Operational semantics for agent communication languages

R. M. van Eijk, F. S. de Boer,
W. van der Hoek, J.-J. Ch. Meyer

UU-CS-1999-08

Operational Semantics for Agent Communication Languages

Rogier M. van Eijk Frank S. de Boer Wiebe van der Hoek
John-Jules Ch. Meyer
Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{rogier, frankb, wiebe, jj}@cs.uu.nl

Abstract

In this paper, we study the operational semantics of agent communication languages. We develop a general multi-agent programming language for systems of concurrently operating agents, where each agent has a mental state consisting of beliefs and goals, and the interaction between the agents proceeds via a rendezvous communication mechanism. We will thereby build upon well-understood concepts from the object-oriented programming paradigm as object classes, method invocations and object creation. The formal semantics of the language is given by means of transition rules that describe its operational behaviour. Moreover, the operational semantics closely follow the syntactic structure of the language, and hence give rise to an abstract machine to interpret the language.

1 Introduction

The research on agent-oriented programming has yielded a variety of programming languages for multi-agent systems, each of which incorporates a particular mechanism of agent communication. Many of these communication mechanisms are based on *speech act theory*, which has originally been developed as a model for human communication. A speech act is defined to be an action that a speaker performs in order to convey part of its mental state to the hearer that the act is directed to. This notion has been fruitfully adopted in agent communication languages as KQML [11] and FIPA-ACL [1], which prescribe the syntax and semantics of a collection of speech act-like messages, each of which is comprised of a performative, a content and some additional parameters as the sender and receiver of the message. Like in speech act theory, they are used to convey information about the sender's mental state and additionally, to give rise to an update of the mental state of the receiver. For instance, there is a message that can be used by a sender to inform that it believes a particular formula to be true as well as a message that can be employed to request some actions to be performed. Both agent communication languages assume an underlying communication mechanism that proceeds via asynchronous message-passing.

As was indicated by Cohen and Levesque (cf. [5]) communicative acts should be considered as *attempts* of the sending agent to get something done from the receiving agent. An important consequence of this view is that there is no guarantee that the receiving agent will actually act in accordance with the purposes the sending agent has attributed to the message. That is, it is very well possible that a receiving agent will simply ignore the message or that it

will do the opposite of what is requested for. Hence, in giving semantics to messages one should not confuse the *effects* that a message has on the receiving agent with the subsequent *reactions* taken by the receiver that it brings about. For instance, one could easily be tempted to describe the meaning of an $ask(\varphi)$ message as that upon its reception, the receiving agent should respond with a $confirm(\varphi)$ message in the situation it believes the formula φ to be true, and with a $disconfirm(\varphi)$ message if it this is not the case. In our opinion however, the reactions towards a message are not to be considered as part of the semantics of the message, but are rather consequences of the characteristics of the receiving agent. The point we advocate in this paper, is to describe the semantics of messages solely in terms of the effects they have on the mental state of the receiving agent, without giving any references to the possible subsequent reactions. For instance, the meaning of an $ask(\varphi)$ message would be that upon its reception the receiving agent administrates that it is asked whether it believes φ to hold. It will be on the basis of its altered mental state that the agent will subsequently decide what actions to perform. This behaviour is however in no way considered to be part of the semantics of the message.

1.1 Types Of Communication

In the research on intelligent agents, it is common practice to think of an agent as an entity having several different mental attitudes. These attitudes are divided into a category of *information attitudes* which are related to the information an agent has about its environment such as belief and knowledge, and secondly, a class of *pro-attitudes* which govern the actions that the agent performs in the world and hence, concern the agent's motivations. Examples of the latter category of attitudes are goals, intentions and commitments (cf. [18]). Our previous work on communication in multi-agent systems has been centred around the agents' information attitudes (cf. [8, 7, 9]). A central topic in this study was the issue of information-passing, which constitutes a form of communication in which agents exchange first-order information among each other. In this report, we take our framework one step further by additionally taking into account agent communication that concerns the pro-attitudes. That is, besides passing information about the multi-agent system, agents also communicate with each other about their motivations. In particular, we will focus on the motivations that directly stem from communication, and refer the reader to [13] for details on non-communicative motivations.

With respect to agent communication that concerns the motivational attitudes a distinction can be made between two different types of interaction. The first category comprises communication that concerns *properties* of the agent system, which are not to be considered as to give information about the current agent system but rather as *specifications* of a pursued configuration of the the agent system. An example of a communicative act that falls in this category is the KQML message $\mathbf{achieve}(\alpha, \beta, \varphi)$, which denotes a request of the agent α to the agent β to accomplish a state in which φ is true. This type of communication is however outside the scope of the current paper; we will study it in a future refinement where we take into account agent expertise in the form of a vocabulary or signature. In this setting, its expertise determines the kind of requests the agent can deal with.

The other category is given by communication that involves executable programming code to change the agent system; i.e. the communication concerns implementation rather than specification, or in other words, is procedural of nature instead of being declarative. An example of a communicative act in this category is the FIPA-ACL message $\langle i, \mathsf{request}(j, a)\rangle$,

which denotes a request of the agent i directed to the agent j to execute the action a. This type of interaction is similar to that of *higher order communication*, which is studied in the field of concurrency theory. In this paradigm, programs themselves are considered as first class communication data that can be passed among the processes in the system (cf. [15]). Upon reception of a program, the receiving process integrates it in its own program and subsequently resumes its computation. Although communication of mobile code gives rise to a very powerful computation mechanism, it also gives rise to a wide range of problems especially with respect to safety and security issues in computer systems, like for instance the unauthorised access to and manipulation of sensitive data and services (cf. [16]).

In this paper, we will not follow the road of higher order communication, but adopt a more traditional communication mechanism that has been fruitfully employed in various distributed programming paradigms. It amounts to the idea that rather than accepting and executing arbitrary programs, a process specifies a collection of programs that other processes can request it to execute.

1.2 Rendezvous and Remote Procedure Call

In the field of concurrency, there is the classical notion of a *rendezvous*, which constitutes a communication mechanism in which one process β executes one of its procedures on behalf of another process α (cf. [3]). In particular, a rendezvous can be viewed upon as to consist of three distinct steps. First, there is the call of a procedure of β by α. This is followed by the execution of this procedure by β, where the formal parameters are replaced by the actual parameters provided by α, while the execution of the calling process α is suspended. Finally, there is the exchange of the result of executing the procedure back to α, which thereupon resumes its execution. It follows that a rendezvous comprises two points of synchronisation. First, there is the call with the exchange of the actual procedure parameters from the caller to the callee and secondly, there is the communication of the results back from the callee to the caller

The notion of a rendezvous is almost equal to that of a *remote procedure call (RPC)* (cf. [4]). The difference between the two notions lies in the fact that for an RPC an entirely new process is created that takes care of the call, whereas in the case of a rendezvous the call is handled by the called process itself (cf. [3]). This implies that in the former case different procedure calls can be handled simultaneously, whereas in the latter case the calls are taken one at a time.

The rendezvous communication mechanism has been adopted in the concurrent programming language POOL (cf. [2]), which constitutes a semantically well-understood object-oriented programming language to program systems of concurrently operating objects. An object in the language is an entity that is assigned a unique identifier to distinguish it from the other objects in the system, a program that governs its behaviour and finally, a collection of methods that it can invoke itself as well as can be invoked by the other objects, the latter proceeding via a rendezvous communication mechanism.

In this paper, we outline a framework for agent communication that builds upon these object-oriented features underpinning the language POOL. The most important aspect to be accounted for is the shift from computing with expressions and values as in POOL to computing with information. That is, rather than that computations are performed in the context of a *local state* that maps variables to their associated values, one of the characteristics of the agent-oriented programming paradigm is that computations are performed relative to

a *mental state* consisting of attitudes as beliefs and goals. One of the immediate implications of this is that in adopting concepts from the object-oriented language POOL the place of expressions is to be filled in by information, which we assume to be expressed in a first-order language.

The remainder of this paper is organised as follows. In section 2, we give a formalism of deriving conclusions form first-order information stores. This is used in section 3, in which a general multi-agent programming language is defined for systems of agents that interact with each other by means of a rendezvous communication scheme. Its semantics that is subsequently defined in terms of a transition system, give a clear operational description of the programming language. We study the relation of the framework with the existing agent communication languages KQML and FIPA-ACL in section 4. Finally, we round off in section 5 by suggesting several issues for future research.

2 First-order Information

In this section, we recapitulate a mechanism of inferring conclusions from first-order information stores, which accounts for conclusions that contain free variables (cf. [10]). The idea of the inference of a formula φ from an information store B is that any free variable in φ is substituted by a closed term such that the resulting formula is a classical consequence of B, which means that it is derivable via the classical first-order consequence relation. We start by defining signatures, formulae and substitutions.

Definition 1 *(Signatures, formulae and substitutions)*

- A *signature* $\mathcal{L}$ is a tuple $\langle \mathcal{R}, \mathcal{F} \rangle$, where $\mathcal{R}$ is a collection of predicate symbols and $\mathcal{F}$ a collection of function symbols.
- The set Var is a collection of variables with typical elements x, y and z, while $ForVar$ is a collection of variables that range over formulae, with typical element ρ.
- The set $Ter(\mathcal{L})$ of terms over $\mathcal{L}$ is inductively defined by: $Var \subseteq Ter(\mathcal{L})$ and secondly, if $t_1, \ldots, t_k \in Ter(\mathcal{L})$ and $F \in \mathcal{F}$ of arity k then $F(t_1, \ldots, t_k) \in Ter(\mathcal{L})$. A term is *closed* if it contains no variables from Var.
- The sets $For(\mathcal{L})$, $Que(\mathcal{L})$ and $OpenQue(\mathcal{L})$ are defined to be the smallest sets S that satisfy the clauses (1,2), (1,2,3) and (1,2,3,4) given below, respectively.

  (1) if $t_1, t_2, \ldots, t_k \in Ter(\mathcal{L})$ and $R \in \mathcal{R}$ of arity k then $(t_1 = t_2),\ R(t_1, \ldots, t_k) \in S$,
  (2) if $\varphi, \psi \in S$ and $x \in Var$ then $\neg\varphi,\ \varphi \wedge \psi,\ \exists x\varphi \in S$
  (3) if $x \in Var$ and $\varphi \in S$ then $?x\varphi \in S$
  (4) $ForVar \subseteq S$

  The connectives $\vee$, $\rightarrow$, $\leftrightarrow$ and $\exists$ can be defined in terms of $\neg$, $\wedge$ and $\forall$ in the usual way. We assume that φ, ψ are typical elements of $For(\mathcal{L})$ and $Que(\mathcal{L})$ and υ is a typical element of $OpenQue(\mathcal{L})$.
- Finally, the set $Sub(\mathcal{L})$ is defined to be the set of formulae of the form $x = t$, where $x \in Var$ and t is a closed term in $Ter(\mathcal{L})$. The term t is referred to as a *witness* for x. Additionally, a set $\Gamma \subseteq Sub(\mathcal{L})$ is called a *substitution*.

The set $For(\mathcal{L})$ consists of the standard first-order formulae, while the set $Que(\mathcal{L})$ introduces an extra quantifier ? to bind variables. A formula of the form $?x\varphi$ denotes that x is a variable in φ that is to be substituted by a witness.[1] Additionally, the formulae in the set $OpenQue(\mathcal{L})$ can contain variables, which are placeholders that can be substituted by formulae in $Que(\mathcal{L})$.

We will not consider substitutions in the classical sense, which also comprise equalities as $x = f(f(y))$, but only those that involve equations between variables and closed terms.

We remark that we will sometimes be a bit loose in the notation of formulae and sets of formulae; i.e. whenever convenient we will let a set $\{\upsilon_1, \ldots, \upsilon_n\}$ of formulae represent the formula $\upsilon_1 \wedge \cdots \wedge \upsilon_n$, or a formula υ represent the set $\{\upsilon\}$.

Definition 2 *(Properties of substitutions)*

- A substitution $\Gamma \subseteq Sub(\mathcal{L})$ is said to be *unambiguous* if there do not exist $x \in Var$ and distinct $t_1, t_2 \in Ter(\mathcal{L})$ with $(x = t_1),\ (x = t_2) \in \Gamma$.
- Secondly, $\Gamma \subseteq Sub(\mathcal{L})$ is *complete* for a formula $\varphi \in Que(\mathcal{L})$ if for each variable $x \in Var$ that is bound by a quantifier ? in φ there exists $t \in Ter(\mathcal{L})$ such that $(x = t) \in \Gamma$.

Definition 3 For each $\psi \in Que(\mathcal{L})$ and $\Gamma \subseteq Sub(\mathcal{L})$ that is unambiguous and complete for ψ, we define the function $\otimes$ by induction on the structure of ψ:

- $P(t_1, \ldots, t_k) \otimes \Gamma = P(t_1, \ldots, t_k)$
- $(\neg\varphi) \otimes \Gamma = \neg(\varphi \otimes \Gamma)$
- $(\varphi_1 \wedge \varphi_2) \otimes \Gamma = (\varphi_1 \otimes \Gamma) \wedge (\varphi_2 \otimes \Gamma)$
- $(\exists x\varphi) \otimes \Gamma = \exists x(\varphi \otimes \Gamma)$
- $(?x\varphi) \otimes \Gamma = \exists x(x = t \wedge (\varphi \otimes \Gamma))$, where $(x = t) \in \Gamma$

The function $\otimes$ is used to substitute the variables that are bound by a quantifier ? in the formula ψ by the values given by Γ. For instance, $?xP(x) \otimes \{x = a\}$ equals $\exists x(P(x) \wedge x = a)$.

Definition 4 *(The ordering $\prec$)*
Let $\vdash$ denote the classical first-order entailment relation on $\mathcal{P}(For(\mathcal{L})) \times \mathcal{P}(For(\mathcal{L}))$. The relation $\prec$ is defined as follows:

$$\Phi \prec \Psi \Leftrightarrow \Psi \vdash \Phi \text{ and } \Phi \nvdash \Psi,$$

for all sets of formulae $\Phi, \Psi \subseteq For(\mathcal{L})$.

The relation $\prec$ gives rise to an ordering of sets of formulae in terms of their logical strength; i.e. if $\Phi \prec \Psi$ we say Ψ is logically stronger than Φ.

[1] In [10], we additionally introduce a quantifier ! that has a similar meaning; i.e. a formula of the form $!x\varphi$ also denotes that x is a variable in φ that is to be substituted by a witness. The difference between both quantifiers is that in the former case x is to be substituted by a witness that has yet to be established, whereas in the case of the quantifier ! it is to be substituted by an already established witness. The interplay of both quantifiers gives rise to a mechanism of using variables outside the scope of the formula they have been introduced in, which is an important feature to model conversations between agents. For instance, the variable x introduced in the formula $?x\varphi$ is subsequently referred to in the formula $!x\psi$. To keep things a little simple here, we restrict ourselves to the quantifier ?.

Definition 5 *(Minimal Unifiers)*

- A substitution $\Delta \subseteq Sub(\mathcal{L})$ is called a *unifier* for the pair (Φ, Ψ), if $\Phi \vdash (\Psi \otimes \Delta)$, for all $\Phi \subseteq For(\mathcal{L})$ and $\Psi \subseteq Que(\mathcal{L})$.
- Additionally, Δ is called a *minimal unifier* if there is no unifier Δ' with $\Delta' \prec \Delta$.

We use the notation $\Phi \vdash_\Delta \Psi$ to denote that Δ is a minimal unifier for (Φ, Ψ).

For instance, the substitution $\{x = a\}$ is a minimal unifier for (Φ, Ψ), where Φ is given by $\{\forall x(P(x) \rightarrow Q(x)), P(a), P(b)\}$ and Ψ equals $\{?xQ(x)\}$, and so is the substitution $\{x = b\}$. This example shows that minimal unifiers are not necessarily unique.

Finally, let *Assign* denote the set of functions $ForVar \rightarrow Que(\mathcal{L})$, which map formula variables to formulae, with typical element θ. We define a function $\cdot$ of type $(OpenQue(\mathcal{L}) \times Assign) \rightarrow Que(\mathcal{L})$ such that $\upsilon \cdot \theta$ yields the formula υ in which all formula variables have been replaced by the formula given by the function θ.

Definition 6 We define $\upsilon \cdot \theta$ by induction on the structure of υ, where we assume that θ is defined for all formula variables in υ.

- $\rho \cdot \theta = \theta(\rho)$
- $P(t_1, \ldots t_k) \cdot \theta = P(t_1, \ldots t_k)$
- $(\neg\varphi) \cdot \theta = \neg(\varphi \cdot \theta)$
- $(\varphi_1 \wedge \varphi_2) \cdot \theta = (\varphi_1 \cdot \theta) \wedge (\varphi_2 \cdot \theta)$
- $(\exists x\varphi) \cdot \theta = \exists x(\varphi \cdot \theta)$
- $(?x\varphi) \cdot \theta =?x(\varphi \cdot \theta)$

We use the notation $\theta\{\varphi/\upsilon\}$ to denote a function that yields φ on the input υ and the value yielded by the function θ on all other inputs. Additionally, we will let $\{\}$ denote the function that is undefined for all inputs.

For instance, we have that $\exists x(x = a \wedge \rho_1 \wedge \rho_2) \cdot \theta$ where $\theta = \{\}\{P(a)/\rho_2\}\{\neg Q(a)/\rho_1\}$, is equal to the formula $\exists x(x = a \wedge \neg Q(a) \wedge P(a))$.

3 A Formal Framework

In this section, we define a general programming language for multi-agent systems that covers the basic ingredients to describe the operational semantics of agent communication languages. In particular, we assume that a multi-agent system consists of a dynamic collection of agents in which new agents can be integrated, and where the interaction between the agents proceeds via a rendezvous communication scheme. Each agent is assigned a mental state comprised of a belief state and a goal state that can be inspected and updated. Additionally, an agent is assumed to be an instance of an agent class that defines the methods the agent can invoke itself as well as can be called by other agents in the system. Finally, the behaviour of an agent is governed by a program consisting of the standard programming constructs of sequential composition, non-deterministic choice, parallel composition and recursion.

3.1 Syntax

Definition 7 *(Preliminaries)*
The set *Ident* denotes the set of agent identifiers with typical elements α, β and γ, *Meth* denotes the collection of methods with typical element m, $\vdash$ denotes an entailment relation to infer conclusions from belief states and $\circ$ is an operator to update belief states. Additionally, the collection *Goal* is comprised of goal states with typical element G. A goal state is a set of tuples of the form $m(\varphi_1, \ldots, \varphi_n) \Rightarrow \alpha$, where m is a method name in *Meth* and $\varphi_1, \ldots, \varphi_n$ are formulae and α is an agent identifier in *Ident*, which represents a goal to execute the method m with actual parameters $\varphi_1, \ldots, \varphi_n$ and where α denotes the agent that the result of the execution is to be sent to.

Definition 8 *(Syntax of programming language)*
The programming language is defined by the following BNF-grammar.

$$a ::= \rho \leftarrow v \mid \rho \leftarrow a \mid \mathsf{test}(v) \mid \mathsf{update}(v) \mid \mathsf{send}(\iota x.v, m(v_1, \ldots, v_n)) \mid m(v_1, \ldots, v_n) \mid \mathsf{accept}(m_1, \ldots, m_n) \mid \mathsf{handle}(m_1, \ldots, m_n) \mid \mathsf{integrate}(C, S)$$

$$S ::= a; S \mid S_1 + S_2 \mid S_1 \,\&\, S_2 \mid \mathsf{skip}$$

$$C ::= \{m_1, \ldots, m_n\}$$

$$A ::= \langle \alpha, S, (B, G), \theta \rangle$$

$$X ::= \{A_1, \ldots, A_n\}$$

where $x \in \mathit{Var}$, $\alpha \in \mathit{Ident}$, $\rho \in \mathit{ForVar}$, $\{v, v_1, \ldots, v_n\} \subseteq \mathit{OpenQue}(\mathcal{L})$, $\theta \in \mathit{Assign}$, $\{m, m_1, \ldots, m_n\} \subseteq \mathit{Meth}$, $B \subseteq \mathit{For}(\mathcal{L})$ and $G \in \mathit{Goal}$.

If an *action* is of the form $\rho \leftarrow v$ it denotes the assignment of the formula v to the formula variable ρ, while an action of the form $\rho \leftarrow a$ stands for the assignment of the result of executing the action a to ρ. The action $\mathsf{test}(v)$ denotes the test whether the formula v follows from the belief state, while the action $\mathsf{update}(v)$ represents the update of the belief state with v. An agent executes the action $\mathsf{send}(\iota x.v, m(v_1, \ldots, v_n))$ to send the agent x that satisfies the formula v a request to execute the method m with the actual parameters $v_1, \ldots, v_n$ and to subsequently wait for the result. We remark that the agent x is not required to be unique. An agent executes the action $m(v_1, \ldots, v_n)$ to invoke this method itself. The action $\mathsf{accept}(m_1, \ldots, m_n)$ denotes the acceptance of a request to execute a method in $m_1, \ldots, m_n$, while the action $\mathsf{handle}(m_1, \ldots, m_n)$ is employed to select an invocation of one of the methods $m_1, \ldots, m_n$ from the goal state and to subsequently execute it. The difference between both actions is that the former fills the goal state whereas the latter empties it. Finally, the action $\mathsf{integrate}(C, S)$ is used to integrate an instance of the agent class C in the agent system, which thereupon will start to execute the statement S.

A *statement* is either the sequential composition $a; S$ of an action and a statement, the non-deterministic choice $S_1 + S_2$ between two statements, the parallel composition $S_1 \,\&\, S_2$ of two statements or an empty statement skip. A *class* C is defined by a set of methods. An *agent* A consists of an identifier α that distinguishes it from all other agents, a program S that governs its behaviour, a mental state (for which we will use the symbol Λ) consisting of a belief state B, a goal state G and a function θ that maps formula variables to formulae. Implicitly, we assume that each agent is an instance of an agent class, which means that the

methods the agent can invoke are those that are defined in its class. Finally, a *multi-agent system* X is a set of agents.

3.2 Operational Semantics

In this section, we define the operational semantics of the programming language.

3.2.1 Transition Systems

An elegant mechanism of defining the operational semantics of a programming language is that of a transition system. Such a system, which was originally developed by Plotkin (cf. [14]) constitutes a means to formally derive the individual computation steps of a program. In its most general form a *transition* looks as:

$$P, \sigma \longrightarrow P', \sigma'$$

where P and P' are two programs and σ and σ' are some stores of information. The transition denotes a computation step of the program P which changes the store of information σ to σ', where P' is identified to be the part of the program P that still needs to be executed.

Transitions are formally derived by means of *transition rules* of the form:

$$\frac{P_1, \sigma_1 \longrightarrow P_1', \sigma_1' \quad \cdots \quad P_n, \sigma_n \longrightarrow P_n', \sigma_n'}{P, \sigma \longrightarrow P', \sigma'}$$

Such a rule denotes that the transition below the line can be derived if the transitions above the line are derivable. Sometimes, we will write transition rules with several transitions below the line. They are used to abbreviate a collection of rules each having one of these transitions as its conclusion. A rule with no transitions above the line is called an *axiom*. A collection of transition rules defines a *transition system*.

The advantage of using transitions systems is that they allow the operational semantics to closely follow the syntactic structure of the language. As an effect, if we view the configurations P, σ as an abstract model of a machine then the transitions specify the actions that this machine can subsequently perform. In fact, this machine would act as an interpreter for the language.

3.2.2 Rules For Actions

In this section, we start the definition of a transition system for our programming language. It consists of rules of the form:

$$\frac{X \cup Y \longrightarrow X \cup Y'}{X \cup Z \longrightarrow X \cup Z'}$$

where the agent system X is required to be disjoint from the agent systems Y, Y', Z and Z'. We will however abbreviate these rules to:

$$\frac{Y \longrightarrow Y'}{Z \longrightarrow Z'}$$

In the transition system, besides agent configurations, which are of the form $\langle \alpha, S, \Lambda, \theta \rangle$, we will employ configurations that are of the form $\langle \alpha, a, \Lambda, \theta \rangle$ and $\langle \alpha, \varphi, \Lambda, \theta \rangle$ in which the second element of the tuple is not a statement but an action a or a result φ of executing an action.

We start with the rules that deal with actions.

Definition 9 *(Transitions for formula variables)*
Let $\varphi = v \cdot \theta$, we have the following transitions:

$$\{\langle \alpha, \rho \leftarrow v, \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \rho \leftarrow \varphi, \Lambda, \theta \rangle\}$$

$$\{\langle \alpha, \mathsf{test}(v), \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \mathsf{test}(\varphi), \Lambda, \theta \rangle\}$$

$$\{\langle \alpha, \mathsf{update}(v), \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \mathsf{update}(\varphi), \Lambda, \theta \rangle\}$$

$$\{\langle \alpha, \mathsf{send}(\iota x.v, \ldots), \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \mathsf{send}(\iota x.\varphi, \ldots), \Lambda, \theta \rangle\}$$

$$\{\langle \alpha, \mathsf{send}(\ldots, m(\ldots, v, \ldots)), \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \mathsf{send}(\ldots, m(\ldots, \varphi, \ldots)), \Lambda, \theta \rangle\}$$

$$\{\langle \alpha, m(\ldots, v, \ldots), \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, m(\ldots, \varphi, \ldots), \Lambda, \theta \rangle\}$$

These six transitions describe the substitution of formula variables in the formula $v \in \mathit{OpenQue}(\mathcal{L})$ by their associated value given by the function θ.

Definition 10 *(Transition rules for assignment)*

$$\frac{\{\langle \alpha, a, \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta' \rangle\}}{\{\langle \alpha, \rho \leftarrow a, \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \rho \leftarrow \varphi, \Lambda', \theta' \rangle\}}$$

$$\{\langle \alpha, \rho \leftarrow \varphi, \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda, \theta\{\varphi/\rho\} \rangle\}$$

The first rule states how the transition for $\rho \leftarrow a$ can be derived from the transition for a, viz. the result φ of executing a is assigned to the formula variable ρ. The second rule then shows that the assignment $\rho \leftarrow \varphi$ results in an update of θ such that it yields φ for the input ρ. We take the result of executing $\rho \leftarrow \varphi$ to be φ in order to model nested assignments as for instance $\rho' \leftarrow (\rho \leftarrow \varphi)$, which assigns φ to both ρ and ρ'.

Definition 11 *(Transition for test)*
Let $\mathcal{S} = \{\Delta_1, \ldots, \Delta_n\}$ be the set of distinct minimal unifiers for the pair (B, φ). We have the following transition:

$$\{\langle \alpha, \mathsf{test}(\varphi), \Lambda, \theta \rangle\} \longrightarrow \{\langle \alpha, \psi, \Lambda, \theta \rangle\},$$

where $\Lambda = (B, G)$. We distinguish the following options for ψ:

1. a substitution Δ, where $\Delta \in \mathcal{S}$
2. a formula $\varphi \otimes \Delta$, where $\Delta \in \mathcal{S}$
3. the formula $\bigwedge_{1 \leq i \leq n} (\varphi \otimes \Delta_i)$

The choices above model different interpretations of the test statement. The first is one in which the result of the test is taken to be a derivable minimal unifier, provided that such a unifier exists. It is either the empty substitution denoting that the formula φ is a classical

consequence of the belief state, or a non-empty substitution defining a witness for each variable in φ bound by a quantifier ?. For instance, the following transition is derivable:

$$\{\langle \alpha, \mathsf{test}(?xP(x)), \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, x = a, \Lambda, \theta\rangle\}$$

where $\Lambda = (P(a) \wedge P(b), G)$. Note that this transition is not necessarily unique, as shown by the following transition that is derivable as well:

$$\{\langle \alpha, \mathsf{test}(?xP(x)), \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, x = b, \Lambda, \theta\rangle\}$$

The second option is to apply the derived substitution to φ and yield this formula as the result of test. In this case, the transition looks like:

$$\{\langle \alpha, \mathsf{test}(?xP(x)), \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, P(a), \Lambda, \theta\rangle\}$$

Finally, there is the option to yield an exhaustive description comprised of all instances of the formula φ. The transition is then given by:

$$\{\langle \alpha, \mathsf{test}(?xP(x)), \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, P(a) \wedge P(b), \Lambda, \theta\rangle\}$$

We refer to the first option as the *substitution* interpretation, while the latter two are called the *derive-one* and the *derive-all* interpretations, respectively. Unless indicated otherwise, we will in this paper assume that the derive-one interpretation is in force.

Definition 12 *(Transition for update)*

$$\{\langle \alpha, \mathsf{update}(\varphi), (B, G), \theta\rangle\} \longrightarrow \{\langle \alpha, \mathit{true}, (B \circ \varphi, G), \theta\rangle\}$$

The transition for updates amounts to the incorporation of the formula φ in the agent's belief state. It is required that φ is a formula in $\mathit{For}(\mathcal{L})$, that is, it does not contain the quantifier ?. We will not go into the details of the operator $\circ$, we assume it to be a parameter of the framework (see [12] for more details on belief revision). We take the formula *true* as the result of the update.

Definition 13 *(Transition for sending a message)*
Provided that $B_1 \vdash_{\{x=\alpha_2\}} ?x\varphi$ holds we have the transition:

$$\{\langle \alpha_1, \mathsf{send}(\iota x.\varphi, m(\varphi_1, \ldots, \varphi_n)), \Lambda_1, \theta_1\rangle, \langle \alpha_2, \mathsf{accept}(\ldots, m, \ldots), \Lambda_2, \theta_2\rangle\} \longrightarrow$$
$$\{\langle \alpha_1, \mathsf{wait}(\alpha_2), \Lambda_1, \theta_1\rangle, \langle \alpha_2, \mathit{true}, \Lambda_2', \theta_2\rangle\}$$

where $\Lambda_1 = (B_1, G_1)$ and $\Lambda_2 = (B_2, G_2)$ and $\Lambda_2' = (B_2, G_2 \cup \{m(\varphi_1, \ldots, \varphi_n) \Rightarrow \alpha_1\})$

In the classical notion of a rendezvous, the computation step of the agent α_2 that follows the first synchronisation, would be the execution of the method invocation $m(\varphi_1, \ldots, \varphi_n)$. However, as mentioned before, a crucial characteristic of agent-oriented programming is that computations are performed relative to a mental state. Hence, the decision to execute the method invocation should be based upon this state rather than that it is executed without any regard for the agent's current beliefs and goals. This is why the invocation $m(\varphi_1, \ldots, \varphi_n)$ is not executed immediately but added to the agent's goal state, along with the identifier α_1

representing the agent that the result of the invocation is to be sent back to. The construct $\mathsf{wait}(\alpha_2)$ is not part of the syntax of the programming language. It denotes that the agent α_1 is waiting for the agent α_2 to return the result of the invocation and is used to simplify the operational description of the rendezvous communication scheme.[2]

Finally, the construct $\iota x.\varphi$ denotes a witness for x such that φ, which we require to be an element of $For(\mathcal{L})$, is true for it. In the above transition the condition $B_1 \vdash_{\{x=\alpha_2\}} ?x\varphi$ requires the witness for x to be α_2. For instance, we have the derivation $\{Agent(\alpha_2)\} \vdash_{\{x=\alpha_2\}} ?xAgent(x)$, where the predicate $Agent(x)$ is used to express that the agent x is part of the multi-agent system (see also definition 17).

Definition 14 *(Transition for method invocation)*
If m is a method declared by $m(\rho_1, \ldots, \rho_n) :- \ T$ we have the transition:

$$\{\langle \alpha, m(\varphi_1, \ldots, \varphi_n), \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S \Rightarrow \alpha, \Lambda, \theta\rangle\},$$

where S equals $T[\varphi_1/\rho_1, \cdots, \varphi_n/\rho_n]$, which denotes the body statement T of m in which the actual parameters φ_i have been simultaneously substituted for the formal parameters ρ_i.

Note that in comparison with standard concurrent programming, the parameter-passing mechanism in our framework is rather high-level, as formulae themselves constitute first-class values with which methods can be invoked. The construct $S \Rightarrow \alpha$ denotes that the result of executing the statement S should be sent back to the agent α.

Definition 15 *(Transition for handling goals)*
If $m(\varphi_1, \ldots, \varphi_n) \Rightarrow \beta \in G$, and m is declared as $m(\rho_1, \ldots, \rho_n) :- \ T$ then:

$$\{\langle \alpha, \mathsf{handle}(\ldots, m, \ldots), \Lambda, \theta, \rangle\} \longrightarrow \{\langle \alpha, S \Rightarrow \beta, \Lambda', \theta\rangle\},$$

where $S = T[\varphi_1/\rho_1, \cdots, \varphi_n/\rho_n]$ and $\Lambda = (B, G)$ and $\Lambda' = (B, G \setminus \{m(\varphi_1, \ldots, \varphi_n) \Rightarrow \beta\})$

The presence of goal states yields the need for a mechanism that controls the selection and execution of goals, which is a mechanism that is not present in the traditional concurrent language POOL. The above transition reflects a straightforward approach in which one of the invocations of a method m is taken from the goal state and identified to be subsequently executed.

Definition 16 *(Transitions for returning the result)*

$$\{\langle \alpha_1, \mathsf{wait}(\alpha_2), \Lambda_1, \theta_1\rangle, \langle \alpha_2, \varphi \Rightarrow \alpha_1, \Lambda_2, \theta_2\rangle\} \longrightarrow$$
$$\{\langle \alpha_1, \varphi, \Lambda_1, \theta_1\rangle, \langle \alpha_2, \mathit{true}, \Lambda_2, \theta_2\rangle\}$$

$$\{\langle \alpha, \varphi \Rightarrow \alpha, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda, \theta\rangle\}$$

[2] To keep things a little simple, we assume that the agent α_1 cannot concurrently invoke more than one method of the agent α_2, which implies that the result of the method invocation will be substituted for this occurrence of the $\mathsf{wait}(\alpha_2)$ construct and not for another occurence somewhere else in the program. Without this assumption we would need to associate with each rendezvous a unique identifier that is maintained along the computation steps of the rendezvous.

If a computation of a method invocation has terminated with a result φ then the second synchronisation of the rendezvous takes place, in which this result is communicated back to the agent α_1. The second rule deals with the case that the method invocation has been executed on behalf of the agent itself.

Definition 17 *(Transition for integration)*

$$\{\langle \alpha, \mathsf{integrate}(C, S), \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \mathit{Agent}(\beta), \Lambda, \theta\rangle, \langle \beta, S, \emptyset, \{\}\rangle\},$$

where β is a fresh agent identifier that does not occur in the agent system, $\emptyset$ denotes the empty mental state and $\{\}$ the function that is undefined for any input.

The transition rule defines the integration of an instance of the agent class C. It shows how the agent system consisting of the agent α is expanded with a new agent β that starts its execution with the statement S where its initial mental state is defined to be the empty one. The methods that the integrated agent can invoke are given by the methods of C. We will postpone the integration of agents with non-empty mental states to the future extension of the framework in which we take into account agent expertise in the form of a vocabulary or signature. A possible approach is to let the integrated agent inherit those beliefs and goals of its creator that are expressed in its expertise signature.

The result of the integration is formulated by the information $\mathit{Agent}(\beta)$, which expresses that the agent with identifier β is part of the multi-agent system.

3.2.3 Rules for Statements

Definition 18 *(Transition rules for the construct $S \Rightarrow \beta$)*

$$\frac{\{\langle \alpha, S, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S', \Lambda', \theta'\rangle\}}{\{\langle \alpha, S \Rightarrow \beta, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S' \Rightarrow \beta, \Lambda', \theta'\rangle\}}$$

$$\frac{\{\langle \alpha, S, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta'\rangle\}}{\{\langle \alpha, S \Rightarrow \beta, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi \Rightarrow \beta, \Lambda', \theta'\rangle\}}$$

The transition for the construct $S \Rightarrow \beta$ can be derived from the transition for the statement S. The first transition rule deals with the case that S does not terminate after one computation step: S' denotes the part of S that still needs to be executed, while the second rule deals with the case that S terminates with a result φ.

Definition 19 *(Transition rule for sequential composition)*

$$\frac{\{\langle \alpha, a, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta'\rangle\}}{\{\langle \alpha, a; S, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S, \Lambda', \theta'\rangle\}}$$

The transition for the sequential composition of an action a and a statement S can be derived from the transition for the action a. Note that the result φ of executing a is simply ignored, because it (possibly) has already been processed in Λ' and θ'. For instance, we can derive the transition $\{\langle \alpha, (\rho \leftarrow \varphi); S, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S, \Lambda, \theta\{\varphi/\rho\}\rangle\}$.

Definition 20 *(Transition rules for non-deterministic choice)*

$$\frac{\{\langle \alpha, S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_1', \Lambda', \theta'\rangle\}}{\begin{array}{c}\{\langle \alpha, S_1 + S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_1', \Lambda', \theta'\rangle\} \\ \{\langle \alpha, S_2 + S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_1', \Lambda', \theta'\rangle\}\end{array}}$$

$$\frac{\{\langle \alpha, S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta'\rangle\}}{\begin{array}{c}\{\langle \alpha, S_1 + S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta'\rangle\} \\ \{\langle \alpha, S_2 + S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta'\rangle\}\end{array}}$$

The transition for the non-deterministic choice $S + T$ between two statements can be derived from the transition of either S or T. The second rule deals with the terminating case.

Definition 21 *(Transition rules for internal parallelism)*

$$\frac{\{\langle \alpha, S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_1', \Lambda', \theta'\rangle\}}{\begin{array}{c}\{\langle \alpha, S_1 \ \& \ S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_1' \ \& \ S_2, \Lambda', \theta'\rangle\} \\ \{\langle \alpha, S_2 \ \& \ S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_2 \ \& \ S_1', \Lambda', \theta'\rangle\}\end{array}}$$

$$\frac{\{\langle \alpha, S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \varphi, \Lambda', \theta'\rangle\}}{\begin{array}{c}\{\langle \alpha, S_1 \ \& \ S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_2, \Lambda', \theta'\rangle\} \\ \{\langle \alpha, S_2 \ \& \ S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, S_2, \Lambda', \theta'\rangle\}\end{array}}$$

The internal parallel composition $S \ \& \ T$ of two statements is modeled by means of an interleaving of the computation steps of S and T. The second transition rule deals with the terminating case, in which the result φ is ignored.

Definition 22 *(Transition rule for skip statement)*

$$\{\langle \alpha, \mathsf{skip}, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \mathit{true}, \Lambda, \theta\rangle\}$$

The statement skip yields the result *true* and thereby leaves the mental state Λ and the function θ unchanged.

3.2.4 Example

Let us consider a small example that illustrates the transitions system developed in the previous sections, as well as hints at the introduction of inheritance mechanisms in the framework, which is a topic that is outside the scope of the present paper and constitutes one of the issues for future refinements.

Let X be an agent system (used in a library, for instance) consisting of a client agent α and additionally two server agents β and γ. Consider the situation that the client α is looking for a biography of Elvis Presley and hence, asks the serving agent β for the name of an appropriate document. After the reception of the request, the server β has the option to examine its own information store (belief state) for the name of a suitable document, but as this information is not available, it passes the question along to the other serving agent γ. This agent in turn inspects its information store, yielding the information that the document b constitutes a biography of Elvis. Subsequently, this information is sent to the agent β that in turn passes it back to α.

In this example, we introduce two agent classes: the class C_1 consisting of the methods

- $answer :-\ \ (\mathsf{accept}(ask) + \mathsf{handle}(ask)); answer$
- $ask(\rho) :-\ \ \mathsf{test}(\rho)$

and the class C_2 that inherits the methods from C_1, where however the definition of the method *ask* is specialised to

$$ask(\rho) :-\ \ \mathsf{test}(\rho) + \mathsf{send}(\iota x.Agent(x), ask(\rho))$$

The method *answer* is a recursive procedure in which in each round either a new *ask*-message is accepted or an *ask*-message is selected from the goal state and subsequently executed. Additionally, the first definition of the method *ask* corresponds to a test on the belief state, while the second allows for the choice to send the message along to another agent.

We assume that the serving agent γ belongs to the class C_1 and that β belongs to the class C_2. Additionally, we use the predicate P to denote that a document is a biography of Elvis Presley. Consider the following initial configuration of the agent system X:

$\{\langle \alpha, \rho \leftarrow \mathsf{send}(\beta, ask(?xP(x))); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, answer, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, answer, \Lambda_3, \theta_3\rangle\},$

where $\Lambda_1 = (\emptyset, \emptyset)$, $\Lambda_2 = (\{\neg P(a), Agent(\gamma)\}, \emptyset)$, $\Lambda_3 = (\{P(b)\}, \emptyset)$ and $\theta_1 = \theta_2 = \theta_3 = \{\}$.

The agent α thus asks the agent β for the information $?xP(x)$ and subsequently adds the result, which is assigned to the formula variable ρ, to its belief state. The following constitutes a derivable computation that starts with the above configuration.

$\longrightarrow$
$\{\langle \alpha, \rho \leftarrow \mathsf{send}(\alpha, ask(?xP(x))); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{accept}(ask) + \mathsf{handle}(ask)); answer, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, answer, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, answer, \Lambda_2', \theta_2\rangle,$
$\langle \gamma, answer, \Lambda_3, \theta_3\rangle\} \longrightarrow$
where $\Lambda_2' = (\{\neg P(a), Agent(\gamma)\}, \{ask(?xP(x)) \Rightarrow \alpha\})$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{accept}(ask) + \mathsf{handle}(ask)); answer, \Lambda_2', \theta_2\rangle,$
$\langle \gamma, answer, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{test}(?xP(x)) + \mathsf{send}(\iota x.Agent(x), ask(?xP(x))) \Rightarrow \alpha); answer, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, answer, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{test}(?xP(x)) + \mathsf{send}(\iota x.Agent(x), ask(?xP(x))) \Rightarrow \alpha); answer, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, (\mathsf{accept}(ask) + \mathsf{handle}(ask)); answer, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{wait}(\gamma) \Rightarrow \alpha); \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, \mathit{answer}, \Lambda_3', \theta_3\rangle\} \longrightarrow$
where $\Lambda_3' = (\{P(b)\}, \{\mathit{ask}(?xP(x)) \Rightarrow \beta\})$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{wait}(\gamma) \Rightarrow \alpha); \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, (\mathsf{accept}(\mathit{ask}) + \mathsf{handle}(\mathit{ask})); \mathit{answer}, \Lambda_3', \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{wait}(\gamma) \Rightarrow \alpha); \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, (\mathsf{test}(?xP(x)) \Rightarrow \beta); \mathit{answer}, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (\mathsf{wait}(\gamma) \Rightarrow \alpha); \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, (P(b) \Rightarrow \beta); \mathit{answer}, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow \mathsf{wait}(\beta); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, (P(b) \Rightarrow \alpha); \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, \mathit{answer}, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \rho \leftarrow P(b); \mathsf{update}(\rho), \Lambda_1, \theta_1\rangle,$
$\langle \beta, \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, \mathit{answer}, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \mathsf{update}(\rho), \Lambda_1, \theta_1\{P(b)/\rho\}\rangle,$
$\langle \beta, \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, \mathit{answer}, \Lambda_3, \theta_3\rangle\} \longrightarrow$

$\{\langle \alpha, \mathit{true}, \Lambda_1', \theta_1\{P(b)/\rho\}\rangle,$
$\langle \beta, \mathit{answer}, \Lambda_2, \theta_2\rangle,$
$\langle \gamma, \mathit{answer}, \Lambda_3, \theta_3\rangle\} \longrightarrow$
where $\Lambda_1' = (\{\} \circ P(b), \emptyset)$

3.2.5 Rules For Failure

One important aspect we have not considered thusfar is that in the current system a method invocation might fail. For instance, there is the action $\mathsf{test}(\varphi)$ that yields such a situation in case φ is not a consequence of the belief state.

The subsequent transitions deal with the situation of failure, for which we use a special symbol δ. We extend the language $\mathit{OpenQue}(\mathcal{L})$ with a new formula of the form $\perp$, which will be used to express that a method invocation has failed.

Example 23
A way to deal with failure situations is given by the following statement:

$$\rho \leftarrow \mathsf{send}(\alpha, m(v_1, \ldots, v_n)); ((\mathsf{test}(\rho \leftrightarrow \perp); S_1) + (\mathsf{test}(\neg(\rho \leftrightarrow \perp)); S_2))$$

If the execution of $\mathsf{send}(\alpha, m(v_1, \ldots, v_n))$ results in a failure situation and hence, its result is equal to $\perp$ then S_1 is subsequently executed and otherwise the statement S_2 is executed.

Definition 24 *(Failure rule for test)*
If there is no substitution $\Delta \in Sub(\mathcal{L})$ with $B \vdash_\Delta \varphi$ then

$$\{\langle \alpha, \mathsf{test}(\varphi), (B, G), \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, (B, G), \theta\rangle\}$$

The transition defines the action $\mathsf{test}(\varphi)$ to fail in case the formula φ is not a consequence of the belief state B.

Definition 25 *(Failure rule for assignment)*

$$\frac{\{\langle \alpha, a, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}{\{\langle \alpha, \rho \leftarrow a, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}$$

This and the following transition rules deal with the propagation of a failure. The rule for assignment states that in case the action a leads to a failure then so does the action $\rho \leftarrow a$.

Definition 26 *(Failure rules for returning the result)*

$$\{\langle \alpha_1, \mathsf{wait}, \Lambda_1, \theta_1\rangle, \langle \alpha_2, \delta \Rightarrow \alpha_1, \Lambda_2, \theta_2\rangle\} \longrightarrow \{\langle \alpha_1, \perp, \Lambda_1, \theta_1\rangle, \langle \alpha_2, \perp, \Lambda_2, \theta_2\rangle\}$$

$$\{\langle \alpha, \delta \Rightarrow \alpha, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \perp, \Lambda, \theta\rangle\}$$

If a method invocation results in a failure then the result of the invocation is taken to be the special formula $\perp$.

Definition 27 *(Failure rule for the construct $S \Rightarrow \beta$)*

$$\frac{\{\langle \alpha, S, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}{\{\langle \alpha, S \Rightarrow \beta, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta \Rightarrow \beta, \Lambda, \theta\rangle\}}$$

If the statement S results in a failure then it is to be propagated to the agent β.

Definition 28 *(Failure rule for sequential composition)*

$$\frac{\{\langle \alpha, a, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}{\{\langle \alpha, a; S, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}$$

A sequential composition yields a failure in case the first action of this composition fails.

Definition 29 *(Failure rules for non-deterministic choice)*

$$\frac{\{\langle \alpha, S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\} \quad \{\langle \alpha, S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}{\{\langle \alpha, S_1 + S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}$$

This rule shows that a non-deterministic choice between two statements leads to a failure situation in case both statements lead to such a situation.

Definition 30 *(Failure rules for internal parallelism)*

$$\frac{\{\langle \alpha, S_1, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\} \quad \{\langle \alpha, S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}{\{\langle \alpha, S_1 \ \& \ S_2, \Lambda, \theta\rangle\} \longrightarrow \{\langle \alpha, \delta, \Lambda, \theta\rangle\}}$$

The failure rule for internal parallelism is similar to that for non-deterministic choice.

4 Related Work

The research on multi-agent systems has resulted in the development of various agent communication languages, none of which however has yet been assigned a satisfactory formal semantics (cf. [17]). And it is this lack of having a clear semantics that in our opinion constitutes one of the major hindrances for an agent communication language to become widely accepted. In this section, we consider the relation of our framework with two of these languages, viz. the agent communication languages KQML and FIPA-ACL.

4.1 KQML

The Knowledge Query and Manipulation Language (KQML) provides a language for the exchange of knowledge and information in multi-agent systems (cf. [11]), which has been developed with the objective of being widely accepted as a standardisation for agent communication. It defines the format of a collection of messages that are exchanged between communicating agents: a KQML message is comprised of a *performative* that indicates the purpose of the message, the actual *content* of the message expressed in some representation language and several *optional arguments* which describe meta-information that will be used in routing the message. These optional arguments include the sender of the message, the intended receiver of it, a name that identifies the message, the ontology that is used in the content of the message and finally, the language in which the content of the message is expressed.

The semantics of a KQML message are given by the following three ingredients:

1. a *precondition* on the mental state of the sending and the receiving agent that should hold *before* the dispatch and reception of the message, respectively.

2. a *postcondition* on the mental state of the sending and the receiving agent that should hold *after* the dispatch and reception of the message, respectively.

3. a *completion* condition that should hold after a conversation has taken place of which this message was a constituent.

The language in which these conditions are expressed consists of logical combinations of the following five operators: $bel(\alpha, \varphi)$ denoting that φ is in the knowledge base of α and $know(\alpha, \varphi)$, $want(\alpha, \varphi)$ and $intend(\alpha, \varphi)$, standing for the fact that α knows φ, wants φ and is committed to φ, respectively. Finally, there is an operator $process(\alpha, m)$ denoting that the message m will be processed by the agent α.

An important feature of the framework is that the communication of a message is not to be considered as an action that occurs in isolation, but that it takes place in the context of a conversation being comprised of a sequence of communications. In this wider context, the semantics of the KQML messages can be thought of as to define what constitutes a correct conversation. That is, the precondition of a message determines the collection of messages that are allowed to precede the message, while the postcondition lays down the messages that are permitted to succeed the message. Additionally, in case the completion condition is a logical consequence of the postcondition, the conversation can be identified to have successfully terminated. For instance, the following sequence of message constitutes a typical conversation:

$$\textbf{advertise}(\alpha, \beta, \textbf{ask-if}(\beta, \alpha, \varphi)),\ \textbf{ask-if}(\beta, \alpha, \varphi) \text{ and } \textbf{tell}(\alpha, \beta, \varphi)$$

In fact, as we will see it constitutes the sequential composition of two completed conversations. Let us examine the constituents of this conversation in more detail. First, the pre-, post- and completion conditions for a message **advertise**(α, β, m) are as follows, where in this case m is given by **ask-if**(β, α, φ):

1. $intend(\alpha, process(\alpha, m))$
2. $know(\alpha, know(\beta, intend(\alpha, process(\alpha, m))))\wedge$

   $know(\beta, intend(\alpha, process(\alpha, m)))$
3. $know(\beta, intend(\alpha, process(\alpha, m)))$

The meaning of the message **advertise**(α, β, m) amounts to letting the agent β know that α has the intention to process the message m. Additionally, the fact that the completion condition coincides with the postcondition, implies that this message constitutes a conversation by itself.

In our framework, we have a slightly different mechanism: there is the primitive of the form $\mathsf{accept}(m_1, \ldots, m_n)$, which reflects the agent's intention to accept a message that is in the collection $m_1, \ldots, m_n$. It has the advantage above the KQML message **advertise**(α, β, m) that it allows the set of acceptable messages to vary over time, as it only specifies the messages that are *currently* accepted. That is, a subsequent occurrence of the primitive in the agent program might specify a sub-, super- or even a completely disjoint set of acceptable messages.

Secondly, the conditions for a message **ask-if**(β, α, φ) are as follows:

1. $\bigvee_{\psi\in\Gamma}(want(\beta, know(\beta, \psi)))\wedge$
   $know(\beta, intend(\alpha, process(\alpha, m)))\wedge$
   $intend(\alpha, process(\alpha, m))$
2. $\bigvee_{\psi\in\Gamma}(intend(\beta, know(\beta, \psi)))\wedge$
   $know(\alpha, \bigvee_{\psi\in\Gamma}(want(\beta, know(\beta, \psi))))$
3. $\bigvee_{\psi\in\Gamma}(know(\beta, \psi))$

where m is given by **ask-if**(β, α, φ) and Γ equals $\{bel(\alpha, \varphi), bel(\alpha, \neg\varphi), \neg bel(\alpha, \varphi)\}$.

The meaning of the message **ask-if**(β, α, φ) amounts to letting the agent α know that β wants to know whether α believes φ, believes $\neg\varphi$ or does not believe φ. The second and third conjunct of the precondition reflect the requirement that the message is to be preceded by an **advertise**(α, β, m) message. Additionally, the completion condition indicates that the conversation of which this message is a constituent, has not successfully terminated until the agent β knows one of the formulae in Γ. As KQML abstracts away from the content language of messages, it is not clear to us why the formula $bel(\alpha, \neg\varphi)$ is an element of the set Γ, as it refers to an operator $\neg$ in the content language (which in principle, does not need to be present at all). In particular, if the agent β wants to know whether α believes $\neg\varphi$ it can use the message **ask-if**$(\beta, \alpha, \neg\varphi)$.

Thirdly, the conditions for the message **tell**(α, β, φ) are as follows:

1. $bel(\alpha, \varphi)\wedge$
   $know(\alpha, want(\beta, know(\beta, bel(\alpha, \varphi))))\wedge$
   $intend(\beta, know(\beta, bel(\alpha, \varphi)))$

2. $know(\alpha, know(\beta, bel(\alpha, \varphi))) \wedge$
   $know(\beta, bel(\alpha, \varphi))$

3. $know(\beta, bel(\alpha, \varphi))$

The meaning of the message is to let the agent β know that α believes the formula φ to be true. The first conjunct of the precondition states that α should indeed believe the formula φ, where the second and third conjunct should indicate that the message is to be preceded by a message of the form **ask-if**(β, α, φ). Here we see why it is so important to have a formal framework to give semantics to a language: mistakes are easily made. The postcondition $\bigvee_{\psi \in \Gamma}(intend(\beta, know(\beta, \psi)))$ where $\Gamma = \{bel(\alpha, \varphi), bel(\alpha, \neg\varphi), \neg bel(\alpha, \varphi)\}$ of the *ask-if* message, does not entail the precondition $intend(\beta, know(\beta, bel(\alpha, \varphi)))$ of the *tell* message. Hence, with this semantics the reception of an **ask-if**(β, α, φ) message is not sufficient to be able to send a **tell**(α, β, φ) message. Additionally, the nesting of knowledge operators seems to be quite arbitrary: it is unclear why the nesting of knowledge operators in the postcondition is restricted to depth two, and for instance does not include a formula as $know(\beta, know(\alpha, know(\beta, bel(\alpha, \varphi))))$.

Our concern with defining the semantics of communication in the manner as employed for KQML, is that it still does not yield an exact meaning of agent communication. The KQML semantics is defined in terms of operators for which the semantics themselves remain undefined. For instance, it is not clear what constitutes the exact difference between the operators *want* and *intend*.

Moreover, in contrast to the approach employed in this paper, there is a gap between the syntactic structure of the language and its semantics, which is due to the use of rather high-level operators. In our framework however, the operational semantics closely follow the syntactic structure of the language, which opens up the possibility of developing an interpreter for the language on the basis of this operational description. This is due to the fact that the configurations of an agent system can be seen as an abstract model of a machine, where its transitions define the actions that the machine can perform. In fact this machine would act as an interpreter for the language.

Thirdly, we believe that the KQML semantics impose too strong requirements on the agents with respect to their reactions towards incoming messages. We believe that these reactions are agent-dependent and hence should not be part of the semantics of messages. For instance, in our framework, consider three different agents each having one the following definitions of the method $ask(\rho)$:

- $ask(\rho) :- \ \mathsf{test}(\rho) + \mathsf{test}(\neg\rho)$
- $ask(\rho) :- \ \rho' \leftarrow \mathsf{test}(\rho); \rho'' \leftarrow \neg\rho'$
- $ask(\rho) :- \ \mathsf{send}(\iota x.Agent(x), ask(\rho))$

The first agent tests its belief state and checks whether it entails the formula ρ or its negation. This corresponds to the reaction imposed by the KQML semantics for the **ask-if**(ρ) message. On the other hand, the second agent checks whether its belief state entails φ and subsequently delivers the negation of what it believes to hold (note that the result of the sequential composition of two actions is given by the result of the second action). In KQML this reaction is simply ruled out. However, in our opinion this reaction is not a result of the semantics of the message but of the characteristics of the receiving agent. We believe that

any reaction should be allowed, as for instance shown by the third agent, which simply passes the message along to another agent and delivers the result that it receives from this agent.

4.2 FIPA-ACL

Besides the language KQML there is a second proposal for a standard agent communication language, which is developed by the organisation FIPA. This language, which we will refer to as the FIPA-ACL, also prescribes the syntax and semantics of a collection of message types (cf. [1]). The format of these messages is almost equal to that of KQML, while their semantics are given by means of:

1. a precondition on the mental state of the sending agent that should hold prior to the dispatch of the message
2. a representation of the effect that the sender can expect as a result of the dispatch

There are four primitive messages; viz. $\langle i, \mathsf{inform}(j, \varphi)\rangle$ and $\langle i, \mathsf{confirm}(j, \varphi)\rangle$ in which the agent i tells the agent j that it believes the formula φ to hold and a message $\langle i, \mathsf{disconfirm}(j, \varphi)\rangle$ in which the agent i tells the agent j that it believes the negation of φ to hold. The difference between the inform and confirm message is that the former is employed in case agent i has no beliefs about the beliefs of j concerning φ, i.e. it does not believe that j believes φ or its negation or is uncertain about φ or its negation, whereas the latter is used in case i beliefs that j is uncertain about j. Thirdly, the disconfirm message is used for situations in which i believes the negation of φ and additionally that j believes φ or is uncertain about φ. This is summerised in the table below, which also shows that the effect of informing φ, confirming φ and disconfirming $\neg\varphi$ is the same.

primitive	precondition	effect
$\langle i, \mathsf{inform}(j, \varphi)\rangle$	$B_i\varphi \wedge \neg B_i(B_j\varphi \vee B_j\neg\varphi \vee U_j\varphi \vee U_j\neg\varphi)$	$B_j\varphi$
$\langle i, \mathsf{confirm}(j, \varphi)\rangle$	$B_i\varphi \wedge B_iU_j\varphi$	$B_j\varphi$
$\langle i, \mathsf{disconfirm}(j, \varphi)\rangle$	$B_i\neg\varphi \wedge B_i(B_j\varphi \vee U_j\varphi)$	$B_j\neg\varphi$

The fourth primitive message is of the form $\langle i, \mathsf{request}(j, a)\rangle$ in which i requests the agent j to perform the action a. The condition for this message is that i believes that the agent j will be the only agent performing a and that it does not believe that j has already an intention of doing a. Additionally, the part of the precondition of a that concerns the mental attitudes of i should additionally hold. All other message are defined in terms of these primitives together with the operators ; for sequential composition and | for non-deterministic choice. An example of a composite message is $\langle i, \mathsf{query\text{-}if}(j, \varphi)\rangle$, which is an abbreviation for: $\langle i, \mathsf{request}(j, \langle j, \mathsf{inform}(i, \varphi)\rangle \mid \langle j, \mathsf{inform}(i, \neg\varphi)\rangle)$. Analogous to KQML there remains a gap between the syntax of the communication language and the semantic description given in terms of high-level modal operators as those for intentions, nested belief and uncertainty. We think however that the operational model outlined in this paper, could act as a first step in the development of an operational description of the FIPA-ACL.

5 Future Research

In this paper, we have outlined a basic programming language for systems of communicating agents that interact with each other via a rendezvous communication scheme. In subsequent research, we will study the extension of the framework with a notion of agent expertise in the form of a vocabulary or signature together with the incorporation of object-oriented features as subtyping and inheritance (cf. [6]). Inheritance would then not be restricted to the inheritance of methods but could also involve the inheritance of (parts of) mental states. Another issue is the study in what way the current framework could be used to develop an operational semantic model for existing agent communication languages as KQML and FIPA-ACL.

References

[1] Foundation For Intelligent Physical Agents. Fipa'97 specification part 2 – agent communication language. Version dated 10th October 1997.

[2] P.H.M. America, J. de Bakker, J.N. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 194–208, St. Petersburg Beach, Florida, 1986.

[3] G.R. Andrews. *Concurrent Programming, Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.

[5] P. Cohen and H. Levesque. Communicative actions for artificial agents. In *Proceedings of the First International Conference on Multi-Agent Systems*, 1995.

[6] L. Crnogorac, A.S. Rao, and K. Ramamohanarao. Analysis of inheritance mechanisms in agent-oriented programming. In Martha Pollack, editor, *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 647–652, Nagoya, Japan, 1997. Morgan Kaufmann Publishers, Inc.

[7] R. M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Information-passing and belief revision in multi-agent systems. In J. P. M. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of 5th International Workshop on Agent Theories, Architectures, and Languages (ATAL'98)*, volume 1555 of *Lecture Notes in Artificial Intelligence*, pages 29–45. Springer-Verlag, Heidelberg, 1999.

[8] R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. A language for modular information-passing agents. In K. R. Apt, editor, *CWI Quarterly, Special issue on Constraint Programming*, volume 11, pages 273–297. CWI, Amsterdam, 1998.

[9] R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Systems of communicating agents. In Henri Prade, editor, *Proceedings of the 13th biennial European Conference on Artificial Intelligence (ECAI-98)*, pages 293–297, Brighton, UK, 1998. John Wiley & Sons, Ltd.

[10] R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Open multi-agent systems: Agent communication and integration. Technical report, Universiteit Utrecht, Department of Computer Science, 1999.

[11] T. Finin, D. McKay, R. Fritzson, and R. McEntire. KQML: An Information and Knowledge Exchange Protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing*. Ohmsha and IOS Press, 1994.

[12] P. Gärdenfors. *Knowledge in flux: Modelling the dynamics of epistemic states*. Bradford books, MIT, Cambridge, 1988.

[13] K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. A formal semantics for an abstract agent programming language. In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, *Proceedings of Fourth International Workshop on Agent Theories, Architectures and Languages (ATAL'97)*, volume 1365 of *LNAI*, pages 215–229. Springer-Verlag, 1998.

[14] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[15] B. Thomsen. A calculus of higher order communicating systems. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 143–153, 1989.

[16] V. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[17] M. Wooldridge. Verifiable semantics for agent communication languages. In *Proceedings 3rd International Conference on Multi-Agent Systems (ICMAS'98)*, pages 349–356, Los Alamitos, California, 1998. IEEE Computer Society.

[18] M. Wooldridge and N. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.