Open Multi-Agent Systems: Agent Communication and Integration

Rogier M. van Eijk,
Frank S. de Boer,
Wiebe van der Hoek and
John-Jules Ch. Meyer

UU-CS-1999-17

Open Multi-Agent Systems: Agent Communication and Integration

Rogier M. van Eijk, Frank S. de Boer,
Wiebe van der Hoek and 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 open-ended nature of multi-agent systems, which refers to the property to allow for the dynamic integration of new agents into an existing system. In particular, the focus of this study is on the issues of agent communication and integration. We define an abstract programming language for open multi-agent systems that is based on concepts and mechanisms as introduced and studied in concurrency theory. Moreover, an important ingredient is the generalisation of the traditional concept of value-passing to a communication mechanism that allows for the exchange of information. Additionally, an operational model for the language is given in terms of a transition system, which allows the formal derivation of computations.

1 Introduction

In the research on multi-agent systems there is an increasing emphasis on the *open-ended* nature of agent systems, which refers to the feature to allow for the dynamic integration of new agents into an existing agent system. In such systems, which are referred to as *open multi-agent systems* (cf. [18]), it is usually impossible that agents possess complete built-in information about the other agents in the system, simply because such information will initially be unavailable. As was already pointed out by Hewitt and de Jong (cf. [13]) the only thing that holds the components of an open system in common, is their ability to communicate. This means that an important ingredient of an open multi-agent system will be the agents' ability to communicate about each other, especially about features like their capabilities and their expertise.

In the subfield of agent research that focuses on agent architectures, various types of agents have been proposed that facilitate the communication process in a multi-agent system. These agents, referred to with terms like *facilitators*, *routers*, *mediators*, *brokers* and so on (cf. [8]) act as intermediaries between communicating agents by providing services like the *matchmaking* between information producers and consumers. This denotes the act of referring agents in need of some piece of information to agents that might be able to provide it. As mentioned before, such facilitating activities are indispensable in the context of open multi-agent systems as the presence of agents, especially of those that have recently joined the system, need not be known to all other agents in the system.

With respect to the integration of new agents into an existing multi-agent system, we distinguish two different situations. First, there is the integration of an agent that already exists outside the system, in which case we refer to the integration as an *agent introduction*. An introduction is typically performed by a facilitating agent. Secondly, in the situation a newly integrated agent constitutes a previously nonexistent entity, we refer to the integration as an *agent creation*, which is a similar notion to that of an *object creation* from the object-oriented programming paradigm. An example of agent creation is the act of *agent cloning*, which is typically performed in situations in which an agent with limited resources, faces an overload of tasks that need to be accomplished (cf. [22]). To obviate this overload, the agent might then produce a clone of itself and subsequently delegate several of its tasks to this newly created agent.

Overview

Although over the last few years various new multi-agent programming languages have been developed, still few of these approaches are completely understood from a semantic point of view. This is mainly due to the fact that the concurrency aspects of these languages lack a clear modular structure, with the result that the interactions between the various agent features are rather difficult to get to grips with. This is why we advocate a modular and incremental approach to the design of multi-agent programming languages. In this respect, a fruitful starting point for the design of multi-agent languages lies in the traditional concepts and mechanisms as introduced and studied in the field of concurrency theory.

In this paper, we build upon the framework developed in [6, 5] by taking into account the open-ended nature of multi-agent systems, which amounts to the property to allow for the integration of new agents into an existing system. As in such open multi-agent system the communication structure is highly dynamic, we consider the concept of a *dynamically evolving communication structure* as for instance employed in the π-calculus (cf. [17]), the concurrent object-oriented language POOL (cf. [1]) and more specifically, its restricted version described in [2] that primarily focuses on the issues of object creation and dynamically evolving communication structures. In Section 2, we discuss the traditional communication mechanism of value-passing from concurrency theory. The generalisation of this communication mechanism to one that allows for the exchange of information between agents, is described in Section 3. Subsequently, in Section 4, we outline an abstract programming language for open multi-agent systems that concentrates on the exchange of information between agents and the integration of new agents. We define the syntax of the language, followed by the operational semantics by means of a transition system. Finally, in Section 5 we wrap up with suggesting several issues for future research.

2 Value-Passing

In classical concurrency theory, bilateral communication between processes proceeds via a mechanism of *value-passing*, which comprises the dispatch of a value by one process and the storage of this value by a receiving process. This is the communication mechanism as used in the well-known concurrent programming paradigm CSP (cf. [15]), in which such values are communicated along communication channels that interconnect the different processes. In particular, this mechanism covers a primitive of the form $\mathsf{tell}(c, e)$ to send the value of the expression e, which is evaluated in the local state of the sending process, along the communication channel c. The other communication primitive is of the form $\mathsf{ask}(c, x)$, which denotes the act of receiving a particular value along the channel c and the subsequent assignment of this value to the variable x.

One of the characteristics of this basic communication mechanism is that it assumes a static network of communication channels. This constitutes a severe drawback for open systems, as the dynamics of the population in these systems give rise to a constantly changing communication structure. This problem is dealt with in paradigms like the π-calculus (cf. [17]) in which communication channels *themselves* constitute values that can be passed among processes. In these settings, the above described communication primitives are generalised to the forms $\mathsf{tell}(y, e)$ and $\mathsf{ask}(y, x)$ also denoting the dispatch and reception of a value, respectively. They differ from the previous ones in that the communication channel along which is communicated is specified by a variable; the value of this variable y constitutes the actual channel along which is communicated.

The communication mechanism in the restricted version of the concurrent programming language POOL (cf. [2]) is similar except that processes indicate to which *process* a value is to be dispatched to or received from, instead of specifying a particular communication channel. In this setting, the communication primitives are also of the form $\mathsf{tell}(y, e)$ and $\mathsf{ask}(y, x)$, but the value of the variable y now constitutes the *identity* of a process the data are to be dispatched to or received from.

In the subsequent section we consider this language in more detail, as it constitutes a fruitful starting point for an open multi-agent framework. We remark that the generalisation of the other object-oriented aspects of the language POOL, like its rendezvous communication mechanism based on method invocations, are studied in [7].

Language With Value-Passing

The (restricted version of the) language POOL (cf. [2]), is used to program a collection of concurrently operating objects, which, in performing their computations, invoke the assistance of newly created objects. Interactions between the objects take place along a dynamically evolving communication structure, which is constructed by the objects themselves via the maintenance and communication of object identities. Such identities, which are stored in programming variables, constitute the means to address objects in the system. That is, only the fact that one of the variables of an object refers to the identity of an other object implies that it can communicate to the referred object.

A program in the language is given by a tuple of class definitions (including a special class called the *root class*) of the form $C \leftarrow S$, where C constitutes the name of the class and S is a programming statement that every object instance of the class will execute. Statements are given by a collection of basic actions, which can be composed using sequential composition, iteration and if-then-else constructs. A basic action is either an assignment $x := e$ in which the value of the expression e is assigned to the variable x, an action $x := \mathsf{new}(C)$ to create an instance of the class C having the side-effect that the identity of the created object is stored in the variable x of the creator, a primitive $\mathsf{ask}(y)$ to receive a value and subsequently assign this value to the variable y, a primitive $\mathsf{ask}(x, y)$ to receive a value from the object which identity is stored in x and store this value in the variable y or finally, an action $\mathsf{tell}(x, e)$ to send the value of the expression e, which is evaluated in the sender's local state, to the object which identity is stored in the variable x. The execution of a program starts with the creation of an instance of the root class, which is identified to be the *root object*. This root object executes the programming statement defined in its class during which it can create new objects of the other classes defined in the program. In this manner, a dynamic environment of concurrently operating objects is obtained, in which objects create other objects and secondly, pass the identities of the newly created objects among each other.

3 Exchange of Information

In traditional concurrent programming languages as for instance CSP and POOL, data is stored in programming variables. In particular, each process in these paradigms maintains a local state that maps variables to their corresponding values. The introduction of the concept of an *intelligent agent* (cf. [23]) as an entity that is assigned a mental state comprised of an informational as well as a motivational attitude (for instance a Belief, Desire and Intention (BDI) agent cf. [20]), brings about a novel view on the representation and manipulation of data in concurrent programming. Rather than a local state mapping variables to values, the *informational attitude* of an agent is effected by more elaborate information stores as belief and knowledge bases, which are usually represented in an expressive formalism as for instance a propositional, first-order or modal language. In this paper, we will employ the name *belief state* as a general term for such type of information stores. The shift from computing with local states to working with belief states has two important consequences on concurrent programming. First, the notion of a variable assignment as in the traditional languages is to be replaced by an operator that performs a form of *information update* and secondly, the place of a value-passing mechanism is to be filled by a mechanism that allows for the *exchange of information* between agents such as the communication of propositional, first-order or modal formulae.

In this paper, we will not say much about the other mental attitude of agents, called the *motivational attitude*, which for example covers attitudes like desires and intentions from the BDI-architectures. This attitude, which drives the agents' goal-directed behaviour, is discussed in more detail in [14]. From this paper, we learn the need for two additional stores: a base which stores goals that need to be satisfied as well as a base of plans which describe in what way these goals can be achieved.

3.1 Addressing Agents

An important component of communication in a multi-agent system concerns the way agents address each other. There are several aspects that play a role here, among which are the agents' identities, names and addresses.

The *identity* of an agent is that aspect that lays down the agent in a unique way. Usually, such an identity is accomplished by means of a unique *identifier* that distinguishes the agent from all other agents.

The second aspect is the *name* of an agent, which is used by other agents to refer to the agent. The relation between a name and an identity is that the former can be used to denote the latter. The most significant difference between the two aspects is that it is impossible that two distinct identities are equal, while it is usually not excluded that two distinct names denote the same identity. A name is *absolute* if it is shared by all agents in the system, while it is called *relative* if it is local to an agent. One of the characteristics of a relative name is that if it is employed by different agents it is likely to have a different denotation.

Thirdly, in order to be able to communicate to an agent one is required to be in possession of this agent's *address*. In general, such an address is given by some physical location messages can be sent to. A striking difference between addresses on the one hand and identities and names on the other, is that addresses do not need to be static. For instance, the address of mobile agents changes as soon as these agents migrate from one site to another, while their name and identity remain the same.

In more abstract settings, agents can address each other by specifying communication channels or agent names. That is, rather than giving the exact location, agents specify a *communication channel* messages are communicated along like for instance in the classical concurrent programming framework CSP or specify the *name* of the agent they communicate with like in the object-oriented language POOL. The task of mapping communication channels and agent names to actual physical locations is then dealt with in the underlying operational system. In our framework, we adopt the latter mechanism of addressing agents by a name (in the form of a variable).

3.2 Receiving Information

In our view, one of the striking differences between classic concurrent programming on the one hand and multi-agent programming on the other hand, is the shift from a value-passing communication mechanism to one that allows for the exchange of information. To illustrate this, let us consider the following typical multi-agent scenario.

Example 1 Consider an agent client that is in search for an agent that can answer a particular question ψ concerning a subject s. The following subsequent events happen. The agent client asks a facilitating agent $\mathit{server1}$ for an agent that is an expert on the subject s. The agent $\mathit{server1}$ examines its belief state, finds out that the agent $\mathit{expert1}$ is such an agent, and answers client with this information. Next, client consults another facilitator $\mathit{server2}$ with the question whether $\mathit{expert1}$ is considered to be a reliable agent. As it subsequently appears that this is not the case, client returns to $\mathit{server1}$ and asks for another agent that is an expert on s. Subsequently, $\mathit{server1}$ replies with the information that $\mathit{expert2}$ is such an appropriate agent. After $\mathit{server2}$ has confirmed client that $\mathit{expert2}$ is a reliable agent, client turns to $\mathit{expert2}$ and asks this agent the question ψ.

The most important point that should become clear from this example is that agents are not so interested in plain values (agent identities, in this case), but rather in values in connection with their properties: for instance, an identity denoting an expert agent on some subject, an identity denoting a reliable agent, an identity that is unequal to another identity, and so on. This originates from the fact that agents compute with belief states consisting of formulae that express properties about variables rather than with local states that simply map variables to their associated values. In fact, this resembles the computation mechanism as used in *constraint-based* programming languages, in which processes compute with information that constrains the range of values that variables can take. Such pieces of information, called *constraints*, are expressed in a first-order language and collected in a *constraint store*, like in Concurrent Constraint Programming (CCP) (cf. [21]). For instance, whereas in standard programming a local state assigns a variable x a value a, a constraint store contains a constraint of the form $x = a$.

The generalisation of traditional value-passing to a mechanism of exchanging information proceeds as follows. The value-passing primitive $\mathsf{ask}(y, x)$ for the reception of values can be generalised to a primitive of the form $\mathsf{ask}(y, \psi)$, where ψ is a formula (expressing some property about some variables).

Let us informally describe the semantics of this primitive. First of all, the belief state of the agent that performs the action should imply what particular agent is denoted by y. That is, it should derive a formula of the form $y = i$, for some agent identifier i. The primitive is then employed to ask the agent i whether the formula ψ holds. Communication subsequently takes place if this agent i, in turn, provides an answer φ that entails the posed question ψ. For instance, consider the predicate $Exp(x, y)$ that denotes that x is an expert on y and the predicate $Rel(x)$ that expresses that x is reliable. If the question ψ is given by $Exp(\mathit{agent1}, \mathit{subject1})$ then the formula $Exp(\mathit{agent1}, \mathit{subject1}) \wedge \neg Exp(\mathit{agent1}, \mathit{subject2})$ provided by i would constitute an appropriate answer for this question.

The communication mechanism becomes more interesting if we allow free variables in questions; the idea being that φ is an answer for ψ if it entails ψ modulo a substitution of the free variables in ψ. For instance, the formula $Exp(\mathit{agent1}, \mathit{subject1}) \wedge \neg Exp(\mathit{agent1}, \mathit{subject2})$ is an answer for the question $Exp(x, \mathit{subject1})$, as it entails the question provided that we substitute $\mathit{agent1}$ for x.

Moreover, in the subsequent communication steps we would like the agents to be able to refer to elements in the domain of discourse that were mentioned in previous steps. For instance, in the above example, we would like the agent to be able to ask a subsequent question concerning x, like the question $Rel(x)$ to ask whether this particular agent x (i.e. $\mathit{agent1}$) is a reliable agent. To achieve this, we proceed as follows: we employ two special symbols ? and ! to distinguish the two different situations; i.e. the case in which a new variable is being introduced and the case a variable corresponds to one that has been introduced in an earlier communication round. The above mentioned consecutive questions would then look like $?xExp(x, \mathit{subject1})$ and $!xRel(x)$. The former denotes the question that can be described as '*which* agent x is an expert on $\mathit{subject1}$', while the latter can be described as 'is *this* particular agent x reliable'.

Finally, analogous to the language POOL, we introduce a variant $\mathsf{ask}(\varphi)$ of the primitive $\mathsf{ask}(x, \varphi)$ to ask the question φ to an *arbitrary* agent.

Summarising, the operators ? and ! give rise to a mechanism of using variables outside the scope of the formula that they have been introduced in. This feature enables a flexible communication mechanism, as in each communication step agents are able to refer to elements in the domain of discourse that have been mentioned in previous communication rounds. In fact, this mechanism has close connections with Discourse Representation Theory (DRT) studied in the field of natural language semantics, which has been developed for the systematic construction of discourse representations (cf. [16, 10]). In this light, variables annotated by the operators $?$ or $!$ can be seen as *discourse markers* that are used in DRT to keep track of the individuals that are under discussion.

Moreover, in the terminology of natural language research on questions, a formula of the form $?x\varphi$ constitutes a *mention-one interrogative*, which amounts to a question to mention one particular entity that satisfies the formula φ, as opposed to *mention-some* and *mention-all* interrogatives that are used to request for some instances or for an exhaustive description, respectively (see [11] for further details).

Finally, another interpretation of the operators $?$ and $!$ is that they denote *input* and *output modes* that indicate the stream of information, for instance like in the concurrent logic programming language Parlog (cf. [3]). That is, a variable $!x$ bound by an output mode (i.e. an *output variable*) denotes a variable for which a value has already been established, and which is supplied by the agent itself, while a variable $?x$ bound by an input mode (i.e. an *input variable*) denotes a new variable, which value is to be supplied by the environment (e.g. the answering agent).

3.3 Sending Information

Next, we consider the act of sending information. In the value-passing communication mechanism as used in the π-calculus and the language POOL, there is a primitive of the form $\mathsf{tell}(y, e)$ to send the value of the expression e. A straightforward generalisation of this primitive is one of the form $\mathsf{tell}(y, \varphi)$ denoting the dispatch of the formula φ to y, where we require that φ follows from the agent's belief state. This requirement results from our focus on sincere agents, i.e. agents that communicate information they themselves believe to be true. Secondly, as φ will constitute an answer to a question posed by another agent, it is required that it does not contain any input variables.

3.4 Synchronous communication

Communication between two agents i and j then comprises an action $\mathsf{tell}(y, \varphi)$ performed by i, where its belief state yields the information that y denotes the agent j and an action $\mathsf{ask}(z, \psi)$ executed by the agent j, where its belief state yields that z denotes the agent i, such that the answer φ entails the question ψ modulo a substitution of the input variables in ψ. We remark that the execution of the send and the receive action constitutes an *atomic* activity, which means that it is not interrupted by other actions. This enables us to concentrate on the basic communication mechanism, without the complications of buffering and labeling messages. Additionally, the choice in favour of synchronous communication does not rule out the asynchronous variant: as is usual in concurrency theory, asynchronous communication can be mimicked by synchronous communication (by using intermediaries that operate as message buffers).

4 Programming Framework

4.1 Preliminaries

In this section, we shape an abstract programming framework that incorporates the communication mechanism as sketched in the previous section. First, we give the standard notions of a signature, term and formula.

Definition 2 *(Signature)*
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 0-ary function symbols are called *constants*.

Additionally, we assume a set Ident of constants to denote agent identifiers with typical elements i, j, k and a set Var of variables, with typical elements u, v, w, x and so on.

Definition 3 *(Terms)*
The set Ter of terms over $\mathcal{L}$ is inductively defined by: $\mathit{Var} \subseteq \mathit{Ter}$ and secondly, if $t_1, \ldots, t_k \in \mathit{Ter}$ and $F \in \mathcal{F}$ of arity k then $F(t_1, \ldots, t_k) \in \mathit{Ter}$. A term is *closed* if it contains no variables from Var.

Definition 4 *(Formulae)* Consider the following clauses:

(1) if $t_1, t_2, \ldots, t_k \in \mathit{Ter}$ 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 $x \in \mathit{Var}$ then $\neg\varphi,\ \varphi \wedge \psi,\ \exists x\varphi \in S$
(3) if $x \in \mathit{Var}$ and $\varphi \in S$ then $!x\varphi \in S$
(4) if $x \in \mathit{Var}$ and $\varphi \in S$ then $?x\varphi \in S$.

The sets For, For_o , For_i and For_{io} are defined to be the smallest sets S that satisfy the clauses (1,2), (1,2,3), (1,2,4) and (1,2,3,4), respectively.

The set For denotes the standard collection of first-order formulae. Formulae in the set For_o additionally may contain output variables, while formulae in For_i may contain input variables. Finally, the set For_{io} collects first-order formulae that can contain input as well as output variables. The logical operators $\vee$, $\rightarrow$ $\leftrightarrow$ and $\forall$ can be defined by means of the operators $\neg$, $\wedge$ and $\exists$, in the usual way.

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 $\{\varphi_1, \ldots, \varphi_n\}$ of formulae represent the formula $\varphi_1 \wedge \cdots \wedge \varphi_n$, or a formula φ represent the set $\{\varphi\}$.

We assume that there is a common ontology that is shared by all agents in a multi-agent system (cf. [12]). That is, we suppose that there exists a common signature $\mathcal{L}$ together with a common entailment relation $\vdash$ on $\mathcal{P}(\mathit{For}) \times \mathcal{P}(\mathit{For})$.

In order to deal with the exchange of information that may contain input and output variables, we need the notion of a ground substitution.

Definition 5 *(Ground substitutions)*
A ground substitution Δ is modeled as a set of formulae of the form $x = t$, where $x \in \mathit{Var}$ and t is a closed term in Ter. We require that Δ binds each variable to at most one term.

To be able to apply a substitution to a formula we need the following notions.

Definition 6 *(Completeness of substitutions)*
A substitution Γ is *o-complete (i-complete)* for a formula $\varphi \in For_{io}$ if for each output variable $!x$ (input variable $?x$) in φ there exists $t \in Ter$ such that $(x = t) \in \Gamma$.

Next, we consider the application of substitutions to formulae: we define an operation $\oplus$ that substitutes the output variables in a particular formula and an operation $\otimes$ that substitutes the input variables. That is, $\psi \oplus \Gamma$ denotes the formula ψ in which the output variables are substituted by the value given by Γ, while $\psi \otimes \Gamma$ denotes this formula in which the input variables have been substituted.

Definition 7 *(Functions $\oplus$ and $\otimes$)*
For each $\psi \in For_{io}$ and substitution Γ that is o-complete for ψ, we define the function $\oplus$ by induction on the structure of ψ:

– $P(t_1, \ldots, t_k) \oplus \Gamma = P(t_1, \ldots, t_k)$
– $(\neg\varphi) \oplus \Gamma = \neg(\varphi \oplus \Gamma)$
– $(\varphi \wedge \psi) \oplus \Gamma = (\varphi \oplus \Gamma) \wedge (\psi \oplus \Gamma)$
– $(?x\varphi) \oplus \Gamma = ?x(\varphi \oplus \Gamma)$
– $(!x\varphi) \oplus \Gamma = \exists x(x = t \wedge (\varphi \oplus \Gamma))$, where $(x = t) \in \Gamma$
– $(\exists x\varphi) \oplus \Gamma = \exists x(\varphi \oplus \Gamma)$

Additionally, for each i-complete substitution Γ for ψ, the definition of the function $\otimes$ is similar except for:

– $(?x\varphi) \otimes \Gamma = \exists x(x = t \wedge (\varphi \otimes \Gamma))$, where $(x = t) \in \Gamma$
– $(!x\varphi) \otimes \Gamma = !x(\varphi \otimes \Gamma)$

The idea behind this definition is that the substitution of a term t for a variable x in a formula φ, is logically equivalent to the formula $\exists x(x = t \wedge \varphi)$.

Finally, the following definition formalises the communication mechanism of asking a question φ and telling an answer ψ.

Definition 8 *(Minimal unifiers)*

– A substitution Δ is called a *unifier* for the pair (φ, ψ) where $\varphi \in For$ and $\psi \in For_i$, if $\varphi \vdash (\psi \otimes \Delta)$.
– Additionally, Δ is a *minimal unifier* if there is no other unifier Δ' with $\Delta \vdash \Delta'$.

We use the notation $\varphi \vdash_\Delta \psi$ to denote that Δ is a minimal unifier for (φ, ψ).

A minimal unifier Δ supplies the values for the input variables in ψ such that the formula φ constitutes an answer for the question ψ. For instance, $Exp(agent1, subject1)$ is an answer for $?xExp(x, subject1)$ under the minimal unifier $x = agent1$.

Note that minimal unifiers are not necessarily unique: for instance, $x = agent1$ and $x = agent2$ are both minimal unifiers for:

$$(Exp(agent1, subject1) \wedge Exp(agent2, subject1), ?xExp(x, subject1)).$$

The non-determinism inherited from this communication mechanism can however be controlled by the programmer: it can choose to send either $Exp(agent1, subject1)$ or $Exp(agent2, subject1)$ instead of their conjunction.

4.2 Syntax

We shape a programming framework for open multi-agent systems in which new agents can be integrated and the communication mechanism allows for the exchange of information. First, we give its syntax.

Definition 9 *(Syntax of the programming language)* Let $\varphi \in \mathit{For}$, $B \subseteq \mathit{For}$, $\psi \in \mathit{For}_{io}$, $\chi \in \mathit{For}_i$, $x \in \mathit{Var}$, $i \in \mathit{Ident}$ and P a procedure in W, then atomic actions a, programming statements S, agents A and agent systems $\mathcal{A}$ are defined as follows:

$$\begin{array}{l} a ::= \mathsf{update}(\varphi) \ \mid \ \mathsf{tell}(x,\psi) \ \mid \ \mathsf{ask}(x,\chi) \ \mid \ \mathsf{ask}(\chi) \ \mid \ \mathsf{integrate}(x,S,B) \\ S ::= a \cdot S \ \mid \ S_1 \,\&\, S_2 \ \mid \ S_1 + S_2 \ \mid \ P \ \mid \ E \\ A ::= \langle i, S, B \rangle \\ \mathcal{A} ::= A \ \mid \ A, \mathcal{A} \end{array}$$

where W is a set of procedure declarations, which are of the form $P : -S$, where P is the name of the procedure and S its body statement.

An agent A is assigned a unique identifier i from the set Ident, which is used to distinguish it from all other agents in the system. Additionally, its behaviour is controlled by a program S. The third constituent of an agent is a belief state B, which we assume is represented by a set of formulae.

An atomic action a is either the operation $\mathsf{update}(\varphi)$ to update the belief state with the formula φ, the action $\mathsf{tell}(x,\psi)$ to send the agent x the formula ψ, the action $\mathsf{ask}(x,\chi)$ to ask the agent x for the formula χ, the action $\mathsf{ask}(\chi)$ to ask χ to an arbitrary agent and finally the action $\mathsf{integrate}(x,S,B)$ to integrate a new agent in the system that will execute the program S with the initial belief state B. The variable x will be used by the creator to denote this new agent.

These primitive actions can be composed to form programms by means of the familiar constructs $\cdot$ for action prefixing, $\&$ for parallel composition, $+$ for non-deterministic choice and procedure calls of the form P. The symbol E denotes the empty (terminated) statement. Usually, we will write statements of the form $S \cdot E$ simply as S.

Finally, an *agent system* $\mathcal{A}$ is a set of agents.

4.3 Transition Systems

We develop the semantics of the programming language by means of a transition system, which constitutes an elegant mechanism of describing operational behaviour and dates back to the original work of Plotkin on semantics of programming languages (cf. [19]). Formally, a *transition* is of the form $\mathcal{A} \xrightarrow{l} \mathcal{A}'$. It denotes a computation step of the agent system $\mathcal{A}$ where $\mathcal{A}'$ is the resulting agent system and the label l either expresses that the transition needs to synchronise with another transition, or states that this is not the case (indicated by the symbol τ).

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

$$\frac{\mathcal{A}_1 \xrightarrow{l_1} \mathcal{A}_1' \ \cdots \ \mathcal{A}_n \xrightarrow{l_n} \mathcal{A}_n'}{\mathcal{A} \xrightarrow{l} \mathcal{A}'} \qquad \text{if } \mathit{cond}$$

Such a rule denotes that the transition below the line can be derived if the transitions above the line are derivable and additionally, the condition *cond* holds. 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*.

4.4 The Transition Rules

In this section, we develop a transition system for the multi-agent programming language, starting with the atomic ations. We abstract from a particular belief revision strategy (cf. [9]); that is, the framework is parameterised by an appropriate belief revision operator $\circ : (\mathcal{P}(For) \times \mathcal{P}(For)) \rightarrow \mathcal{P}(For)$ for which we for instance assume that $(B \circ \varphi) \vdash \varphi$ holds. The transition for an update of the belief state is then as follows.

Definition 10 *(Transition for belief revision)*

$$\langle i, \mathsf{update}(\varphi), B\rangle \xrightarrow{\tau} \langle i, E, B \circ \varphi\}\rangle$$

Next, we consider the integration of new agents. The action $\mathsf{integrate}(x, S, B')$ extends the current agent system with a new agent j that starts to execute the statement S, where its belief state is given by B'. The variable x will be used by the integrating agent to refer to the integrated agent.

Definition 11 *(Transition for agent integration)*

$$\langle i, \mathsf{integrate}(x, S, B'), B\rangle \xrightarrow{\tau} \langle i, E, B \circ (x = j)\rangle, \langle j, S, B'\rangle$$

where j is a fresh agent identifier from $Ident$.

Next, we define the transitions for the actions of sending and receiving information, which model the local effects of a synchronous communication step. We use two types of labels: a label of the form (i, j, φ), which denotes that i tells to j the formula φ and a label of the form $(i, j, \varphi), \Delta$, which denotes that i asks j the formula φ, where Δ denotes a possible substitution for the input variables in φ.

Definition 12 *(Transitions for sending and asking)*
Provided that there exists a substitution Γ with $B \vdash \Gamma$ that is o-complete for φ, and a substitution Δ that is i-complete for $\varphi \oplus \Gamma$, we have the following transitions:

$$\begin{array}{lll}
\langle i, \mathsf{tell}(x, \varphi), B\rangle \xrightarrow{(i, j, \varphi \oplus \Gamma)} \langle i, E, B\rangle & \text{if } B \vdash \varphi \wedge (x = j) \\
\langle i, \mathsf{ask}(x, \varphi), B\rangle \xrightarrow{(i, j, \varphi \oplus \Gamma), \Delta} \langle i, E, B \circ \Delta\rangle & \text{if } B \vdash (x = j) \\
\langle i, \mathsf{ask}(\varphi), B\rangle \xrightarrow{(i, j, \varphi \oplus \Gamma), \Delta} \langle i, E, B \circ \Delta\rangle &
\end{array}$$

The values for the output variables in the formula φ come from the agents belief state B and are collected in the substitution Γ. The formula $\varphi \oplus \Gamma$ then constitutes the formula φ in which all these output variables are substituted by the values given by Γ.

In the transition for telling, the condition $B \vdash \varphi$ denotes the property of sincerity: any dispatched formula is to be believed true by the sender. Note that in the transitions for asking, the set Δ of values for the input variables of φ is guessed, that is, it denotes a *possible* set of values. Only in the transition rule for synchronous communication a *real* set Δ is established; that is, one that is based on the information provided by the sender. Note that this set Δ is stored in the receiver's belief state for later use; i.e. to provide the substitutions for the variables that are bound by the operator ! in subsequent questions and answers.

Definition 13 *(Transition for synchronous communication)*

$$\frac{A_1 \xrightarrow{(i,j,\varphi)} A_1' \qquad A_2 \xrightarrow{(j,i,\psi),\,\Delta} A_2'}{\mathcal{A}, A_1, A_2 \xrightarrow{\tau} \mathcal{A}, A_1', A_2'} \quad \text{if } \varphi \vdash_\Delta \psi$$

Communication between the agents A_1 and A_2 takes place if the information φ provided by A_1 constitutes an answer for the question ψ posed A_2 modulo the substitution Δ of the input variables in ψ. (Of course, the agents A_1 and A_2 are required to be distinct and not to occur in $\mathcal{A}$.)

The remaining transitions are quite standard.

Definition 14 *(Transition for sequential composition)*

$$\frac{\langle i, a, B\rangle \xrightarrow{l} \langle i, E, B'\rangle}{\langle i, a \cdot S, B\rangle \xrightarrow{l} \langle i, S, B'\rangle}$$

The transition for the sequential composition $a \cdot S$ is given by the transition for the action a. The statement S constitutes the part of this statement that needs to be executed next.

Definition 15 *(Transition for non-deterministic choice and internal parallelism)*

$$\frac{\langle i, S_1, B\rangle \xrightarrow{l} \langle i, S_1', B'\rangle}{\begin{array}{c}\langle i, S_1 + S_2, B\rangle \xrightarrow{l} \langle i, S_1', B'\rangle \\ \langle i, S_2 + S_1, B\rangle \xrightarrow{l} \langle i, S_1', B'\rangle\end{array}} \qquad \frac{\langle i, S_1, B\rangle \xrightarrow{l} \langle i, S_1', B'\rangle}{\begin{array}{c}\langle i, S_1 \,\&\, S_2, B\rangle \xrightarrow{l} \langle i, S_1' \,\&\, S_2, B'\rangle \\ \langle i, S_2 \,\&\, S_1, B\rangle \xrightarrow{l} \langle i, S_2 \,\&\, S_1', B'\rangle\end{array}}$$

The transitions for a non-deterministic choice between two statements are given by the transitions of exactly one of them, while the transitions for a parallel composition are given by an interleaving of the transitions of both of the statements.

Definition 16 *(Transition for a procedure call)*
If $P : -S$ is a procedure declaration in W then we have the following transition rule:

$$\frac{\langle i, S, B\rangle \xrightarrow{l} \langle i, S', B'\rangle}{\langle i, P, B\rangle \xrightarrow{l} \langle i, S', B'\rangle}$$

The transition for a procedure call P is derived from the transition of its body S.

Analogous to internal parallelism, external parallelism is modeled by means of an interleaving semantics. Models for true concurrency, which are perhaps even more natural in the context of agent systems, are among the subjects of future research.

Definition 17 *(Transitions for external parallelism)*
Provided that A_1 and A_2 do not occur in $\mathcal{A}$ we have the transitions:

$$\frac{A_1 \xrightarrow{\tau} A_1'}{\mathcal{A}, A_1 \xrightarrow{\tau} \mathcal{A}, A_1'} \qquad \frac{A_1 \xrightarrow{\tau} A_1', A_2}{\mathcal{A}, A_1 \xrightarrow{\tau} \mathcal{A}, A_1', A_2}$$

The second rule allows for the integration of a new agent A_2 in the system.

4.5 Example

Let us return to the agent system as described in Example 1. Suppose the configuration of the agent *client* is given by $A_1 = \langle 1, S_1, B_1 \rangle$, where the program S_1 is given by:

$$S_1 = \mathsf{ask}(u, ?x Exp(x, s)) \cdot (T_1 + U_1)$$

$$T_1 = \mathsf{ask}(v, !x Rel(x)) \cdot \mathsf{ask}(x, \psi)$$

$$\begin{aligned} U_1 = {} & \mathsf{ask}(v, !x \neg Rel(x)) \cdot \mathsf{ask}(u, ?z!x(Exp(z, s) \wedge \neg(z = x))) \cdot \\ & \mathsf{ask}(v, !z Rel(z)) \cdot \mathsf{ask}(z, \psi) \end{aligned}$$

and the belief state B_1 equals $(u = 2) \wedge (v = 3)$. In this program, the agent u is asked for an expert x on s. If subsequently, according to agent v, this expert x appears to be reliable then x is asked the question ψ. However, if x turns out to be unreliable then u is asked for another expert z on s (which is unequal to x). In case the agent v can subsequently confirm that z is reliable then z is asked the question ψ.

Additionally, the configurations of *server1* and *server2* equal $A_2 = \langle 2, P, B_2 \rangle$ and $A_3 = \langle 3, Q, B_3 \rangle$, respectively, where B_2 equals $(x = 1) \wedge Exp(4, s) \wedge Exp(5, s)$ and B_3 equals $(x = 1) \wedge \neg Rel(4) \wedge Rel(5)$, while P and Q are procedures declared as:

$$P :- \ (\mathsf{tell}(x, Exp(4, s) \cdot P)) + (\mathsf{tell}(x, Exp(5, s)) \cdot P)$$

$$Q :- \ (\mathsf{tell}(x, \neg Rel(4)) \cdot Q) + (\mathsf{tell}(x, Rel(5)) \cdot Q)$$

Finally, we assume that some configurations A_4 and A_5 for the agents *expert1* and *expert2* are given. As an example of the transition system, we show the derivation of the first transition of this multi-agent system A_1, A_2, A_3, A_4, A_5.

1. The following transition is an axiom, where φ equals $?x Exp(x, s)$
   $\langle 1, \mathsf{ask}(u, \varphi), B_1 \rangle \xrightarrow{(1, 2, \varphi), x = 4} \langle 1, E, B_1 \circ (x = 4) \rangle$
2. From 1. and the rule for sequential composition:
   $\langle 1, S_1, B_1 \rangle \xrightarrow{(1, 2, \varphi), x = 4} \langle 1, T_1 + U_1, B_1 \circ (x = 4) \rangle$
3. The following is an axiom:
   $\langle 2, \mathsf{tell}(x, Exp(4, s)), B_2 \rangle \xrightarrow{(2, 1, Exp(4, s))} \langle 2, E, B_2 \rangle$
4. Via 3. and the rule for sequential composition:
   $\langle 2, \mathsf{tell}(x, Exp(4, s)) \cdot P, B_2 \rangle \xrightarrow{(2, 1, Exp(4, s))} \langle 2, P, B_2 \rangle$
5. Via 4. and the rule for non-deterministic choice:
   $\langle 2, (\mathsf{tell}(x, Exp(4, s)) \cdot P) + (\mathsf{tell}(x, Exp(5, s)) \cdot P), B_2 \rangle \xrightarrow{(2, 1, Exp(4, s))} \langle 2, P, B_2 \rangle$

6. From 5. and the rule for procedure calls:
$\langle 2, P, B_2\rangle \xrightarrow{(2,1,Exp(4,s))} \langle 2, P, B_2\rangle$
7. The rule for communication applied to 2. and 6. yields:
$\langle 1, S_1, B_1\rangle, A_2, A_3, A_4, A_5 \xrightarrow{\tau} \langle 1, T_1 + U_1, B_1 \circ (x = 4)\rangle, A_2, A_3, A_4, A_5$
as $(x = 4)$ is a minimal unifier for $(Exp(4, s), ?x Exp(x, s))$.

5 Conclusions and Future Work

In this paper, we have outlined an abstract programming language for open multi-agent systems that concentrates on the exchange of information and the integration of new agents. The language is given a clear operational model in terms of a transition system that is used for the formal derivation of the computation steps of a multi-agent system. Moreover, the configurations of the transition system can be viewed upon as an abstract model of a machine, while the transitions specify the subsequent actions this machine can perform. In this way, the machine would act as an interpreter for the language.

In subsequent research, we aim to mould the given operational semantics for the language to one that is *compositional*. This property, which means that the semantics of a composed statement or agent system can be derived from the semantics of its components, constitutes the next step towards top down design facilities for open multi-agent systems as well as to the development of specification and verification techniques.

Secondly, we aim to refine the framework to one in which each agent has its own individual signature. This will give rise to a natural hierarchy among the agents in terms of their signature, which constitutes a starting-point for the introduction of such object-oriented features as sub-typing and inheritance (cf. [4]).

References

1. 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.
2. P.H.M. America and F.S. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6:269–316, 1994.
3. K. Clark and S. Gregory. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.
4. 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.
5. 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.
6. 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.

7. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Operational semantics for agent communication languages. Technical report UU-CS-1999-08, Universiteit Utrecht, Department of Computer Science, 1999.
8. 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.
9. P. Gärdenfors. *Knowledge in flux: Modelling the dynamics of epistemic states*. Bradford books, MIT, Cambridge, 1988.
10. J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14(1):39–100, 1991.
11. J. Groenendijk and M. Stokhof. Questions. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 1055–1124. Elsevier/MIT Pess, Amsterdam/Cambridge, Mass., 1997.
12. T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*. Kluwer Academic Publishers, 1993.
13. C. Hewitt and P. de Jong. Analyzing the roles of descriptions and actions in open systems. In *Proceedings of 3rd National Conference on Artificial Intelligence (AAAI-83)*, pages 162–167, Washington, D.C., 1983.
14. 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.
15. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
16. H. Kamp. A theory of truth and semantic interpretation. In J. Groenendijk, T. Janssen, and M. Stokhof, editors, *Formal Methods in the Study of Language*, pages 277–322. Mathematical Centre, Amsterdam, 1981.
17. R. Milner, J. Parrow, and D. Walke. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
18. M.H. Nodine and A. Unruh. Facilitating open communication in agent systems: The infosleuth infrastructure. 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 281–295. Springer-Verlag, 1998.
19. G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
20. A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, Cambridge, Massachusettes, 1991.
21. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245, 1990.
22. O. Shehory, K. Sycara, P. Chalasani, and S. Jha. Agent cloning. In *Proceedings 3rd International Conference on Multi-Agent Systems (ICMAS'98)*, pages 463–464. IEEE Computer Society, 1998. Poster Abstract.
23. M. Wooldridge and N. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.