Semantics of Agent Communication: An Introduction

Rogier M. van Eijk

Institute of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands,
`rogier@cs.uu.nl`

Abstract. Communication has been one of the salient issues in the research on concurrent and distributed systems. This holds no less for the research on multi-agent systems. Over the last few years the study of agent communication, and in particular the semantics of agent communication, has attracted increased interest. The present paper provides an introduction to this area. Since agent communication builds upon concepts and techniques from concurrency theory, we start by giving a short historical overview that covers shared-variable concurrency, message-passing, rendezvous, concurrent constraint programming and agent communication. Standard approaches of agent communication identify three different layers: a content layer, message layer and communication layer. To this model we add an extra level, namely the layer of the multi-agent system. Subsequently, we discern three approaches in developing the semantics of programming languages: the axiomatic, operational and denotational approach. Additionally, we discuss semantic aspects of agent communication, including communication histories, compositionality, observable behaviour, failure sets and full abstractness. We illustrate these issues by means of the framework ACPL (Agent Communication Programming Language). Finally, we briefly consider the specification and verification of agent communication.

1 Introduction

The introduction of novel application areas has urged the development of new programming concepts and techniques to assist both the programmer and end-user in managing the inherent complexity of computer software. A concept that plays a prominent role in the research of the late 1990s and the beginning of the third millennium is that of an *agent*. This concept has found its shape in the field of artificial intelligence and builds upon notions from other disciplines of research as philosophy, economics, sociology and psychology. Although in artificial intelligence, there is no real consensus on what exactly constitutes an agent, there are some generally accepted properties attributed to it. In fact, this can also be said about a related notion from computer science, namely that of an *object*, which over the years, despite a lack of consensus on its definition, nonetheless has proven to be a successful concept for the design of a new generation of programming languages.

In short, an *agent* is an autonomous entity that shows both a reactive and pro-active behaviour by perceiving and acting in the environment it inhabits [47]. Moreover, it has a social ability to interact with other agents in multi-agent systems, like the capability to share knowledge through communication, to coordinate its activities with those of

other agents, to cooperate with other agents or to compete with them. In the stronger conception of agency, an agent is additionally assumed to have a mental state consisting of *informational* attitudes, like knowledge and belief, as well as *motivational* attitudes, like goals, desires and intentions. In other words, rather than being thought of as a computational entity in the traditional sense, an agent is viewed upon as a more elaborate software entity that embodies particular human-like characteristics. For instance, an issue in the rapidly growing research area of electronic commerce, is the study whether agents can assist humans in their tedious tasks of localising, negotiating and purchasing goods [32]. In general, negotiation activities comprise the exchange of information of a highly complex nature, requiring the involved parties to employ high-level modalities as knowledge and belief about the knowledge and belief of the other parties. Moreover, more elaborate negotiations also involve aspects of argumentation to explain the reasons why particular offers are proposed.

We could say that emerging novel application areas require the development of new programming paradigms, since the emphasis of programming involves a shift from the traditional performance of *computations* towards the employment of the more involved concepts of *interaction* and *communication*. In particular, in the new paradigms, the central focus is on computer programs that interact and communicate at a higher level of abstraction. That is, rather than a mere exchange of low-level data, communication between agents can for instance involve propositions that are believed to be true or false, actions that are requested to be performed and goals that are to be achieved.

Over the last few years the study of agent communication, and in particular the semantics of agent communication, has attracted increased interest. The present paper provides an introduction to the area. It is organised as follows. In Section 2, we give a short historic overview of the research on communication in concurrent programming languages that covers shared-variable concurrency, message-passing, rendezvous, concurrent constraint programming and agent communication. We discuss the standard model of agent communication, which consists of a content layer, message layer and communication layer. To this model we add one extra level, namely the layer of the multi-agent system. Subsequently, in Section 3, we discern three approaches for developing the semantics of agent communication: the axiomatic, operational and denotational approach. Finally, in Section 4, we discuss compositionality, observability, specification and verification of agent communication.

2   From Shared Variables to Agent Communication

In our conception, the study of agent communication can be thought of as a next step in a long history of research on concurrent programming languages. Over the years the area of concurrency theory has produced many concepts, mechanisms and techniques for a clear understanding of the concurrency and communication aspects of programming languages. We give an overview.

2.1   Shared Variable Concurrency

In the early days, research on programming languages was concerned with languages for *sequential* programming. Characteristic of a sequential program is that its computation

starts with the execution of the first action after which control moves along the subsequent actions of the program as dictated by the programming constructs. In other words, the execution of a sequential program is given by a single thread of control.

Later on, in the mid 1960s, people began to develop and study programming languages for computer systems in which execution is not a sequential process but where instead, different activities can occur concurrently. In particular, in a program of a *concurrent* programming language one cannot discern a single thread of control; there are multiple active processes each of which is governed by its individual thread. In comparison with sequential languages, programs in a concurrent programming language are far more complex. This is due to the fact that the concurrently operating processes of a program should be coordinated in such a way that they can cooperate with each other, but on the other hand their individual computations do not interfere.

Among the first programming languages for concurrent systems are the ones developed for *shared variable* concurrency [11]. According to this paradigm, a program is composed of a set of concurrent processes that communicate by means of a collection of shared variables. One of the key problems encountered for these languages is that it should be prohibited for different processes to have simultaneous access to the same variables. In other words, their activities need to be coordinated, for if they do have simultaneous access to a particular variable then unexpected behaviour can occur. This is illustrated in the following example.

Example 1. (*Simultaneous access*)
Suppose that the execution of an action $x := e$ consists of an evaluation of the expression e after which the computed value is assigned to the variable x. Consider a program that consists of two concurrently executing processes A and B, which are defined as follows:

$$A \equiv x := x + 1$$
$$B \equiv x := x + 2.$$

Process A increases the value of x by 1, while process B increases it with 2. If the initial value of the variable x is equal to 0, we would expect that its resulting value is 3. However, consider the following scenario. The process A computes the value of $x + 1$ and finds it to be equal to 1. Before it assigns this value to the variable x, the other process B executes the action $x := x + 2$. The evaluation of the expression $x + 2$ yields the value 2, which is subsequently assigned to the variable x. Meanwhile, the agent A finishes its execution by overwriting the current value 2 of x by the value 1. Thus, in this scenario, the final value of x is equal to 1 instead of the expected value 3.

In general, for each of the processes in a program for shared-variable concurrency, one can identify *critical sections* in which it is necessary that the process has exclusive access to particular shared data. Correspondingly, it should be ensured that only a restricted subset (typically, just *one* process) is executing its critical section at the same time. Many techniques have been developed for the synchronisation of processes that have shared data, among which the most prominent ones are *semaphores* [12] and *monitors* [27]. In real life, a semaphore allows only a restricted number of trains on a particular railroad track. In a computer, it allows only a restricted number of processes to be in their critical section. A monitor not only defines the procedures that can be invoked

to operate on a particular set of shared variables, but also coordinates the execution of these procedures. Thus, a monitor can be thought of as implementing a screen around the shared procedures and data.

2.2   Distributed Programming

Next in the development of concurrent programming languages are the ones developed for computer systems in which processes do not operate on a shared memory but, instead, are distributed over multiple sites. Characteristic for a *distributed program* is that its computation is split up into smaller computations, each of which is delegated to one of the distributed processes. For these processes to be able to interact, they should have the possibility to exchange their computed results among each other. In the research on such *distributed programming languages*, which started in the 1970s, a prominent place is held by the languages of the Communicating Sequential Programming paradigm (csp) [28], like occam [29]. In csp, interaction between the distributed processes is accomplished via an underlying communication network that connects the different sites, along which the processes can exchange messages with each other. Since there are no shared variables, the synchronisation issues for mutual exclusion, as sketched above, do not arise in a distributed environment. However, other problems remain, like, for instance, the problem that in a distributed program, processes can be waiting for particular data to arrive that however will never be supplied. In other words, it should be ensured that a distributed program is free from the possibility of *deadlock*. In a situation of deadlock, the execution of a program is blocked because none of its processes can proceed. A typical cause of deadlock is the fact that all processes are waiting for *another* process to make the next move, such that consequently no process can make a next step. This issue is illustrated in the following example.

Example 2. (*Deadlock*)
Let us consider a distributed programming language that comprises actions $c!e$ and $c?x$ for communication between processes. The execution of the former action consists of evaluating the expression e after which the computed value is sent along a communication channel c. The execution of the latter action consists of receiving a particular value along the communication channel c, which is subsequently assigned to the variable x. Consider a concurrent program that is comprised of the two processes A and B, which are defined as follows:

$$A \equiv c?x \cdot d!f(x)$$
$$B \equiv d?y \cdot c!g(y),$$

where $\cdot$ denotes sequential composition. The process A first receives a value along the communication channel c. After that, it applies the function f to this value, and subsequently sends it along the channel d. Concurrently, the process B first receives a value along the channel d that is assigned to the variable y. The function g is subsequently applied to the variable y, which yields a result that is sent along the channel c. However, the processes find themselves in a deadlock situation; that is, the first step of the execution of A is to receive a value along the channel c. This value can be supplied by process B, but not until this process has received a value along the communication channel d. The latter value can be provided by the agent A in turn, but only after it has received a value

along c. Consequently, neither of the processes can proceed as each of them is waiting for the other process to make the first move.

Writing concurrent programs that are free from deadlock is by no way an easy task, since deadlock situations may not present themselves as obviously as in the above example. Therefore, various techniques have been developed to enable a profound analysis of the behaviour of concurrent programs.

There are two types of communication. The first one is called *synchronous* communication, like in csp, which corresponds to a form of communication in which a process that wants to communicate a particular data item to another process, waits until the recipient is ready to receive it. The second kind of communication is referred to as *asynchronous* communication, which denotes a form of communication in which a process sends a particular data item irrespective of the current status of the recipient. That is, if at the moment of communication, the latter process is not ready to process the message, this message is, for instance, temporarily stored in a buffer from which it can be extracted as soon as the recipient is able to handle it.

2.3   Concurrent Object-Oriented Programming

In the practice of writing programs in the above distributed programming languages, an important pattern of interaction appeared to be that between a client and a server process: the client wants a particular task to be performed and the server is able to do this. The interaction pattern between these two processes comprises the communication of a message from the client to the server, followed by a suspension of the client and the execution of the corresponding task by the server. After completion of the task, the computed result is sent to the client that subsequently resumes its computation.

This two-way exchange of data between a client and a server can be implemented in csp via two synchronous communication steps. In the first step, data is communicated from the client to the server, while the second step comprises the communication of data from the server to the client. In the meantime, the execution of the client is blocked. Normally, one server handles requests from multiple clients, each of which has its own communication channels that connect it to the server. Due to all these concurrent interactions, it can become quite hard for a programmer to keep understanding what is going on. This led to the introduction of the concept of a *rendezvous* [4], which collects the above steps of the interaction between a client and a server into one compound programming construct. This delivers the programmer from defining each individual step of the interaction.

The concept of a *remote procedure call* [7] is almost similar to that of a rendezvous. However, in a remote procedure call, an entirely new server process is created to handle the call. The client process can thus be viewed upon as performing the corresponding procedure itself; the execution only takes place at a remote site.

The rendezvous communication mechanism has been adopted in a new generation of distributed programming languages, which are the languages for *concurrent object-oriented programming* [2]. In this paradigm, a program consists of a collection of processes, which are called *objects*. These objects have their own set of variables and additionally are assigned a set of methods that can be invoked to operate on these

variables. In fact, an object gives rise to a form of *data encapsulation*, since other objects can inspect and change the state of the object only through the invocation of one of its methods. Typical examples of this paradigm are the object-oriented languages of [1], which are inspired by the actor model of computation [25], and the language pool [3]. The latter language has been designed to program populations of concurrently operating objects that dynamically evolve over time. That is, in this language, objects have the capability to create new objects, which causes the object population to increase. Communication between the objects takes place via method invocations, which are based on the rendezvous communication mechanism.

2.4 Concurrent Constraint Programming

In addition to the above paradigms for *procedural* programming, we consider the related research area of *declarative* programming. In essence, a declarative program specifies a particular problem that needs to be solved. The execution of the program then amounts to finding a solution for it. One class of concurrent declarative languages are the concurrent versions of the logic programming language prolog [21, 42], like for instance the language parlog [10].

At the end of the 1980s, the Concurrent Constraint Programming (ccp) [36] was developed, which presents a new perspective on the underlying philosphy of logic programming. In constraint programming, a problem is expressed declaratively by means of a set of constraints on variables; Any solution to the problem must satisfy all these constraints. The paradigm assumes as input a particular constraint system, which is an abstract model of information. A constraint system consists of a a set of basic pieces of information that are expressed in a constraint language (such as a decidable fragment of first-order logic), which can be combined by means of a conjunction operator. Moreover, the constraint system contains a particular ordering relation of the constraints. Examples of constraints are: $z - y = x$, $x + y \geq 4$ and $P(x, y) \wedge R(y, x)$.

The revolutionary starting point of ccp is that it abandons the traditional *memory-as-valuation* concept of von Neumann-computing, which underlies the traditional programming languages. In the traditional view, the memory of a computer is an assignment of values to variables. However, in ccp, computation is based upon a novel view, namely the view of the comuter memory as a *constraint* on the range of values that variables can take. The idea is that this constraint is refined over and over again, until it represents the final result of the computation.

The computational model of ccp is based upon a set of concurrently operating processes that communicate with each other by means of a global store. This store is represented by a conjunction of constraints that express partial information on the values of the variables that are involved in their computations. The idea is that the multiple processes refine the partial information by adding new constraints to the store, until ultimately, the store contains the final solution to the problem. An example of an implemented concurrent constraint programming language is the language Oz [41].

In ccp, the operation `tell`(φ) is used to add a constraint φ to the store. In order for the processes to communicate and synchronise with each other, there is an additional operation `ask`(φ) that is used to test if the store entails the constraint φ. If the test succeeds then the corresponding process resumes its execution, otherwise its execution

is suspended until φ is indeed entailed by the store through updates by other processes. So, a process that executes $\mathtt{ask}(x \geq 1)$ to ask for the information $x \geq 1$, can can resume its execution after for instance, two other processes have executed $\mathtt{tell}(x + y \geq 4)$ and $\mathtt{tell}(y = 3)$, respectively.

The introduction of the ccp paradigm means an important step in the research on concurrent programming, because it yields a novel view on programming. Instead of the manipulation of variables, which is characteristic for the imperative languages, programming in this paradigm amounts to the computation with *information*.
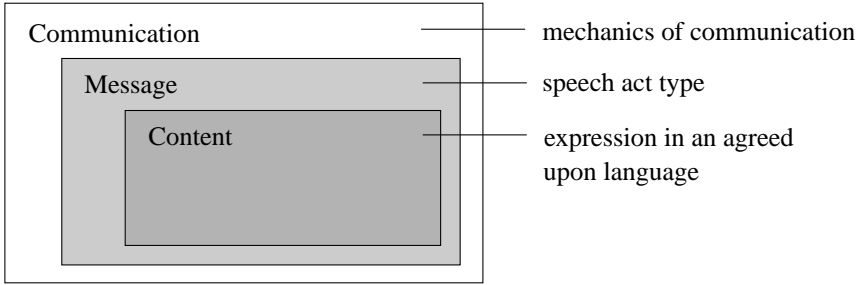
2.5   Agent Communication

In our opinion, the study of programming languages for multi-agent systems can be thought of as a next step in the research on concurrent programming languages. An essential aspect of multi-agent systems is that communication between agents proceeds at a higher level of abstraction in comparison with for instance object-oriented systems. That is, in object-oriented programming, an object is an encapsulated unit of data, with which other objects can interact through an invocation of one of its methods. Communication between agents takes place at a higher level of abstraction, involving propositions that are believed to be true or false, actions that are requested to be executed and goals that are to be achieved. One of the first proposed agent-oriented programming languages is the language agent-0 [39] in which agents are directly programmed in terms of mental concepts as their beliefs, capabilities and commitments. Other programming languages followed, like the languages placa [44], concurrent metatem [20], desire [9], agentspeak [35] and 3apl [26].

With respect to their communication aspects, there is a close connection between the paradigm of concurrent constraint programming and the field of multi-agent programming. In both paradigms, the communication of *information* plays a central role. However, whereas ccp is suited for processes that communicate with each other by means of a global store, in multi-agent systems, agents are typically *distributed* over multiple sites [8].

One of the topics of current research on agent communication is the development of standard *agent communication languages* that enable agents from different platforms to interact with each other on a high level of abstraction [31, 40]. The most prominent communication languages are the language kqml [18] and the language fipa-acl [19, 33]. In essence, an agent communication language provides a set of communication acts that agents in a multi-agent system can perform. The purpose of these acts is to convey information about an agents own mental state with the objective to effect the mental state of the communication partner.

Communication actions of agent communication languages are comprised of a number of distinct layers. Figure 1 depicts the three-layer model of kqml. The first layer of kqml consists of the informational content of the communication action. This content is expressed in some agreed-upon language, like a propositional, first-order or some other knowledge representation language. This correpsonds to the constraint language of ccp. The second layer of the communication action expresses a particular attitude towards the informational content in the form of a speech act. Examples of speech acts are $\mathtt{tell}$ to express that the content is believed to hold, $\mathtt{untell}$ to express that φ is not believed to

Fig. 1. Layers of the agent communication language kqml

hold or `ask` to ask whether the content is believed to hold. Finally, the third layer deals with the mechanics of communication, involving aspects like the channel along which the communication takes place and the direction of the communication (that is, sent or received).

An example of a communication action is: $c \,!\, \texttt{ask}(p)$. The content layer of the action consists of the proposition p, the message layer of the speech act `ask` and the communication layer of the communication channel c and the operator '!'. As noted, the operator '!' indicates that the message is sent along the communication channel, while the anticipated receipt of messages is indicated by the operator '?'.

For a clear understanding of agent communication we find it important not to consider communication actions in isolation, but to study them in the larger context of the multi-agent system in which they are performed. In this larger context, we can study aspects of conversations and dialogues, such as the specific order in which communication actions are executed, the conditions under which they take place and the effects they have on the (mental) states of the agents that are involved (see also [23]).

Therefore, we add one extra level to the three-layer model of kqml, namely the layer of the *multi-agent system*. We consider multi-agent systems that are defined in terms of a programming language. We assume the programming language to contain basic programming concepts, such as actions to examine and manipulate an agent's mental state, the aforementioned communication actions for interaction between agents, operators to make complex agent programs such as sequential composition '·', non-deterministic choice '+', parallel composition '&' and recursion and finally, operators to combine individual agent programs to form multi-agent programs like parallel composition '||'.

3   Semantic Approaches

One of the most prominent issues in the study of agent communications concerns their semantics. The current situation is that agent communication languages like kqml and fipa-acl are not fully understood from a semantical point of view [46].

In this paper, we consider some essential semantic aspects of agent communication. We will do this on the basis of ACPL (Agent Communication Programming Language), which is a formal framework that identifies basic aspects of agent communication [13–17]. In contrast to the languages fipa-acl and kqml, ACPL is supported by a semantic

foundation. The computational model of ACPL consists of an integration of the two different paradigms of ccp (Concurrent Constraint Programming) and csp (Communicating Sequential Processes). The constraint programming techniques are used to represent and process information, whereas the communication mechanism of ACPL is described in terms of the synchronous handshaking mechanism of csp.

We consider each of the four layers of agent communication. We start with the semantics of the content layer. Following constraint programming, in ACPL, information from the content layer is represented in terms of a constraint system. A constraint system is an abstract model of information. For the current purposes one can think of it as a set of basic pieces of information, which can be combined to form complexer constraints by means of a conjunction operator $\wedge$. For instance, constraints can be formulas from propositional logic, like p and $p \rightarrow q$. Constraints are ordered by means of an information-ordering. For instance, q contains less information than $p \wedge (p \rightarrow q)$. Usually, the reverse of the information-ordering is considered, which is called the entailment relation, denoted as $\vdash$. For instance, we have $p \wedge (p \rightarrow q) \vdash q$. The entailment relation defines the semantics of the content layer.

The second layer of agent communication involves speech act types. We assume an extension of the entailment relation of the constraint system that includes speech acts. For instance, given the constraints ψ and φ, we can stipulate:

$$\mathtt{untell}(\varphi) \vdash \mathtt{untell}(\psi) \;\Leftrightarrow\; \psi \vdash \varphi,$$

which expresses the anti-monotonicity of the speech act $\mathtt{untell}$. So, for instance, we have $\mathtt{untell}(p) \vdash \mathtt{untell}(p \wedge q)$, or in other words $\mathtt{untell}(p \wedge q)$ contains less information than $\mathtt{untell}(p)$. Other stipulations are for instance:

$$\mathtt{tell}(\neg\varphi) \vdash \mathtt{untell}(\varphi)$$
$$\mathtt{untell}(\varphi) \nvdash \mathtt{tell}(\neg\varphi),$$

which express some possible relations between the speech acts $\mathtt{tell}$ and $\mathtt{untell}$. The reason why $\mathtt{untell}(\varphi)$ does not entail $\mathtt{tell}(\neg\varphi)$ is that an agent can believe neither φ nor $\neg\varphi$ to hold.

The third layer involves the communication channel and the direction of communication. There are many sorts of communication channels like one-to-one, one-to-many, many-to-one and many-to-many channels. Usually, we will consider one-to-one channels that have a unique sender and recipient associated with them. At this level, we consider the interplay between sending and anticipating the receipt of communication actions. In ACPL, the basic communication mechanism is synchronous. A synchronous communication step consists of a handshake between an agent that performs a communication action of the form $c\,!\,\mathtt{speech\_act}_1(\varphi_1)$ and an agent that performs a matching communication act of the form $c\,?\,\mathtt{speech\_act}_2(\varphi_2)$ along the same channel c. For them to match it is required that the sent message $\mathtt{speech\_act}_1(\varphi_1)$ contains at least as much information as the message that is anticipated to be received, or in terms of the entailment relation:

$$\mathtt{speech\_act}_1(\varphi_1) \vdash \mathtt{speech\_act}_2(\varphi_2).$$

For instance, employing the above-mentioned relations between the speech acts $\mathtt{tell}$ and $\mathtt{untell}$, we have that $c\,!\,\mathtt{tell}(\neg p)$ matches with $c\,?\,\mathtt{untell}(p)$, but $c\,!\,\mathtt{untell}(p)$ does

not match with $c\,?\,\mathtt{tell}(\neg p)$. By means of the synchronous communication mechanism different forms of asynchronous communication can be modelled, such as for instance sending a question without waiting for its answer (see [17] for more details).

Finally, we consider the semantics of the fourth layer. This layer consists of the context in which the communication actions take place: the multi-agent system. We assume multi-agent systems to be developed in terms of a particular programming language. The semantics of programming languages provide a rigorous mathematical description of the meaning of their symbols. An important motive to develop the semantics of a programming languages is that it defines a precise standard for its implementations. Moreover, the semantics allows us to study and understand the interplay between communication acts and the programming constructs.

Before we continue let us consider some simple examples.

Example 3. (Semantic distinctions)
The semantics of an agent communication language among others should allow us to identify in what ways (if at all) the following programs differ:

$$(1)\ c\,!\,\mathtt{tell}(p) \cdot c\,!\,\mathtt{tell}(q)$$
$$(2)\ c\,!\,\mathtt{tell}(q) \cdot c\,!\,\mathtt{tell}(p)$$
$$(3)\ c\,!\,\mathtt{tell}(p \wedge q)$$
$$(4)\ c\,!\,\mathtt{tell}(p \wedge q) + c\,!\,\mathtt{tell}(p)$$

In ACPL, (1) and (2) have a different meaning because of the different order in which messages are exchanged. Programs (1) and (3) semantically differ because of the different number of exchanged messages. However, and this may be to some readers' surprise, there are circumstances under which there is no semantic difference between (3) and (4). We will come back to this issue later when we discuss full abstractness.

In the research on semantics of programming languages, there are several different methods to provide a language with a semantics [24, 43]. The most important methods are the axiomatic approach, operational approach and denotational approach.

Axiomatic Semantics. The first approach to the semantics of programming languages is the axiomatic approach, which constitutes an implicit form of giving semantics. In this approach, the meaning of the language is not explicitly defined but given in terms of properties that the language concepts satisfy. Usually these properties are formally derived by means of inference rules from a set of axioms.

The current approaches to the semantics of agent communication languages as kqml and fipa-acl belong to this class. In these frameworks, the semantics of a program P is defined by a triple

$$\{pre\}\ P\ \{post\},$$

where pre denotes a precondition that holds before the execution of P and $post$ constitutes a postcondition that hold afterwards. For instance, in [30], a semantics of kqml is presented in which these conditions are based on *speech act theory*, which is a model of human communication [5, 38]. Additionally, following the approach of [45], the axiomatic semantics of the message $\mathtt{tell}(\varphi)$ communicated from the agent i to the agent j can be defined by:

$$\{\mathbf{B}_i\varphi\}\ \mathtt{tell}(\varphi)\ \{\Diamond\mathbf{B}_j\varphi\},$$

which expresses that if before the execution of the action $\texttt{tell}(\varphi)$ the sender i believes the information φ to hold, then afterwards this information is eventually (denoted by the operator $\Diamond$) believed to hold by the recipient j.

The axiomatic approach is not generally thought of as a satisfactory way of giving formal semantics [37]. In general, the knowledge of only its properties is not sufficient for a thorough understanding of the language. The approach is therefore typically used to provide preliminary specifications of programming languages, which give the user insight in the important aspects of the languages. The axiomatic semantics should however be underpinned by other forms of semantics, in particular an *operational* semantics that gives insight in the implementations of the language, and a *denotational* semantics that provides an exact meaning of the language concepts.

Operational Semantics. An intuitive view of the execution of a program is to describe it in terms of the evolution of an abstract machine. The state of this machine is comprised of a control part consisting of the instructions that are to be executed and secondly, a data compartment that collects the data and information structures that are being manipulated. The execution of the program is then a sequence of subsequent transitions of the abstract machine, where the point of control moves along the program instructions. This form of semantics is referred to as *operational semantics* [34]. A major advantage of having an operational semantics is that an implementation of the language can be based upon an implementation of the corresponding abstract machine. To illustrate the approach, we consider the following (simplified version of a) transition from ACPL:

$$\langle c\,!\,\texttt{tell}(\varphi) \cdot P, state \rangle \overset{c\,!\,\texttt{tell}(\varphi)}{\longrightarrow} \langle P, state \rangle \quad \text{if } state \vdash \varphi.$$

The operational reading of the program $c\,!\,\texttt{tell}(\varphi) \cdot P$ is that provided that φ is true of the agent's current state, which is noted by the condition $state \vdash \varphi$, it amounts to sending the information φ along the communication channel c, which is denoted by the label $c\,!\,\texttt{tell}(\varphi)$, after which the program P denotes the part of the program that will be executed next. The state of the agent remains invariant under the transition. In the semantic framework of ACPL, this action of telling information is just one part of a communication step, the other part is given by the transition of a corresponding agent in the system that anticipates the receipt of a matching message along c.

Denotational Semantics. In this methodology, each syntactic entity of a programming language is assigned a meaning, which is called its denotation. This form of semantics has the advantage that the different parts of a programming language can be studied in isolation; i.e., it gives a precise definition of what each individual language concept really means. To illustrate this approach, we consider the (simplified form of the) denotational semantics of ACPL.

This semantics makes use of *communication histories*. A communication history is a sequence of communication actions that have taken place. There are two kinds of communication histories: local and global. A local communication history consists of the communication actions that an individual agent has performed. It contains actions of the form $c!\texttt{speech\_act}(\varphi)$ and $c?\texttt{speech\_act}(\varphi)$. A global history is comprised of the communication actions that have taken place in a multi-agent system. It contains tuples

of the form:
$$(c, \texttt{speech\_act}_1(\varphi_1), \texttt{speech\_act}_2(\varphi_2)).$$

Here, c denotes the channel along which has been communicated, $\texttt{speech\_act}_1(\varphi_1)$ and $\texttt{speech\_act}_2(\varphi_2)$ denote the matching communication actions of the sending and receiving agent, respectively. Matching means $\texttt{speech\_act}_1(\varphi_1) \vdash \texttt{speech\_act}_2(\varphi_2)$.

The semantics $[\![\ ]\!]$ of ACPL maps a program to the set of communication histories that it generates. This is a *set* because programs can give rise to more than one execution due to non-determinism. For instance, we have:

$$[\![c \,!\, \texttt{tell}(\varphi) \cdot P]\!] \;=\; \{c \,!\, \texttt{tell}(\varphi) \cdot h \mid h \in [\![P]\!]\},$$

which says that the meaning of the program $c \,!\, \texttt{tell}(\varphi) \cdot P$ is given by the set of local communication histories h as generated by the program P, which are *prefixed* with the act of telling the information φ along the channel c.

4   Semantic Properties

In defining the denotational semantics of programming languages, the principle of *compositionality* plays a crucial role. This principle states that the meaning of a compound program can be derived from the meaning of its components. For instance, the denotational semantics of the parallel composition of two agent programs P_1 and P_2 can be derived from the denotational semantics of its two programs. That is, the global communication history of the multi-agent program $P_1 \,||\, P_2$ consisting of matching communication actions can be derived from the local communication actions of the individual agent programs P_1 and P_2. Formally, this can be defined as follows:

$$[\![P_1 \,||\, P_2]\!] = \{h \mid h \!\upharpoonright\! P_1 \in [\![P_1]\!] \text{ and } h \!\upharpoonright\! P_2 \in [\![P_2]\!]\}$$

where $h \!\upharpoonright\! P_i$ denotes the projection of the global communication history h to the communication actions of the agent P_i, for $i = 1, 2$. So, for instance, we have:

$$[\![P_1]\!] = \{(c \,!\, \texttt{ask}(p)) \cdot (d \,?\, \texttt{untell}(p)) \cup$$
$$(c \,!\, \texttt{ask}(p)) \cdot (d \,?\, \texttt{tell}(p))\}$$

$$[\![P_2]\!] = \{(c \,?\, \texttt{ask}(p)) \cdot (d \,!\, \texttt{tell}(\neg p))\}$$

$$[\![P_1 \,||\, P_2]\!] = \{(c, \texttt{ask}(p), \texttt{ask}(p)) \cdot (d, \texttt{tell}(\neg p), \texttt{untell}(p))\}$$

The reason why the global history $(c, \texttt{ask}(p), \texttt{ask}(p)) \cdot (d, \texttt{tell}(\neg p), \texttt{tell}(p))$ is not part of $[\![P_1 \,||\, P_2]\!]$ is that the communication acts $d \,!\, \texttt{tell}(\neg p)$ and $d \,?\, \texttt{tell}(p)$ do not match.

An equivalent formulation of the principle of compositionality is that if one of the components of a program is replaced by a component that has exactly the same meaning, the meaning of the program is preserved. Formally, this is phrased as follows:

If $[\![P_1]\!] = [\![P_2]\!]$ then for all contexts C we have $[\![C[P_1]\,]\!] = [\![C[P_2]\,]\!].$

In the area of concurrency, the semantics of programming languages are usually defined relative to a notion of *observable behaviour*, which exactly captures the aspects of the behaviour of the systems that an external observer is interested in. In reasoning about the behaviour of multi-agent systems, we are typically not interested in all details of the execution of the system. Important aspects are the communication histories and the agents' mental states at some specific points during the execution, such as for instance right before and after a performed communication action.

Furthermore, a semantics is called *correct* if the observable behaviour can be extracted from the semantics. As an example we take as our observable behaviour whether a multi-agent system enters into a deadlock situation or successfully terminates. In order for the above denotational semantics $\llbracket\;\rrbracket$ to be correct, it needs to be refined with deadlocking behaviour. A solution to this is the introduction of *failure sets* [6]. In ACPL, failure sets consist of all communication actions that do not match with the current communication action that an agent wants to execute next. The corresponding form of semantics is referred to as *failure semantics*.

Finally, the semantics of a programming language can make unnecessary distinctions. This is the case if two programs have a different meaning but this difference cannot be observed, that is, there is no context in which they exhibit different observable behaviour. A semantics that does not make such unnecessary distinctions with respect to the observable behaviour is called *fully-abstract*. The failure semantics of ACPL is proven to be fully-abstract [16].

For instance, consider again the programs (3) and (4) of Example 3. It can be formally proven that there does not exist a context in which the programs (3) and (4) exhibit different observable behaviour. As the failure semantics of ACPL is fully-abstract, both programs thus have the same failure semantics. The crucial observation here is that any communication action that matches $c\,!\,\mathtt{tell}(p)$ also matches $c\,!\,\mathtt{tell}(p\wedge q)$. In general, we could say that sending a message includes sending all messages that contain less information. A similar property holds for the anticipated receipt of messages. There is no observable difference between the following programs (5) and (6):

$$(5)\; c\,?\,\mathtt{tell}(p)$$
$$(6)\; c\,?\,\mathtt{tell}(p) + c\,?\,\mathtt{tell}(p\wedge q)$$

Any communication action that matches $c\,?\,\mathtt{tell}(p\wedge q)$ also matches $c\,?\,\mathtt{tell}(p)$. In other words, anticipating the receipt of a messages includes anticipating the receipt of all messages that contain more information.

Once the semantics of a programming language has been established, it allows us to consider the *specification* and *verification* of agent communication. Verification amounts to the process of checking whether a program satisfies desired behaviour as expressed by a specification. Specifications are usually defined in what is called an *assertion language*. An example of an assertion in the assertion language of ACPL is the following assertion Ψ:

$$\forall i(h(i) = (c, \mathtt{ask}(p), \mathtt{ask}(p)) \rightarrow \exists j(j > i \wedge$$
$$((h(j) = (d, \mathtt{tell}(p), \mathtt{tell}(p)) \wedge Bel_B(p))) \vee h(j) = (d, \mathtt{untell}(p), \mathtt{untell}(p)))).$$

If we suppose that c and d are one-to-one communication channels that connect the agents A and B, the above assertion expresses that if at some point i in the communication

history h agent A asks agent B whether the proposition p holds then at some point j later in history, either agent B tells A that it believes p to hold after which B also believes that φ holds or agent B tells A that it does not believe p to hold.

In the above assertion we find an example of a *conversation policy* [22], namely the policy that if an agent A is asked by an agent B whether a particular proposition holds then A subsequently answers B whether it believes the proposition to hold or not.

Note that both the multi-agent programming language and the assertion language have their own syntax and semantics. They are linked through the underlying computational model: A particular multi-agent program satisfies a particular assertion if the assertion is true for all computations that the multi-agent program gives rise.

In [17], a compositional verification calculus for ACPL is defined. This calculus can be used to verify that a particular multi-agent system satisfies the above assertion Ψ. It is comprised of rules of the form:

$$\frac{P_1 \; sat \; \Phi_1 \cdots P_n \; sat \; \Phi_n}{P \; sat \; \Phi}$$

where P denotes a multi-agent program that is composed of the components $P_1, \ldots, P_n$ and Φ constitutes an assertion that is obtained from the assertions $\Phi_1, \ldots, \Phi_n$. These rules can be used to formally derive the specification of the behaviour of the program P can from the specification of its components. On the basis of this calculus it is possible to implement (semi-)automatic verification procedures. This is a subject of future research.

5   Concluding Remarks

In this paper, we have considered the semantics of agent communication. We have sketched the research on communication in concurrent programming paradigms, starting with communication via shared variables and resulting in communication in multi-agent systems. We have considered the four different layers that play a role in giving semantics to agent communication and the main approaches for developing semantics of programming languages. On the basis of the ACPL framework (Agent Communication Programming Language) we have discussed semantic issues involved in programming agent communication, including communication histories, compositionality, observational behaviour, failure semantics and full abstractness. Finally, we have considered the specification and verification of agent communication. In our view, these issues play an important part in defining a semantic foundation for agent communication languages as kqml and fipa-acl, which is a subject of further research.

Acknowledgements

References

1. G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.
2. G. Agha, P. Wegner, and Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, Massachusetts, 1993.
3. P.H.M. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.
4. G.R. Andrews. *Concurrent Programming, Principles and Practice*. The Benjamin Cummings Publishing Company, Inc., Redwood City, California, 1991.
5. J.L. Austin. *How to do Things with Words*. Oxford University Press, Oxford, 1962.
6. J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Readies and failures in the algebra of communicating processes. *SIAM Journal on Computing*, 17:1134–1177, 1988.
7. A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.
8. A.H Bond and L. Gasser. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.
9. F. Brazier, B. Dunin-Keplicz, N. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *Proceedings of International Conference on Multi-Agent Systems (ICMAS'95)*, pages 25–32. MIT Press, 1995.
10. K. Clark and S. Gregory. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.
11. E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
12. E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.
13. R.M. van Eijk. *Programming Languages for Agent Communication*. PhD thesis, Utrecht University, Mathematics and Computer Science, 2000.
14. 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.
15. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. On dynamically generated ontology translators in agent communication. *International Journal of Intelligent Systems*, 16(5):587–607, 2001.
16. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Fully-abstract model for the exchange of information in multi-agent systems. *Theoretical Computer Science*. To appear, 2002.
17. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. A verification framework for agent communication. *Autonomous Agents and Multi-Agent Systems*. To appear, 2002.
18. 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.
19. Foundation For Intelligent Physical Agents FIPA. Specification part 2 – agent communication language. Version dated 10th October 1997, 1997.
20. M. Fisher. A survey of concurrent MetateM– the language and its applications. In *Proceedings of First International Conference on Temporal Logic (ICTL'94)*, volume 827 of *Lecture Notes in Computer Science*, pages 480–505. Springer-Verlag, 1994.

21. P. Gibbens. *Logic with Prolog*. Oxford Applied Mathematics and Computing Science Series. Oxford University Press, New York, 1988.

22. M. Greaves, H. Holmback, and J. Bradshaw. What is a conversation policy? In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 118–131. Springer-Verlag, Heidelberg, 2000.

23. F. Guerin and J. Pitt. A semantic framework for specifying agent communication languages. In *Proceedings of fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, pages 395–396, Los Alamitos, California, 2000. IEEE Computer Society.

24. C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, Cambridge, Massachusetts, 1992.

25. C. Hewitt. Viewing control as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

26. K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J.-J.Ch Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2:357–401, 1999.

27. C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

28. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

29. G. Jones. *Programming in Occam*. Prentice-Hall International, New York, NY, 1987.

30. Y. Labrou and T. Finin. Semantics for an agent communication 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 *Lecture Notes in Artificial Intelligence*, pages 209–214. Springer-Verlag, 1998.

31. Y. Labrou, T. Finin, and Y. Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14(2):45–52, 1999.

32. P. Noriega and C. Sierra, editors. *Agent Mediated Electronic Commerce*, volume 1571 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

33. J. Pitt and A. Mamdani. Some remarks on the semantics of FIPA's agent communication language. *Autonomous Agents and Multi-Agent Systems*, 2(4):333–356, 1999.

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

35. A.S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In W. van der Velde and J.W. Perram, editors, *Agents Breaking Away*, volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 42–55. Springer-Verlag, 1996.

36. V.A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Massachusetts, 1993.

37. D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc. Newton, Massachusetts, 1986.

38. J.R. Searle. *Speech acts: An essay in the philosophy of language*. Cambridge University Press, Cambridge, England, 1969.

39. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

40. M.P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

41. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343, Berlin, 1995. Springer-Verlag.

42. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts, 1986.

43. R.D. Tennent. *Semantics of Programming Languages*. Prentice Hall, Hertfordshire, 1991.

44. S.R. Thomas. *PLACA, an Agent Oriented Programming Language*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, 1993.

45. M. Wooldridge. Verifying that agents implement a communication language. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 52–57, 1999.
46. M. Wooldridge. Semantic issues in the verification of agent communication. *Autonomous Agents and Multi-Agent Systems*, 3(1):9–31, 2000.
47. M. Wooldridge and N. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.