots, E_n \rangle$, where each of the E_i is a subset of the expressions of a programming language $\mathcal{L}$, i.e. $E_i \subseteq \mathcal{L}$. Moreover we assume that the expressions in $\mathcal{L}$ are inductively defined, and we require that the translation function τ is defined compositionally as defined above. A mapping from agents

$\langle E_1, \ldots, E_n \rangle$ from $\mathcal{L}$ to agents $\langle F_1, \ldots, F_m \rangle$ is induced by τ by means of a pre-specified selection criterion which determines for each expression $e \in E_i$ a corresponding target F_j such that $\tau(e) \in F_j$. In this manner we allow the simulation of the different components of an agent.

Definition 3.6 *(expressibility)*
Let $\mathcal{O}_\mathcal{A} : \mathcal{A} \to \Omega_\mathcal{A}$ and $\mathcal{O}_\mathcal{B} : \mathcal{B} \to \Omega_\mathcal{B}$ be two functions from the set of *all* agents $\mathcal{A}$ from $\mathcal{L}$ and *some* suitable set of agents $\mathcal{B}$ from $\mathcal{L}'$ to the corresponding sets of observables.
Then we say that $\mathcal{L}'$ *has at least the same expressive power* as $\mathcal{L}$ if there is a mapping $\tau : \mathcal{A} \to \mathcal{B}$ and a mapping $\delta : \Omega_\mathcal{B} \to \Omega_\mathcal{A}$ which satisfy the following conditions:

E1 τ and δ are compositional,

E2 τ is a translation bisimulation.

Let $\mathcal{L} \leq \mathcal{L}'$ be shorthand for $\mathcal{L}'$ has at least the same expressive power as $\mathcal{L}$. Then it follows from definition 3.6 that $\leq$ is transitive. So, if $\mathcal{L}'$ has at least the expressive power of $\mathcal{L}$, and $\mathcal{L}''$ has at least the expressive power of $\mathcal{L}'$, then we also have that $\mathcal{L}''$ has at least the expressive power of $\mathcal{L}$. (The result follows from the fact that the composition of two compositional translation functions again is compositional and the fact that composing two translation bisimulations again yields a translation bisimulation.)

A slightly stronger notion than that of expressive power is the notion of eliminability of a programming operator. A programming operator is eliminable from a given programming language if the operator can be expressed by other operators in the same language. A simple example from imperative programming is the eliminability of the **repeat** ... **until** operator in the presence of the **while** operator.

Definition 3.7 *(eliminability)*
Let $\mathcal{O}_\mathcal{A} : \mathcal{A} \to \Omega_\mathcal{A}$ and $\mathcal{O}_\mathcal{B} : \mathcal{B} \to \Omega_\mathcal{B}$ be two functions from the set of *all* agents $\mathcal{A}$ from $\mathcal{L}$ and *some* set of agents $\mathcal{B}$ from $\mathcal{L}'$ to the corresponding sets of observables.
Then we say that a programming operator F in the language $\mathcal{L}$ is *eliminable* if there is a mapping $\tau : \mathcal{A} \to \mathcal{B}$ and a mapping $\delta : \Omega_\mathcal{B} \to \Omega_\mathcal{A}$ which satisfy the following conditions:

F1 τ and δ are compositional,

F2 τ is a translation bisimulation, and

F3 $\mathcal{L}' \subset \mathcal{L}$, in particular F is not a programming operator of $\mathcal{L}'$.

F1 and **F2** are equivalent to **E1** and **E2** in definition 3.6. As will become clear later on, the definition allows that (minor) modifications are made to the semantics of the subset of the original language to compensate for the lack of the eliminated operator. This will be illustrated in this paper by the elimination of the notion of event from AgentSpeak(L).

3.1 Outline of the Proof

The proof of the claim that 3APL has at least the same expressive power as AgentSpeak(L) consists of two steps. These steps correspond to the two main conceptual differences between AgentSpeak(L) and 3APL. The concepts referred to are, respectively, the concept of *event* and the concept of *intention*, which do not have an obvious counterpart in 3APL. In the first step (section 4), we construct an intermediate language called AgentSpeak(1) which is a subset of AgentSpeak(L) that does not include events. We show that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L). As a consequence, we have that events are eliminable in the formal sense of definition 3.7. This shows that the notion of event is redundant in AgentSpeak(L).

In the second step of the proof (section 5), we show that AgentSpeak(L) intentions can be transformed into 3APL goals. For this purpose, we use a subset of 3APL called AgentSpeak(2). We show that AgentSpeak(2) has at least the same expressive power as AgentSpeak(1). By transitivity, it then follows that 3APL has at least the same expressive power of AgentSpeak(L), since each of the intermediate programming languages defined in the transformation steps have at least the same expressive power as AgentSpeak(L).

4 The Eliminability of Events

In this section we define a compositional translation function τ_1 of AgentSpeak(L) to a language without events, called AgentSpeak(1), and show that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L). First, the syntax and semantics of the two languages AgentSpeak(L) and AgentSpeak(1) are defined. The definition of the syntax and semantics of AgentSpeak(L) is based on the description of AgentSpeak(L) in [8]. The semantic rules have been changed at a number of places, however, to correct for some omissions in [8]. The semantics of AgentSpeak(1) is given by means of a Plotkin-style transition system.

4.1 The Syntax of AgentSpeak(L)

The beliefs of AgentSpeak(L) agents are given by a fragment of first-order logic, namely the set of literals. A signature for this language provides for the vocabulary to construct terms and formulae in the usual way.

Definition 4.1 *(signature for AgentSpeak(L))*
A *signature* for AgentSpeak(L) is a tuple $\langle \mathsf{Pred}, \mathsf{Func}, \mathsf{Cons}, \mathsf{Acts} \rangle$ where

- Pred is a set of *predicate symbols*,
- Func is a set of *function symbols*,
- Cons is a set of *constant symbols*,
- Acts is a set of *action symbols*.

We assume that Pred, Func, Cons, and Acts are disjoint sets.

Definition 4.2 *(terms)*
Let Var be a set of *variables*. The set of *terms* T is inductively defined by:

[Syn-1] $\mathsf{Var} \subseteq \mathsf{T}$,

[Syn-2] $\mathsf{Cons} \subseteq \mathsf{T}$,

[Syn-3] If $f \in \mathsf{Func}$ of arity n and $t_1, \ldots, t_n \in \mathsf{T}$, then $f(t_1, \ldots, t_n) \in \mathsf{T}$.

Definition 4.3 *(belief atoms, literals)*
The set of *atoms* At and *literals* Lit are defined by:

[Syn-4] $\mathsf{At} = \{P(t_1, \ldots, t_n) \mid P \in \mathsf{Pred}$ of arity $n, t_1, \ldots, t_n \in \mathsf{T}\}$,

[Syn-5] $\mathsf{Lit} = \mathsf{At} \cup \neg\mathsf{At}$.

A ground atom is also called a *base belief.* The set of literals are called *belief literals.*

Definition 4.4 *(actions)*
The set of *actions* A is defined by:

[Syn-6] If $a \in \mathsf{Acts}$ of arity n and $t_1, \ldots, t_n \in \mathsf{T}$, then $a(t_1, \ldots, t_n) \in \mathsf{A}$.

The actions of an agent are the basic means of the agent to change its environment. The set of these actions can be viewed as specifying the capabilities of the agent.

Definition 4.5 *(goals)*
The set of AgentSpeak(L) *goals* G is defined by:

[Syn-7] If $\phi \in \mathsf{At}$, then $!\phi \in \mathsf{G}$,

[Syn-8] If $\phi \in \mathsf{At}$, then $?\phi \in \mathsf{G}$,

A goal $!\phi$ is called an *achievement goal.* An achievement goal $!\phi$ expresses that the agent has a goal to achieve a state of affairs where ϕ is the case. A goal $?\phi$ is called a *test goal.* A test goal is a test on the belief base to check if something is or is not believed to be the case.

Definition 4.6 *(triggering events)*
The set of *triggering events* E_t is defined by:

[Syn-9] If $!\phi \in \mathsf{G}$, then $+!\phi \in \mathsf{E}_t$.

In [8], four types of triggering events are defined. Besides the triggering event $+!\phi$ in definition 4.6, three other types of triggering events, $-!\phi, +?\phi$, and $-?\phi$, are defined. Triggering events are triggered when an addition (+) or deletion (-) to the set of goals or beliefs occurs. The formal semantics in [8], however, does not make any reference to the last three triggering events. Therefore, we do not consider the latter type of triggering events in this paper. The triggering event $+!\phi$ is generated in case a plan for an achievement goal $!\phi$ has to be found. The triggering event is posted when the agent tries to execute a plan and encounters a subgoal $!\phi$. It thus signals the need for a plan for $!\phi$.

Definition 4.7 *(plan rules)*
The set of *plan rules* P is defined by:

[Syn-10] If $e \in \mathsf{E}_t$, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n \in (\mathsf{G} \cup \mathsf{A})$, then
$e : b_1 \wedge \ldots \wedge b_n \leftarrow h_1; \ldots; h_n \in \mathsf{P}$.

Definition 4.8 *(head, context, body)*
Let $e : b_1 \wedge \ldots \wedge b_n \leftarrow h_1; \ldots; h_n$ be a plan rule.

- $e : b_1 \wedge \ldots \wedge b_n$ is called the *head* of the plan rule,
- $h_1; \ldots; h_n$ is called the *body* of the plan rule, and
- $b_1 \wedge \ldots \wedge b_n$ is called the *context* of the plan rule.

We will also call the body of a plan rule the *plan* specified by the rule. Plan rules in AgentSpeak(L) specify the means for achieving a goal, and can be viewed as recipes coding the know-how of the agent. The triggering event in the head of the plan rule indicates the achievement goal for which the body of the plan rule specifies the means for accomplishing it. (Recall that a triggering event signals the need to find a plan for the corresponding achievement goal.) The context of the plan rule describes the circumstances which must hold for the plan to be a suitable option. Empty bodies are allowed in plan rules. We use the symbol $\Box$ to denote an empty body. The reason for introducing this feature is that it serves to simplify the proof rules of AgentSpeak(L).

Definition 4.9 *(intentions)*
The set of *intentions* I is defined by:

[Syn-11] If $p_1, \ldots, p_z \in \mathsf{P}$, then $[p_1 \ddagger \ldots \ddagger p_z] \in \mathsf{I}$.

Intentions are stacks of plans which keep track of the adopted plans to achieve goals of the agent. A plan in a stack is supposed to specify the means for a (sub)goal in a plan at the next entry in the stack (if there is such an entry). The stack thus keeps a record of all the plans adopted for all the (sub)goals encountered so far in the plans on the stack. An intention is constructed during execution. In case the agent encounters an achievement goal in a plan in an intention that it is executing, an event is generated. A suitable (instance) of a plan is searched for, and in case such a plan is found, it is added to the stack. Therefore, not all intentions of the form as specified in Syn-11 are legal, since the definition allows plans at an entry in the stack which are unrelated to the next entry in the stack. Such intentions are not allowed. Upon completion of the execution of a plan in an intention the plan rule is removed again. The symbol T is used to denote the intention $[+!\mathsf{true} : \mathsf{true} \leftarrow \Box]$ and is called the *true intention*.

Definition 4.10 *(events)*
The set of *events* E is defined by:

[Syn-12] If $e \in \mathsf{E}_t$ and $i \in \mathsf{I}$, then $\langle e, i \rangle \in \mathsf{E}$.

Events are pairs of triggering events and intentions. The intention part of an event indicates which intention gave rise to the triggering event. An event of the form $\langle e, T \rangle$ is called an *external* event. All other events are called *internal* events. As we will see, external events are never generated by the AgentSpeak(L) agent itself (the true intention T does not give rise to triggering events). This may imply that they only have meaning in a multi-agent setting.

Definition 4.11 *(AgentSpeak(L) agents)*
An *agent* is a tuple $\langle \mathtt{E}, \mathtt{B}, \mathtt{P}, \mathtt{I} \rangle$, where

- $\mathtt{E} \subseteq \mathsf{E}$ is a set of *events*,

- $\mathtt{B} \subseteq \mathsf{At}$ is a set of *base beliefs*,
- $\mathtt{P} \subseteq \mathsf{P}$ is a set of *plans*, and
- $\mathtt{I} \subseteq \mathsf{I}$ is a set of *intentions.*

AgentSpeak(L) agents consist of beliefs, intentions, and plan rules. Agents also keep a record of the events that are generated. However, not all agents allowed by definition 4.11 are legal. Some initialisation conditions from which execution is started need to be imposed. The initialisation conditions are the following: (i) $\mathtt{E} = \emptyset$, and (ii) all $i \in \mathtt{I}$ are of the form $[+!\mathsf{true} : \mathsf{true} \leftarrow !P(\vec{t})]$. The conditions (i) and (ii) do not constrain an agent in any essential way and are natural to impose. Condition (i) expresses that no events have been generated when execution begins. Condition (ii) expresses that an agent may only have adopted a number of simple achievement goals when execution begins.

The definition of AgentSpeak(L) agents we have given differs in some respects from that in [8]. One of the more important differences is that we do not put (basic) actions in a set to keep record of which actions have to be executed. Instead, we formally define the semantics of actions as updates on the belief base. If the agent needs to keep track of actions which are executed or need to be executed, it can store this information in the belief base. Another difference is that we have not included the three selection functions from [8] for selecting intentions and plans. We think this is an aspect which should not be included in the definition of the operational semantics, but is better viewed as a feature of an interpreter implementing the agent language. The selection functions can be looked upon as defining part of the control structure for an interpreter for AgentSpeak(L) (cf. [3]).

4.2 Semantics of AgentSpeak(L)

The operational semantics of AgentSpeak(L) is given by a proof system. The proof system allows the derivation of computation steps of agents. The proof system consists of a set of proof rules which define a derivability relation $\vdash$. Each rule corresponds to a specific type of computation step. The derivability relation is defined on so-called *configurations*, which consist of the dynamically changing parts of the agent during execution, i.e. the set of generated events, the beliefs, and the intentions of the agent.

Definition 4.12 *(BDI configuration)*
A *BDI configuration* is a tuple $\langle \mathtt{E}, \mathtt{B}, \mathtt{I} \rangle$, where

- $\mathtt{E} \subseteq \mathsf{E}$ is a set of events,
- $\mathtt{B} \subseteq \mathsf{At}$ is a base of beliefs, i.e. a set of base beliefs,
- $\mathtt{I} \subseteq \mathsf{I}$ is a set of intentions.

In the remainder of this section, we assume that an agent $\langle \mathtt{E}, \mathtt{B}, \mathtt{P}, \mathtt{I} \rangle$ has been fixed. Furthermore, we assume that a function specifying the update semantics of the basic actions $\mathcal{T} : \mathsf{A} \times \wp(\mathsf{At}) \to \wp(At)$ is given.

The first semantic rule, is a rule for dealing with internal events. An internal event is generated to achieve a goal, denoted by the triggering event part of the event. A plan which specifies the means for achieving the goal is added to the intention which triggered the event, and the event is dropped.

Definition 4.13 *(proof rule* IntendMeans*)*

$$\frac{\langle\{\ldots,\langle +!P(\vec{t}),[p_1\ddagger\ldots\ddagger p_z]\rangle,\ldots\},\mathtt{B},\mathtt{I}\rangle}{\langle\{\ldots\},\mathtt{B},\mathtt{I}\cup\{[p_1\ddagger\ldots\ddagger p_z\ddagger p]\theta\gamma\}\rangle}$$

where

- $p_z = e : \phi \leftarrow !P(\vec{t}); g_2; \ldots; g_l$,
- $p = +!P(\vec{s}) : \psi \leftarrow h_1; \ldots; h_n \in' \mathtt{P}$, such that p has no occurrences of variables which also occur in the event or intention set,
- θ is a most general unifier for $+!P(\vec{t})$ and $+!P(\vec{s})$, and
- $\mathtt{B} \models \forall(\psi\theta\gamma)$, for a substitution γ such that $\gamma(X)$ is not a variable which also occurs in the event or intention set.

The substitution θ in the plan unifies the triggering event $+!P(\vec{s})$ in the head of the plan rule and the triggering event $+!P(\vec{t})$ in the event. Because of the match, plan $h_1; \ldots; h_n$ is a plan for achieving achievement goal $!P(\vec{t})$. The substitution γ retrieves specific parameters related to the current situation from the belief base by a derivation for the context ψ of the plan rule. A new intention is constructed by pushing the plan on the intention part of the event. The composition of the two substitutions $\theta\gamma$ is used to instantiate variables in this new intention (and thus variables in the plan). The rule IntendMeans deals with internal events. We do not give a rule for external events. Such a rule is given in [8], but since an agent will only generate internal events the rule is redundant.

There is one important difference between the rule IntendMeans as given here and the one given in [8] which concerns the renaming of variables. We use $p \in' \mathtt{P}$ to denote that p is a variant of a rule in $\mathtt{P}$, i.e. a plan rule in which variables may have been renamed uniformly. Such a renaming is necessary to avoid interference between variables which occur in the plan rule with variables that occur in the intention. This issue is discussed in more detail in [4]. Furthermore, the values retrieved by substitutions should be applied to the whole intention and not just to a part of it, as is done in [8]. Otherwise, value-passing would be of limited use.

The generation of an internal event is defined in the next rule. It is the only rule in the system that creates events (cf. remark above). An internal event is generated if during the execution of a plan from an intention an achievement goal is encountered. To achieve the goal a plan has to be found, and an event recording the achievement goal and the intention which gave rise to it is generated.

Definition 4.14 *(proof rule* ExecAch*)*

$$\frac{\langle\mathtt{E},\mathtt{B},\{\ldots,j,\ldots\}\rangle}{\langle\mathtt{E}\cup\{\langle +!P(\vec{t}),j\rangle\},\mathtt{B},\{\ldots\}\rangle}$$

where

- $j = [p_1\ddagger\ldots\ddagger p_{z-1}\ddagger(e : \phi \leftarrow !P(\vec{t}); h_2; \ldots; h_n)]$.

The execution of an action in an intention means updating the belief base according to the update semantics of the action, and removing the action from the intention. The belief base is updated according to the semantic function $\mathcal{T}$ which specifies the update semantics of actions.

Definition 4.15 *(proof rule* ExecAct*)*

$$\frac{\langle \mathtt{E}, \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger (e : \phi \leftarrow a(\vec{t}); h_2; \ldots; h_n)], \ldots\}\rangle}{\langle \mathtt{E}, \mathtt{B}', \{\ldots, [p_1 \ddagger \ldots \ddagger (e : \phi \leftarrow h_2; \ldots; h_n))], \ldots\},\rangle}$$

such that

- $\mathcal{T}(a(\vec{t}), \mathtt{B}) = \mathtt{B}'$.

The execution of a test is a check on the belief base. The test is removed from the intention when it has been executed. Such a check may retrieve data from the belief base, which is recorded in a substitution γ. The substitution γ is applied to the remaining part of the intention to instantiate variables with their corresponding parameters.

Definition 4.16 *(proof rule* ExecTest*)*

$$\frac{\langle \mathtt{E}, \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger (e : \phi \leftarrow ?P(\vec{t}); h_2; \ldots; h_n)], \ldots\}\rangle}{\langle \mathtt{E}, \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger (e : \phi \leftarrow h_2; \ldots; h_n)]\gamma, \ldots\}\rangle}$$

where

- γ is a substitution such that for all X in the domain of γ, $\gamma(X)$ is not a variable which also occurs in the event or intention set,
- $\mathtt{B} \models \forall(P(\vec{t})\gamma)$.

The rule CleanStackEntry to be defined below was omitted in [8], as also noted in [1]. The rule implements the notion of an intention being *executed* in definition 16 in [8]. It is used for the removal of a plan that has been completely executed. The entry occupied by this plan is popped from the intention so that execution may continue with the remainder of the intention (which triggered the completed plan). Besides removing the plan, the achievement goal which gave rise to the plan at the next entry in the intention must also be removed (completed plan execution indicates that the goal has been achieved). An empty body $\Box$ in a plan rule in an intention indicates that the plan has been executed completely.

Definition 4.17 *(proof rule* CleanStackEntry*)*

$$\frac{\langle \mathtt{E}, \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger p_z \ddagger (+!P(\vec{t}) : \phi \leftarrow \Box)], \ldots\}\rangle}{\langle \mathtt{E}, \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger p'_z], \ldots\}\rangle}$$

where

- $p_z = e : \psi \leftarrow !P(\vec{t}); h_2; \ldots; h_n$,
- $p'_z = e : \psi \leftarrow h_2; \ldots; h_n$.

In case that a plan has been executed completely but there is no remaining part of the intention consisting of plans still to be executed, the intention itself may be removed from the intention set. The rule $\mathsf{CleanIntSet}$ is used for this purpose. It was not given in [8], as was also noted in [1]. However, from a formal point of view the rule may be considered redundant because it does not change the observable behaviour of an agent.

Definition 4.18 *(clean rule* $\mathsf{CleanIntSet}$*)*

$$\frac{\langle \mathtt{E}, \mathtt{B}, \{\ldots, [+!P(\vec{t}) : \phi \leftarrow \Box], \ldots\}\rangle}{\langle \mathtt{E}, \mathtt{B}, \{\ldots\}\rangle}$$

The set of semantic rules given in this section defines the operational semantics of AgentSpeak(L). Although we have based our description of the proof rules of AgentSpeak(L) on [8], we have made a number of changes to the rules as they are presented in [8]. The rules we give do not modify the language in any essential way ([9]). Apart from minor differences, the more important issue of renaming variables has already been discussed.

4.2.1 Computations and Observables

The proof system for AgentSpeak(L) defines a derivation relation $\vdash$ on BDI configurations. A proof rule allows to derive from (an instantiation) of the configuration C in the premise of that rule the configuration C' in the conclusion of that same rule. This relation is written as $C \vdash C'$. A derivation defines the legal computations of an agent.

Definition 4.19 *(BDI derivation)*
A *BDI derivation* is a finite or infinite sequence of BDI configurations, i.e. $C_0, \ldots, C_n, \ldots$, where each C_{i+1} is derivable from C_i according to a proof rule, i.e. $C_i \vdash C_{i+1}$.

The language is goal- or intention-driven, i.e. by executing its intentions the agent computes results which are stored in the belief base of the agent. Thus, the agent can be viewed upon as computing a series of updates on the beliefs of the agent. This suggests that taking the belief base of the agent as the observables of the system is a good choice.

Definition 4.20 *(observables)*
Let $\mathcal{C}^L$ be the set of all (legal) BDI configurations, and let $\langle \mathtt{E}, \mathtt{B}, \mathtt{I}\rangle \in \mathcal{C}^L$ be such a configuration. The function $\mathcal{O}^L : \mathcal{C}^L \to \wp(\mathsf{At})$ is defined by $\mathcal{O}^L(\langle \mathtt{E}, \mathtt{B}, \mathtt{I}\rangle) = \mathtt{B}$. $\mathcal{O}^L$ yields the *observable* of a given configuration.

For our purpose, this means that if we can show that two agent languages are capable of producing the same sequences of (observable) belief bases and there exists a natural translation of the agents in one of the languages to agents in the other language, the latter has at least the same expressive power as the former.

4.3 The syntax of AgentSpeak(1)

The definition of the language AgentSpeak(1) is given by the set of syntactic rules Syn-1 to Syn-11. I.e., AgentSpeak(1) is a proper subset of AgentSpeak(L) equivalent to AgentSpeak(L) without events. To show that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L) we first have to define a compositional translation function for mapping AgentSpeak(L) agents onto AgentSpeak(1) agents.

A natural candidate for this function is the function that maps all syntactic expressions of AgentSpeak(L) on the same expressions in AgentSpeak(1), except for AgentSpeak(L) events, which are mapped onto AgentSpeak(1) intentions. I.e., we define the translation function to be the identity function, except for events, which need to be mapped on some other notion since events do not occur in AgentSpeak(1).

The choice to map events onto intentions is explained as follows. Events are only used to indicate that a plan to achieve some achievement goal has to be found. The creation of events thus forms an intermediate step in the process of creating a new intention by pushing a suitable plan onto that intention. By incorporating the stack building into the semantic rule which generates the event in AgentSpeak(L) we may skip this intermediate step and we can do without events in AgentSpeak(1).

Definition 4.21 *(translation function τ_1)*
The translation function τ_1 translating AgentSpeak(L) into AgentSpeak(1) is defined as the identity, except for events, for which it is defined by:

- $\tau_1(\langle e, j\rangle) = j$.

It is easy to see that τ_1 is compositional.

Definition 4.22 *(AgentSpeak(1) agent)*
An *AgentSpeak(1) agent* is a tuple $\langle \mathtt{B}, \mathtt{P}, \mathtt{I}\rangle$ where

- $\mathtt{B} \subseteq \mathsf{At}$ is a set of base beliefs,
- $\mathtt{P} \subseteq \mathsf{P}$ is a set of plans, and
- $\mathtt{I} \subseteq \mathsf{I}$ is a set of intentions.

The set of *legal* AgentSpeak(1) agents is a restriction on the set of agents from definition 4.22. The same restriction on the initial intention set of an AgentSpeak(L) agent is imposed on an AgentSpeak(1) agent: all $i \in \mathtt{I}$ should be of the form $[+!\mathsf{true} : \mathsf{true} \leftarrow !P(\vec{t})]$. The condition expresses that an agent may only have adopted a number of simple achievement goals when execution begins. By definition, an AgentSpeak(L) agent $\langle \mathtt{E}, \mathtt{B}, \mathtt{P}, \mathtt{I}\rangle$ is mapped onto an AgentSpeak(1) agent by τ_1 as follows: $\tau_1(\langle \mathtt{E}, \mathtt{B}, \mathtt{P}, \mathtt{I}\rangle) = \langle \mathtt{B}, \mathtt{P}, \mathtt{I} \cup \tau_1(\mathtt{E})\rangle$. τ_1 is lifted to sets of expressions point-wise, i.e. $\tau_1(S) = \{\tau_1(s) \mid s \in S\}$.

4.4 Semantics of AgentSpeak(1)

The proof system used to specify the semantics of AgentSpeak(L) is replaced by a transition system giving the semantics for AgentSpeak(1). Transition systems are a means to define the operational semantics of programming languages ([7]). Formally, a transition system is a deductive system which allows to *derive* the transitions of an agent. A transition system consists of a set of *transition rules* which specify the meaning of the programming constructs in a language. Transition rules transform *configurations*. In AgentSpeak(1) a configuration is the same as a mental state, i.e. a pair $\langle \mathtt{B}, \mathtt{I}\rangle$ of beliefs and intentions.

Definition 4.23 *(AgentSpeak(1) configuration)*
An *AgentSpeak(1) configuration* is a tuple $\langle \mathtt{B}, \mathtt{I} \rangle$, where

- $\mathtt{B} \subseteq \mathsf{At}$ is belief base,
- $\mathtt{I} \subseteq \mathsf{I}$ is a set of intentions.

By definition, an AgentSpeak(L) configuration is mapped onto an AgentSpeak(1) configuration by τ_1 as follows: $\tau_1(\langle \mathtt{E}, \mathtt{B}, \mathtt{I} \rangle) = \langle \mathtt{B}, \mathtt{I} \cup \tau_1(\mathtt{E}) \rangle$.

The first transition rule, for plan application, corresponds to the proof rule IntendMeans of AgentSpeak(L). The rule, however, does circumvent the notion of event, and directly pushes a plan for achieving the achievement goal in the intention on the intention. Thus, the AgentSpeak(1) rule integrates the rules IntendMeans and ExecAch into one rule.

Definition 4.24 *(plan application rule)*
Let θ be a most general unifier for $!P(\vec{t})$ and $!P(\vec{s})$, and γ a substitution.

$$\frac{+!P(\vec{s}) : \phi \leftarrow h_1; \ldots; h_n \in' \mathtt{P} \text{ and } \mathtt{B} \models \forall(\phi\theta\gamma)}{\langle \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger p_z], \ldots\} \rangle \longrightarrow_1 \langle \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger p_z \ddagger p]\theta\gamma, \ldots\} \rangle}$$

where

- $p_z = e : \phi \leftarrow !P(\vec{t}); g_2; \ldots; g_l$,
- $p = +!P(\vec{s}) : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n$ has no occurrences of variables which also occur in the intention set, and
- $\gamma(X)$ is not a variable also occurring in the intention set.

The other four rules of AgentSpeak(1) are the transition rule variants of the rules ExecAct, ExecTest, CleanStackEntry, and CleanIntSet respectively. In these rules, the set of events is dropped from the configurations.

Definition 4.25 *(execution rule for actions)*

$$\frac{\mathcal{T}(a(\vec{t}), \mathtt{B}) = \mathtt{B}'}{\langle \mathtt{B}, \{..., [p_1 \ddagger ... \ddagger (e : \phi \leftarrow a(\vec{t}); h_2; ...; h_n)], ...\} \rangle \longrightarrow_1 \langle \mathtt{B}', \{..., [p_1 \ddagger ... \ddagger (e : \phi \leftarrow h_2; ...; h_n))], ...\} \rangle}$$

Definition 4.26 *(execution rule for first-order tests)*
Let γ be a substitution such that $\gamma(X)$ is not a variable which also occurs in the intention set.

$$\frac{\mathtt{B} \models \forall(P(\vec{t})\gamma)}{\langle \mathtt{B}, \{..., [p_1 \ddagger ... \ddagger (e : \phi \leftarrow ?P(\vec{t}); h_2; ...; h_n)], ...\} \rangle \longrightarrow_1 \langle \mathtt{B}, \{..., [p_1 \ddagger ... \ddagger (e : \phi \leftarrow h_2; ...; h_n)]\gamma, ...\} \rangle}$$

Definition 4.27 *(clean stack entry rule)*

$$\frac{}{\langle \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger p_z \ddagger (+!P(\vec{t}) : \phi \leftarrow \Box)], \ldots\} \rangle \longrightarrow_1 \langle \mathtt{B}, \{\ldots, [p_1 \ddagger \ldots \ddagger p_z'], \ldots\} \rangle}$$

where

- $p_z = e : \psi \leftarrow !P(\vec{t}); h_2; \ldots; h_n$,
- $p'_z = e : \psi \leftarrow h_2; \ldots; h_n$.

Definition 4.28 *(clean rule for intention set)*

$$\frac{}{\langle \mathtt{B}, \{\ldots, [+!P(\vec{t}) : \phi \leftarrow \Box], \ldots\}\rangle \longrightarrow_1 \langle \mathtt{B}, \{\ldots\}\rangle}$$

4.4.1 Computations and Observables

The transition system for AgentSpeak(1) defines a transition relation $\longrightarrow_1$ on configurations. A transition rule allows the derivation of (an instantiation) of such a transition in case the conditions in the premise of the rule are satisfied.

The transition relation $\longrightarrow_1$ is the counterpart of AgentSpeak(1) for the derivation relation $\vdash$ of AgentSpeak(L). The choice of observables for AgentSpeak(1) is the same as that for AgentSpeak(L), the belief base of an AgentSpeak(1) configuration. Therefore, we can use the identity function as decoder δ to map observables from AgentSpeak(1) to AgentSpeak(L). As a consequence, δ is compositional.

Definition 4.29 *(observables)*
Let $\mathcal{C}^1$ be the set of all (legal) AgentSpeak(1) configurations. The function $\mathcal{O}^1 : \mathcal{C}^1 \to \wp(\mathsf{At})$ is defined by $\mathcal{O}^1(\langle \mathtt{B}, \mathtt{I}\rangle) = \mathtt{B}$ for all $\langle \mathtt{B}, \mathtt{I}\rangle \in \mathcal{C}^1$. $\mathcal{O}^1$ yields the *observable* of an AgentSpeak(1) configuration.

4.5 AgentSpeak(1) simulates AgentSpeak(L)

To show that τ_1 is a translation bisimulation and that AgentSpeak(1) simulates AgentSpeak(L), we have to show that every computation step of an AgentSpeak(L) agent can be simulated by the translated AgentSpeak(1) agent, and vice versa. Because all but one rule for AgentSpeak(1) are just notational variants of the rules for AgentSpeak(L) this is not too difficult. Therefore, it suffices to show that the computation steps defined by the rules IntendMeans and ExecAch can be simulated in AgentSpeak(1), and that the plan application rule of AgentSpeak(L) can be simulated by AgentSpeak(L).

Theorem 4.30 The proof rule ExecAch of AgentSpeak(L) can be simulated by AgentSpeak(1).

Proof: By inspection of the proof rule ExecAch and the definition of τ_1, it is easy to see that the AgentSpeak(L) configuration in the premise of the rule and the configuration of the conclusion of the rule are mapped onto the same AgentSpeak(1) configuration. The rule ExecAch thus defines an internal step, and the computation step corresponding to this rule is simulated by performing no AgentSpeak(1) step at all. ■

Theorem 4.31 The proof rule IntendMeans of AgentSpeak(L) can be simulated by the plan application rule for AgentSpeak(1).

Proof: Let $A = \langle \mathtt{E} \cup \{\langle +!P(\vec{t}), [p_1 \ddagger \ldots \ddagger p_z]\rangle\}, \mathtt{B}, \mathtt{I}\rangle$ and $A' = \langle \mathtt{E}, \mathtt{B}, \mathtt{I} \cup \{[p_1 \ddagger \ldots \ddagger p_z \ddagger p]\theta\gamma\}\rangle$ such that $A \vdash A'$ by means of the rule IntendMeans. Translation function τ_1 maps A onto $B = \tau(A) = \langle \mathtt{B}, \mathtt{I} \cup \tau_1(E)\{[p_1 \ddagger \ldots \ddagger p_z]\}\rangle$. By applying the rule for plan application of AgentSpeak(1) we get: $B' = \langle \mathtt{B}, \mathtt{I} \cup \tau_1(E)\{[p_1 \ddagger \ldots \ddagger p_z \ddagger p]\theta\gamma\}\rangle$. It is easy to see that $\tau_1(A') = B'$. ($\tau_1(i\theta\gamma) = \tau_1(i)\theta\gamma$ for an intention i.) Note that the plan application rule is applicable since proof rule IntendMeans is applicable. ■

Note that although rule IntendMeans defines an internal step, it can not be simulated by not performing any step at all. The reason for this is that $\tau(C) \neq \tau(C')$ if $C \vdash C'$ by the rule IntendMeans.

Theorem 4.32 The plan application rule of AgentSpeak(1) can be simulated by AgentSpeak(L).

Proof: To simulate the plan application rule, we need the proof rule IntendMeans as well as the proof rule ExecAch. Note that since both steps are internal steps, it is allowed to use both rules to prove weak bisimulation. Suppose A is an AgentSpeak(L) configuration, and $B \longrightarrow_1 B'$ by means of the plan application rule, where $B = \tau(A)$ and B' is an AgentSpeak(1) configuration. The configurations must be of the form: $B = \tau(A) = \langle \mathtt{B}, \mathtt{I} \cup \{[p_1 \ddagger \ldots \ddagger p_z]\}\rangle$, and $B' = \langle \mathtt{B}, \mathtt{I} \cup \{[p_1 \ddagger \ldots \ddagger p_z \ddagger p]\theta\gamma\}\rangle$. By inspection of the translation function, and by rule ExecAch we get a configuration $A \vdash A'$ of the form $A' = \langle\{\mathtt{E} \cup \{\langle +!P(\vec{t}), [p_1 \ddagger \ldots \ddagger p_z]\rangle\}, \mathtt{B}, \mathtt{I}\rangle$. And, finally, by rule IntendMeans we get $A'' = \langle \mathtt{E}, \mathtt{B}, \mathtt{I} \cup \{[p_1 \ddagger \ldots \ddagger p_z \ddagger p]\theta\gamma\}\rangle$. It is easy to see that $\tau(A'') = B'$. ■

We summarise the results of this section. AgentSpeak(1) syntactically is a proper subset of AgentSpeak(L). This corresponds to the condition **F3** in the definition of eliminability 3.7. The translation function τ_1 and the decoder δ both are compositional, which corresponds to condition **F1** in definition 3.7. And, finally, by the proofs given above τ_1 is a translation bisimulation. This corresponds to condition **F2** in definition 3.7. Taken together, this concludes the proof that events are eliminable from AgentSpeak(L). As a consequence, we also have that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L).

In the other direction, although we did not give a formal proof for it, we claim that AgentSpeak(L) also has at least the same expressive power as AgentSpeak(1). The proof is easily derived from the proofs in this section. The translation function needed for the bisimulation is the identity function. As a consequence, AgentSpeak(L) and AgentSpeak(1) have the same expressive power.

5 The Transformation of Intentions to Goals

In this section we define a translation of AgentSpeak(1) to a language called AgentSpeak(2). In AgentSpeak(2) intentions have been replaced by complex goals of 3APL. We define a compositional translation function τ_2. Apart from the replacement of intentions by complex goals, there have been made a number of minor other syntactic changes to the language AgentSpeak(1). AgentSpeak(2) is a proper subset of 3APL.

5.1 Syntax of AgentSpeak(2)

The main difference between the syntax of AgentSpeak(1) and AgentSpeak(2) is that the latter does not have intentions. Also, a number of other changes in the syntax of AgentSpeak(1) goals and plan rules have been made to compensate for minor differences between the syntax of AgentSpeak(L) and 3APL. The symbol ! marking that a goal is an achievement goal in $!\phi$ is simply dropped. $?\phi$ is written as $\phi?$. And finally, a plan $+!P(\vec{t}) : \phi \leftarrow h_1; \ldots; h_n$ is written as $P(\vec{t}) \leftarrow \phi \mid h_1; \ldots; h_n$. Because of the change in the syntax of plan rules, we no longer have any need for triggering events. Therefore, triggering events are dropped from the language.

Definition 5.1 *(syntax)*
The syntax of AgentSpeak(2) is given by the syntactic rules Syn-1 to Syn-6, and the three rules Syn-7a,Syn-8a, and Syn-10a which replace the rules Syn-7 for achievement goals, Syn-8 for test goals, and Syn-10 for plan rules of AgentSpeak(L). One other rule is added, namely Syn-9a, which defines complex goals, i.e. sequential compositions of simple goals.

[Syn-7a] If $\phi \in \mathsf{At}$, then $\phi \in \mathsf{G}$,

[Syn-8a] If $\phi \in \mathsf{At}$, then $\phi? \in \mathsf{G}$,

[Syn-9a] If $h_1, \ldots, h_n \in (\mathsf{G} \cup \mathsf{A})$, then $h_1; \ldots; h_n \in \mathsf{G}$,

[Syn-10a] If $\phi \in \mathsf{At}$, $b_1, \ldots, b_n$ are belief literals, and $h_1, \ldots, h_n \in (\mathsf{G} \cup \mathsf{A})$, then $\phi \leftarrow b_1 \wedge \ldots \wedge b_n \mid h_1; \ldots; h_n \in \mathsf{P}$.

Because of the syntactic changes made to AgentSpeak(1) the translation function τ_2 is more complicated than τ_1. τ_2 maps intentions to the complex goals as defined by rule Syn-9a. To do this, the stack of plans (intention) needs to be unravelled into a sequence of actions, tests, and achievement goals. For this purpose, we define an auxiliary function τ_2', and use a function *body* to obtain the body of a plan rule. τ_2' removes the achievement goals at the head of the plans in an intention, since these goals are implemented by plans at the next lower entry in the stack except for the first entry (legal intentions have this feature). The remainder of the plans is transformed into a sequential goal. Recall that a plan rule p in an intention $[\ldots \ddagger p]$ is the one executed first (cf. semantic rules for AgentSpeak(1)). Therefore, the plan specified by p is added to the front of the complex goal which is constructed by τ_2'.

Definition 5.2 *(translation function τ_2)*
The translation function τ_2 translating AgentSpeak(1) into AgentSpeak(2) is defined as the identity, except for the following cases:

- $\tau_2(!\phi) = \phi$,
- $\tau_2(?\phi) = \phi?$,
- $\tau_2(+!P(\vec{t}) : \phi \leftarrow h_1; \ldots; h_n) = P(\vec{t}) \leftarrow \phi \mid h_1; \ldots; h_n$,
- $\tau_2'([p_1 \ddagger \ldots \ddagger (+!P(\vec{t}) : \phi \leftarrow h_1; h_2; \ldots; h_n)]) = h_2; \ldots; h_n; \tau_2'([p_1 \ddagger \ldots \ddagger p_{z-1}])$,
- $\tau_2'([]) = \square$,
- $\tau_2([p_1 \ddagger \ldots \ddagger p_z]) = \mathit{body}(\tau_2(p_z)); \tau_2'([p_1 \ddagger \ldots \ddagger p_{z-1}])$.

As before, it is easy to see that τ_2 is compositional, and therefore satisfies condition **E1** of definition 3.6. Also, note that the composition $\tau_1 \circ \tau_2$ translates AgentSpeak(L) expressions to AgentSpeak(2) expressions.

AgentSpeak(2) agents consist of a belief base, a goal base, and a plan base. The goal base of an AgentSpeak(2) agent replaces the intention set of an AgentSpeak(1) agent.

Definition 5.3 *(AgentSpeak(2) agent)*
An *AgentSpeak(2) agent* is a tuple $\langle \mathtt{B}, \mathtt{P}, \mathtt{G} \rangle$ where

- $\mathtt{B} \subseteq \mathsf{At}$ is a set of base beliefs,
- $\mathtt{P} \subseteq \mathsf{P}$ is a set of plans, and
- $\mathtt{G} \subseteq \mathsf{G}$ is a set of goals.

By definition, an AgentSpeak(1) agent $\langle \mathtt{B}, \mathtt{P}, \mathtt{G} \rangle$ is mapped onto an AgentSpeak(2) agent by τ_2 as follows: $\tau_2(\langle \mathtt{B}, \mathtt{P}, \mathtt{I} \rangle) = \langle \mathtt{B}, \tau_2(\mathtt{P}), \tau_2(\mathtt{I}) \rangle$. τ_2 is lifted point-wise to sets.

5.2 Semantics of AgentSpeak(2)

Since intentions have been dropped from the language, the transition system for AgentSpeak(1) has to be modified such that the transition rules apply to goals instead of intentions. The configurations transformed by AgentSpeak(2) agents are now pairs $\langle \mathtt{B}, \mathtt{G} \rangle$ of belief and goal bases.

Definition 5.4 *(configuration)*
An AgentSpeak(2) *configuration* is a pair $\langle \mathtt{B}, \mathtt{G} \rangle$, where $\mathtt{B}$ is a belief base, and $\mathtt{G}$ is a set of AgentSpeak(2) goals, either simple or complex.

By definition, an AgentSpeak(1) configuration is mapped onto an AgentSpeak(2) configuration by τ_2 as follows: $\tau_2(\langle \mathtt{B}, \mathtt{I} \rangle) = \langle \mathtt{B}, \tau_2(\mathtt{I}) \rangle$.

The main difference between the transition rules of AgentSpeak(1) and those of AgentSpeak(2) is that the transition rules of AgentSpeak(2) exploit the recursive capabilities of transition systems. That is, the transition rules of AgentSpeak(2) are defined on the syntactic structure of agents and goals. The rules decompose a complex goal to its elementary parts, and the semantics of a complex goal is derived in this way from the rules for the elementary parts. For example, the rule for sequential composition decomposes a sequential goal, and transformation of the sequential goal is derived from the transformation of the first part of the sequential goal.

The rule for goal bases selects a goal which is executable, or a goal which can be used in the plan application rule. This is the only rule for defining the transition relation $\longrightarrow_2$ on configurations $\langle \mathtt{B}, \mathtt{G} \rangle$. The transition relation $\longrightarrow_2$ is derived from the more basic relation $\longrightarrow$ which operates on a pair $\langle \mathtt{B}, \pi \rangle$ of belief base and goal.

Definition 5.5 *(goal base)*

$$\frac{\langle \mathtt{B}, \pi \rangle_V \longrightarrow_\theta \langle \mathtt{B}', \pi' \rangle}{\langle \mathtt{B}, \{\ldots, \pi, \ldots\} \rangle \longrightarrow_2 \langle \mathtt{B}', \{\ldots, \pi', \ldots\} \rangle}$$

where V is the set of variables in $\{\ldots, \pi, \ldots\}$.

In case the (complex) goal that is selected turns out to be(gin with) an achievement goal, a plan must be found which will achieve the goal. The transition rule for plan application of AgentSpeak(2) thus corresponds to the rule IntendMeans of AgentSpeak(L).

Definition 5.6 *(rule application)*
Let θ be a most general unifier for $P(\vec{t})$ and $P(\vec{s})$, and γ a substitution.

$$\frac{P(\vec{s}) \leftarrow \phi \mid h_1; \ldots; h_n \in' \mathtt{P} \text{ and } \mathtt{B} \models \forall(\phi\theta\gamma)}{\langle \mathtt{B}, P(\vec{t})\rangle_V \longrightarrow_{\theta\gamma} \langle \mathtt{B}, (h_1; \ldots; h_n)\theta\gamma\rangle}$$

where

- $P(\vec{s}) : b_1 \wedge \ldots \wedge b_m \leftarrow h_1; \ldots; h_n$ has no occurrences of variables which are also in V (the set of variables occurring in the goal base), and
- $\gamma(X) \notin V$ for all X in the domain of γ.

The execution of a basic action consists in an update on the belief base. $\emptyset$ denotes the empty substitution and the symbol E denotes termination of a goal.

Definition 5.7 *(execution rule for actions)*

$$\frac{\mathcal{T}(a(\vec{t}), \mathtt{B}) = \mathtt{B}'}{\langle \mathtt{B}, a(\vec{t})\rangle_V \longrightarrow_{\emptyset} \langle \mathtt{B}', E\rangle}$$

A test is a check on the belief base.

Definition 5.8 *(execution rule for first-order tests)*
Let γ be a substitution such that $\gamma(X) \notin V$ for all X in the domain of γ.

$$\frac{\mathtt{B} \models \forall(P(\vec{t})\gamma)}{\langle \mathtt{B}, P(\vec{t})?\rangle_V \longrightarrow_{\gamma} \langle \mathtt{B}, E\rangle}$$

The rule for sequential composition first decomposes a complex goal into its simpler constituents, executes the first part of the goal, and composes the remaining part sequentially with the second half of the goal. Substitutions created during execution are also applied to the second half of the goal.

Definition 5.9 *(execution rule for sequential composition)*
Let θ be a substitution.

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

5.3 Computations and Observables

An AgentSpeak(2) computation is a finite or infinite sequence of AgentSpeak(2) configurations where each consecutive pair is related by the transition relation $\longrightarrow_2$. As for AgentSpeak(1), we define the notion of observables for AgentSpeak(2) to be the belief base and the decoder δ to be the identity function.

Definition 5.10 *(observables)*
Let $\mathcal{C}^2$ be the set of all AgentSpeak(2) configurations, and let $\langle \mathtt{B}, \mathtt{G}\rangle \in \mathcal{C}^2$ denote such a configuration. The function $\mathcal{O}^2 : \mathcal{C}^2 \to \wp(\mathsf{At})$ is defined by $\mathcal{O}^2(\langle \mathtt{B}, \mathtt{G}\rangle) = \mathtt{B}$. $\mathcal{O}^2$ yields the *observable* of a given AgentSpeak(2) configuration.

5.4 AgentSpeak(2) simulates AgentSpeak(1)

To show that τ_2 is a translation bisimulation and that AgentSpeak(2) simulates AgentSpeak(1), we have to show that every computation step of an AgentSpeak(1) agent can be simulated by the corresponding translated AgentSpeak(2) agent, and vice versa. Since the transition rules of AgentSpeak(1) define the legal computation steps, it suffices to show that every transition rule of AgentSpeak(1) can be simulated by the derived transition relation $\Rightarrow_2$ of AgentSpeak(2), and vice versa. We show this for the most complex case, namely the case that a plan rule is applied in the computation step.

Also note that the rules for cleaning the intention stacks CleanStackEntry and CleanIntSet do not have to be simulated (the rules are simulated by performing no step at all) since the translation function τ_2 maps the configuration of the premise of these rules onto the same AgentSpeak(2) configuration as the conclusion of the rules.

Theorem 5.11 The plan application rule for AgentSpeak(1) can be simulated by the plan application rule for AgentSpeak(2).

Proof: We use $\mathtt{P}_1$ to denote the set of plans of the AgentSpeak(1) agent, and $\mathtt{P}_2$ to denote the set of plans of the AgentSpeak(2) agent.

- ($\Rightarrow$:) Suppose that $A \longrightarrow_1 A'$ by the plan application rule of AgentSpeak(1). In that case, $A = \langle \mathtt{B}, \mathtt{I} \cup \{[p_1 \ddagger \ldots \ddagger p_z]\}\rangle$ and $A' = \langle \mathtt{B}, \mathtt{I} \cup \{[p_1 \ddagger \ldots \ddagger p_z \ddagger p]\theta\gamma\}\rangle$, where $p_z = e : \phi \leftarrow\, !P(\vec{t}); g_2; \ldots; g_l$, and $p = +!P(\vec{s}) : \psi \leftarrow h_1; \ldots; h_n \in' \mathtt{P}_1$. The achievement goal $!P(\vec{t})$ in plan p_z is translated by τ_2 to $P(\vec{t})$, and the plan p to $P(\vec{s}) \leftarrow \psi \mid h_1; \ldots; h_n \in' \mathtt{P}_2$. By an application of the plan application rule for AgentSpeak(2) we get:

  $$\langle \mathtt{B}, P(\vec{t})\rangle_V \longrightarrow_{\theta\gamma} \langle \mathtt{B}, (h_1; \ldots; h_n)\theta\gamma\rangle$$

  Using the definition of τ_2, τ_2' and this transition as a premise for the rule for sequential composition, we then get:

  $$\langle \mathtt{B}, P(\vec{t}); \tau_2'([p_1 \ddagger \ldots \ddagger p_z])\rangle \longrightarrow_{\theta\gamma} \langle \mathtt{B}, \tau_2([p_1 \ddagger \ldots \ddagger p_z \ddagger p])\theta\gamma\rangle$$

  And finally, by an application of the transition rule for goal bases, we have:
  $\tau_2(A) \longrightarrow_2 \tau_2(A')$.

- ($\Leftarrow$:) Suppose that $\tau(A) \longrightarrow_2 B'$ by using the rule for plan application of AgentSpeak(2). This means that goal $P(\vec{t}); \pi \in \mathtt{G}$ for some π (possibly empty). By inspection of the translation function τ_2, this goal must have originated from an intention in the configuration A. But in that case, we can apply the plan application rule of AgentSpeak(1) to transform the intention and add the plan used in the AgentSpeak(2) computation step to the stack of plans, yielding a configuration A'. The translation function τ_2 maps this configuration to B', $\tau(A') = B'$.

■

We summarise the results of this section. Both the translation function τ_2 and the decoder δ are compositional. This corresponds to condition **E1** in the definition of relative expressive power 3.6. Theorem 5.11 proves that τ_2 is a translation bisimulation, which corresponds to condition **E2** in definition 3.6. Taken together, this concludes the proof that AgentSpeak(2) has at least the same expressive power as AgentSpeak(1). Since AgentSpeak(2) is a proper subset of 3APL (cf. [4]), this also shows by transitivity of the expressiveness relation that 3APL has at least the same expressive power as AgentSpeak(2). The main difference between AgentSpeak(2) and 3APL resides in the set of (plan) rules, since 3APL allows rules with more general heads. These rules make a more general type of goal revision possible (cf. [4, 3]).

6 Conclusion

The conclusion which can be drawn from the results concerning the expressive power is that every agent which can be programmed in AgentSpeak(L) can also be programmed in 3APL. On the other hand, a number of features of 3APL which are discussed in [4] were not needed to simulate AgentSpeak(L). These features include more (imperative) programming constructs like parallel composition and non-deterministic choice, and a more general goal revision mechanism. Although the lack of regular programming constructs may not be considered as too great a difference, the lack of a more general revision mechanism is of more interest. Since AgentSpeak(L) lacks this more general revision mechanism in AgentSpeak(L), we believe that it is impossible to simulate, in the sense outlined in this paper, this mechanism in AgentSpeak(L). Therefore, we conjecture that 3APL has strictly more expressive power than AgentSpeak(L).

Another conclusion which can be drawn from the expressiveness results is that the notions of events and intentions can be identified. This conclusion is based on the fact that events and intentions are simulated by goals (they are mapped onto goals by the composed translation function $\tau_1 \circ \tau_2$). Similarly, one may conclude that there is no need to maintain a complete stack of plans as is done in AgentSpeak(L). The same reason applies: No such thing is needed in the simulation of AgentSpeak(L) agents by 3APL agents. The bookkeeping for which events and intentions are used, therefore, only complicates the proof system. Since there is no loss of expressiveness, stacks and (triggering) events might be viewed as one possible implementation of the agent language, but should not be incorporated into the semantics of the agent language.

There might be, however, one intuitive use of intentions which is not incorporated in AgentSpeak(L). One could argue that it is of use to an agent that it keeps track of the goals it is pursuing and the plans it is trying to use to achieve these goals as is done in an intention. For example, if a plan fails to achieve a goal, this plan could be dropped, the old goal could be retrieved (from the next entry in the intention structure) and a new plan could be tried. However, the complexity both from a theoretical and practical perspective of this kind of 'backtracking' should not be underestimated. First of all, new types of rules should have to be introduced which could change intention structures in this way. These rules would make the semantics considerably more complex. Furthermore, it is not (yet) clear when and how to use this kind of 'backtracking'. Research into integrating these kinds of possibilities into a formal semantics is still to be done, as far as we know.

Besides the operational semantics for AgentSpeak(L), in [8] also an algorithm for an interpreter for AgentSpeak(L), or a *control structure* as we would like to call it, is defined. This control structure specifies to some extent in which order the proof rules should be used to execute AgentSpeak(L) agents. For example, in every cycle of the interpreter first an event is processed and then an intention is processed (achieve goal, execute action or test). We have looked in more detail at specifying the semantic structure imposed on AgentSpeak(L), and also for 3APL, by an interpreter in [3], using the results obtained in the present paper that events and intentions can be transformed into goals.

References

[1] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A Formal Specification of dMARS. In Munindar P. Singh, Anand Rao, and Michael J. Wooldridge, editors, *Intelligent Agents IV (LNAI 1365)*, pages 155–176, 1998.

[2] Matthias Felleisen. On the expressive power of programming languages. In G. Goos and J. Hartmanis, editors, *3rd European Symposium on Programming (LNCS 432)*, pages 134–151. Springer-Verlag, 1990.

[3] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Control Structures of Rule-Based Agent Languages. *Accepted for ATAL98*, 1998.

[4] 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, 1998.

[5] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[6] David M. R. Park. *Concurrency and Automata on Infinite Sequences (LNCS 104).* Springer-Verlag, 1980.

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

[8] Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. van der Velde and J.W. Perram, editors, *Agents Breaking Away*, pages 42–55. Springer, 1996.

[9] Anand S. Rao. Private communication, March 1997.

[10] Krister Segerberg. Modal logics with linear alternative relations. *Theoria*, 36:301–322, 1970.

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