Process Algebra for Agent Communication: A General Semantic Approach

Rogier M. van Eijk[1], Frank S. de Boer[1],
Wiebe van der Hoek[2], and John-Jules Ch. Meyer[1]

[1] Institute of Information and Computing Sciences
Utrecht University, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands
{rogier,frankb,jj}@cs.uu.nl

[2] Department of Computer Science
University of Liverpool,
Liverpool L69 7ZF, United Kingdom
wiebe@csc.liv.ac.uk

Abstract. In this paper, we consider the process algebra ACPL, which models the basics of agent communication. This algebra combines the information-processing aspects of Concurrent Constraint Programming (CCP) with a generalisation of the synchronous handshaking communication mechanism of Communicating Sequential Processes (CSP). The operational semantics of ACPL is given in terms of a transition system that consists of local and global transition rules. The local rules describe the operational behaviour of agents, like the local effects of communication actions. The global rules define the operational behaviour of multi-agent systems including the matching of communication actions. We show how ACPL provides a general basis to address the semantics of agent communication languages such as KQML and FIPA-ACL. Finally, we address several extensions of the basic algebra.

1 Introduction

One of the topics of current research on multi-agent systems 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 [21,30]. The most prominent communication languages are the language KQML [13] and the language FIPA-ACL [14,24]. 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 agent's 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 other knowledge representation language. The second layer of the communication action expresses a particular attitude towards the informational content in the form of a *speech*

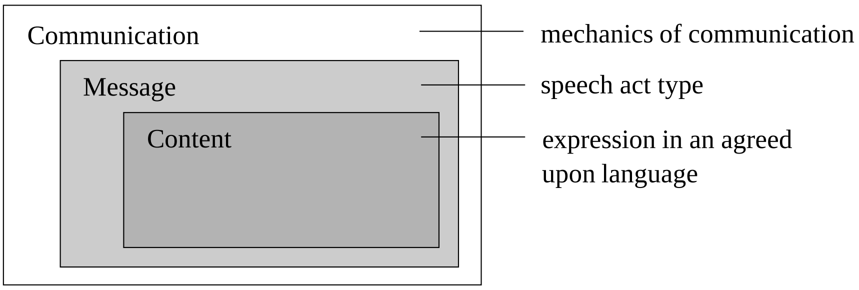
Communication
Message
Content
mechanics of communication
speech act type
expression in an agreed upon language

Fig. 1. Layers of the agent communication language KQML

act [1,29]. Examples of speech acts are `tell` to express that the content is believed to hold, `untell` to express that the content is not believed to 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 “!”. The operator “!” indicates that the message is sent along the communication channel (the anticipated receipt of messages will be 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 [17]). 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 particular 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 and operators to combine individual agent programs to form multi-agent programs.

Process Algebra

The main principle of structured programming, as originally advocated by Dijkstra, is that under all circumstances a programmer must keep the program within his or her intellectual grasp [7,8]. During the last decades many formalisms have been developed to obtain a thorough understanding of the different aspects of programming. *Process Algebra* is the common name of a family of abstract programming notations for reasoning about concurrently executing, communicating computer systems. These frameworks concentrate on the essential features of programming and thereby abstract from all implementation details.

In a process algebra, a process is defined in terms of a set of basic operators, like sequential composition, choice, parallelism and looping. These operators are typically given semantics through a *structural operational semantics*, originally developed by Plotkin [25]. An example is one of the rules that governs parallel composition:

$$\frac{P \xrightarrow{l} P'}{P \parallel Q \xrightarrow{l} P' \parallel Q}$$

The rule states that if it is possible for a process P to perform a computation step l, yielding process P', then it is also possible for the parallel composition of P and Q, to perform the computation step l, yielding the parallel composition of P' and Q. This reflects the interleaving model of parallel execution.

Usually, processes are considered with respect to a particular sort of observational behaviour. Two processes are considered equivalent if they exhibit the same observational behaviour. Equivalences can be formally proven by means of rewriting systems that consist of axioms and inference rules in the form of algebraic equivalences. An example of an algebraic equivalence is:

$$P \parallel Q = Q \parallel P$$

expressing that parallel composition is a commutative operator.

The main algebraic approaches to concurrency are Hoare's Communicating Sequential Processes (CSP) [19,6], Milner's Calculus of Communicating Systems (CCS) [22], and Bergstra & Klop's Algebra of Communicating Processes (ACP) [3]. Over the last years, many extensions and refinements of these algebras have been developed, like extensions with time [2], channel-passing (e.g., π-calculus [23]), constraints [5], and higher-order communication [32] in which processes themselves can be passed in a communication step.

In [4,9,10,11,12], a process algebra for agent communication has been developed. The computational model of this algebra, which is called Agent Communication Programming Language (ACPL), consists of an integration of the declarative paradigm of Concurrent Constraint Programming (CCP) [27] and the imperative programming paradigm of CSP. The constraint programming techniques are used to represent and process information, whereas the communication mechanism of ACPL is based upon a generalisation of the synchronous handshaking mechanism of CSP. The generalisation consists of the exchange of *information*, i.e., constraints, instead of the communication of simple *values*. In essence, a communication step consists of a handshake between an agent that sends information φ and an agent that anticipates the receipt of information ψ, where for successful communication it is required that φ contains at least as much information as ψ.

The paper is organised as follows. In Section 2, we define the syntax of the multi-agent language ACPL. The structural operational semantics of this language is described in Section 3. The subject of Section 4 is the application of the framework to agent communication languages as FIPA-ACL and KQML. Finally, in Section 5, we wrap up and provide some pointers to extensions of the basic algebra.

2 Syntax

In this section, we introduce the syntax of ACPL, which like Concurrent Constraint Programming is parameterised by a constraint system that is used to represent information.

Definition 1 (Constraint systems)
A *constraint system* C is a tuple $(C, \sqsubseteq, \wedge, \mathit{true}, \mathit{false})$, where C (the set of constraints, with typical element φ) is a set ordered with respect to $\sqsubseteq$, $\wedge$ is the least upperbound operation, and $\mathit{true}, \mathit{false}$ are the least and greatest elements of C, respectively.

A constraint system is an abstract model of information. It consists of a set of basic pieces of information, which can be combined to form more complex 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. That is, $\varphi \sqsubseteq \psi$ denotes that φ contains less information than ψ. For instance, q contains less information than $p \wedge (p \rightarrow q)$. Usually, the reverse of the information-ordering is used, which is called the entailment relation (e.g., a PROLOG interpreter, a theorem-prover and so on), denoted as "$\vdash$". For instance, we have $p \wedge (p \rightarrow q) \vdash q$. The entailment relation for instance indicates how the agent deals with negations; i.e., whether it employs a negation-as-failure strategy, a finite-failure strategy, and so on. Moreover, it can be thought of representing the agent's decision-making capabilities.

In order to model *hiding* of local variables and *parameter passing* in constraint programming, in [28] the notion of a constraint system is enriched with a hiding operator $\exists_x$ (which in the CCP literature is called a cylindrification operator, following [18]). This operator satisfies the usual properties of existential quantification, such as $\varphi \vdash \exists_x \varphi$, $\varphi \vdash \psi$ implies $\exists_x \varphi \vdash \exists_x \psi$, $\exists_x \exists_y \varphi \equiv \exists_y \exists_x \varphi$ and $\exists_x(\varphi \wedge \exists_x \psi) \equiv \exists_x \varphi \wedge \exists_x \psi$, where $\equiv$ denotes logical equivalence. We use the notation $\varphi[y/x]$ to denote the formula $\exists_x(d_{xy} \sqcup \varphi)$, which can be intepreted as the formula obtained from φ by replacing all the free occurrences of x by y. We also assume the generalisation $\varphi[\bar{y}/\bar{x}]$ to sequences of variables.

Aditionally, in order to model the dynamics of belief bases, we assume a particular *belief update operator* [15]. We use the notation $\varphi \circ \psi$ to denote an agent's belief base φ that has been updated with the information ψ. A constraint system together with existential quantification $\exists_x$ and an update operator $\circ$ constitute a *belief system*.

The main objective of the programming language defined below is to provide a generic framework for the exchange of information in multi-agent systems, which abstracts from the specific nature of the underlying belief system.

In the following definition, we assume a given set Chan of (unidirectional) communication channels, with typical element c.

In the remainder of this paper, we assume a particular belief system $\mathcal{B}$ to be given.

Definition 2 (Basic actions)
The *basic actions* of the programming language are defined as follows:

$$a ::= c!\varphi \mid c?\varphi \mid \mathsf{query}(\varphi) \mid \mathsf{update}(\varphi).$$

The execution of the output action $c!\varphi$ consists of sending the information φ along the channel c, which has to synchronise with a corresponding input $c?\psi$, for some ψ with $\varphi \vdash \psi$. In other words, the information φ can be sent along a channel c only if some information entailed by φ is anticipated to be received. The execution of an input action $c?\psi$, which consists of anticipating the receipt of the information ψ along the channel c, also has to synchronise with a corresponding output action $c!\varphi$, for some φ with $\varphi \vdash \psi$. The execution of a basic action $\mathsf{query}(\varphi)$ by an agent consists of checking whether the private store of the agent entails φ. On the other hand, the execution of $\mathsf{update}(\varphi)$ consists of updating the belief base with φ.

In the following definition, we assume a given set *Proc* of procedure identifiers, with typical element p.

Definition 3 (Statements)
The behaviour of an agent is then described by a *statement* S:

$$S ::= a \cdot S \;\mid\; S_1 + S_2 \;\mid\; S_1 \,\&\, S_2 \;\mid\; \mathsf{loc}_x S \;\mid\; p(\bar{x}) \;\mid\; \mathsf{skip}.$$

Statements are thus built up from the basic actions using the following standard programming constructs: action prefixing, denoted by "·"; non-deterministic choice, denoted by "+"; internal parallelism, denoted by "&"; local variables, denoted by $\exists_x S$, which indicates that x is a local variable in S; and (recursive) procedure calls of the form $p(\bar{x})$, where $p \in \mathit{Proc}$ constitutes the name of the procedure and $\bar{x}$ denotes a sequence of variables which constitute the actual parameters of the call. We assume that no information on a local variable x can be communicated. Hence, we additionally require that in $\exists_x S$ the variable x does not occur free in a communication of S; that is, for every communication action $c?\varphi$ or $c!\varphi$ of S we have $\exists_x \varphi \equiv \varphi$. Finally, skip denotes the empty statement.

Definition 4 (Multi-agent systems)
A *multi-agent system* A is defined as follows:

$$A ::= \langle D, S, \varphi \rangle \;\mid\; A_1 \parallel A_2 \;\mid\; \delta_H(A).$$

A *basic agent* in a multi-agent system is represented by a tuple $\langle D, S, \varphi \rangle$. The set D consists of procedure declarations of the form $p(\bar{x}) :- S$, where $\bar{x}$ denote the formal parameters of p and S denotes its body. In order to facilitate the operational description of procedure calls, we assume that D satisfies the following property:

$$\text{if } p(\bar{x}) :- S \in D \text{ then } p(\bar{y}) :- S[\bar{y}/\bar{x}] \in D$$

for all $\bar{x}$ and $\bar{y}$, where $S[\bar{y}/\bar{x}]$ denotes the statement S in which each constraint φ is replaced by $\varphi[\bar{y}/\bar{x}]$. The statement S in $\langle D, S, \varphi \rangle$ describes the behaviour of the agent with respect to its *private* store φ. The threads of S, i.e. the concurrently executing substatements of S, interact with each other via the private store of the basic agent by means of the actions $\mathsf{query}(\psi)$ and $\mathsf{update}(\psi)$. As in the operational semantics below the set D of procedure declarations will not change, we usually omit it from notation and simply write $\langle S, \varphi \rangle$ instead of $\langle D, S, \varphi \rangle$.

Additionally, a multi-agent system itself consists of a collection of concurrently operating agents that interact with each other only via a synchronous information-passing mechanism by means of the communication actions $c!\psi$ and $c?\psi$. Our choice for synchronous communication is motivated by the fact it can be used to model asynchronous communication as well, as we will see in Section 4. Note that we restrict to the parallel composition of agent systems, and leave sequential composition, non-deterministic choice and recursion at the level of multi-agent systems out of consideration.

Finally, the encapsulation operator δ_H with $H \subseteq \mathit{Chan}$, which stems from the process algebra ACP, is used to define local communication channels [3]. That is, $\delta_H(A)$ denotes a multi-agent system in which the communication channels in H are for internal use only and hence, cannot be used for communication with agents outside the system.

3 Operational Semantics

In this section, we consider the structural operational semantics of ACPL.

3.1 Transition Systems

The central idea of structural operational semantics is to define the meaning of a program directly in terms of the behaviour that it exhibits. More specifically, the behaviour of a program can be modelled as a sequence of *transitions* between consecutive *configurations*. A configuration denotes the state of the program at a particular point in its execution. A transition corresponds to an individual computation step of the program, reflecting the effects on the current configuration.

A simple and elegant formalism to define structural operational semantics is via the well-known technique of *transition systems*, originally developed in [25]. In short, a transition system collects a set of rules that are used for the formal derivation of computation steps of a program. These rules define the effects that the different programming constructs have on the current configuration of the program. Such a configuration not only contains a description of the state of the program, but also the part of the program that still needs to be executed after the transition. In its most general form, a *transition* looks as follows:

$$\langle P, \sigma \rangle \longrightarrow \langle P', \sigma' \rangle.$$

It denotes a computation step of the program P which changes the current state of the system σ to the state σ', where P' is identified to be the part of the program P that still needs to be executed. Assuming that programs have been defined *inductively*, we can define the transitions of a program in terms of the transitions of its components. For example, the transitions of a sequential composition of two programs P_1 and P_2 can be derived from the transitions of P_1 and the transitions of P_2. That is, the compound program performs the computation steps that the program P_1 executes, and upon termination of P_1, the computation steps that P_2 performs.

In general, transitions are formally derived by means of *transition rules*, which are of the following format:

$$\frac{\begin{array}{c}\langle P_1, \sigma_1\rangle \longrightarrow \langle P_1', \sigma_1'\rangle \\ \vdots \\ \langle P_n, \sigma_n\rangle \longrightarrow \langle P_n', \sigma_n'\rangle\end{array}}{\langle P, \sigma\rangle \longrightarrow \langle P', \sigma'\rangle} \quad \text{if } cond$$

Such a rule denotes that the transition below the line can be derived if the transitions above the line are derivable, provided that the condition $cond$ holds. Sometimes, we 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*, and is simply written as $\langle P, \sigma\rangle \longrightarrow \langle P', \sigma'\rangle$. A *transition system* is then a set of of transition rules.

In order to be able to describe communication, we can make use of *labelled transition systems* in which the transitions are of the following form:

$$\langle P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle.$$

The label l in this transition is used to denote the *mode* of the computation step. In our case, we distinguish between three different modes; viz., internal computation steps, input actions and output actions.

The advantage of using transitions systems is that they allow the operational semantics to closely follow the syntactic structure of the language. As an effect, if we view the configurations of the form $\langle P, \sigma\rangle$ as states of an abstract machine then the transitions specify the actions that this machine should perform. In fact, this machine could act as an *interpreter* for the language.

3.2 Local Transitions of ACPL

The structural operational semantics of ACPL is defined by means of a *local* and a *global* transition system. Given a set of declarations D, a local transition is of the form

$$\langle S, \varphi\rangle \xrightarrow{l} \langle S', \psi\rangle$$

where either l equals τ in case of an *internal* computation step, that is, a computation step which consists of the execution of skip or a basic action of the form $\mathsf{query}(\varphi)$ or $\mathsf{update}(\varphi)$, or l is of the form $c!\varphi$ or $c?\varphi$, in case of a communication step. We employ the symbol E to denote successful termination.

Definition 5 *(Transitions for basic actions)*

$$\begin{array}{ll}\langle \mathsf{query}(\varphi), \psi\rangle \xrightarrow{\tau} \langle E, \psi\rangle & \text{if } \psi \vdash \varphi \\ \langle \mathsf{update}(\varphi), \psi\rangle \xrightarrow{\tau} \langle E, \psi \circ \varphi\rangle & \\ \langle c!\varphi, \psi\rangle \xrightarrow{c!\varphi} \langle E, \psi\rangle & \\ \langle c?\varphi, \psi\rangle \xrightarrow{c?\varphi} \langle E, \psi\rangle & \end{array}$$

The actions $\mathsf{query}(\varphi)$ and $\mathsf{update}(\varphi)$ are the familiar operations from CCP which allow an agent to inspect and update its private store. The semantics of the basic action update is defined in terms of the belief update operator $\circ$ from the belief system $\mathcal{B}$. By adding new information the store can become inconsistent, such as for instance if the action $\mathsf{update}(x = 1)$ is performed in a situation where the store contains the information $x = 0$. We assume that such conflicts are resolved by the belief update operator $\circ$.

In the third transition, the information φ to be sent does not necessarily follow from the agent's belief base ψ. The programmer can try to accomplish sincerity by letting the output action precede by a test that φ follows indeed from the information store:

$$\mathsf{query}(\varphi) \cdot c!\varphi$$

In this case, the communication action will only be executed in case the test $\mathsf{query}(\varphi)$ has been successfully executed first. However, in the case of multiple concurrently operating threads in an agent, it is possible that the consecutive execution of these two actions is interleaved by another action that updates the private store. Due to this intermediate update it is possible that the information φ will not be entailed by the store at the moment of communication. Thus, sincerity assumes either that an agent has one single thread or that its belief base shows monotonically increasing behaviour; i.e., information is only added to the store and not removed from it.

In the fourth transition, the information φ that is anticipated to be received is not automatically added to the agent's belief state. The addition of this information can be controlled by the programmer through a subsequent execution of the update operator:

$$c?\varphi \cdot \mathsf{update}(\varphi).$$

Furthermore, we have the following rules for action prefixing, procedure calls and the programming constructs for non-deterministic choice and parallel composition.

Definition 6 *(Transition for prefixing)*

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

The computation step of a prefixed statement $a \cdot S$ corresponds to the execution of its prefix a. That is, the transition of the prefixed statement $a \cdot S$ is inferred from the transition of the action a, in which the label l is propagated together with the change of the private store from ψ to ψ'. Finally, the statement S is identified to be the part of $a \cdot S$ that needs to be executed next.

Definition 7 *(Transition for internal parallelism)*

$$\frac{\langle S_1, \psi\rangle \xrightarrow{l} \langle S_1', \psi'\rangle}{\begin{array}{c}\langle S_1 \,\&\, S_2, \psi\rangle \xrightarrow{l} \langle S_1' \,\&\, S_2, \psi'\rangle \\ \langle S_2 \,\&\, S_1, \psi\rangle \xrightarrow{l} \langle S_2 \,\&\, S_1', \psi'\rangle\end{array}}$$

A derivation rule that has two transitions below the line, is a shorthand notation for two derivation rules that each have one of these two transitions as its conclusion. The execution of a parallel statement $S \mathbin{\&} T$ is modelled as an interleaving of the computation steps of S and T. That is, an execution step of the composed statement $S \mathbin{\&} T$ is given by a computation step of one the statements S and T. Therefore in the above transition rule, the transition of the statement S_1 induces a transition of the compound statement $S_1 \mathbin{\&} S_2$ in which it acts as the left operand, as well a transition of the compound statement $S_2 \mathbin{\&} S_1$ in which it acts as the right operand. The statements $S_1' \mathbin{\&} S_2$ and $S_2 \mathbin{\&} S_1'$ then denote the part of the composed statements that remains to be executed, respectively.

Definition 8 *(Transition for non-deterministic choice)*

$$\frac{\langle S_1, \psi\rangle \xrightarrow{l} \langle S_1', \psi'\rangle}{\begin{array}{c}\langle S_1 + S_2, \psi\rangle \xrightarrow{l} \langle S_1', \psi'\rangle \\ \langle S_2 + S_1, \psi\rangle \xrightarrow{l} \langle S_1', \psi'\rangle\end{array}}$$

The computation steps of a non-deterministic choice $S + T$ are given by the transitions of either of the statements S and T. Hence, in the transition rule above, the transition of S_1 yields a transition for the compound statement $S_1 + S_2$ as well as for the compound statement $S_2 + S_1$. The part of the non-deterministic choice that remains to be executed is given by S_1'. The rule reflects that "$+$" is a commutative operator. Due to non-deterministic choice the execution of a multi-agent system can lead to different ending states. Note the difference with the rule for internal parallelism in which the statement S_2 remains to be executed as well.

Definition 9 *(Transition for local variables)*

$$\frac{\langle S, \varphi \circ \exists_x \psi\rangle \xrightarrow{l} \langle S', \psi'\rangle}{\langle \mathsf{loc}_x^{\varphi} S, \psi\rangle \xrightarrow{l} \langle \mathsf{loc}_x^{\varphi'} S', \psi \circ \exists_x \varphi'\rangle}$$

The syntax of the language is extended with a construct of the form $\mathsf{loc}_x^{\varphi} S$ denoting that in the statement S the variable x is a local variable, where the constraint φ collects the information on the local variable x. In this notation, the statement $\mathsf{loc}_x S$ is written as $\mathsf{loc}_x^{true} S$, denoting that the local constraints on x are initially empty.

The idea of the transition rule is that the transition of the construct $\mathsf{loc}_x^{\varphi} S$ is derived from the transition of the statement S. In order to achieve this, we need to replace the constraints on the global variable x in the state ψ (if present) by the constraints φ on the local variable x. This yields the state $\varphi \circ \exists_x \psi$. The statement S is then executed relative to this state. After one computation step, φ' denotes the new state and S' represents the part of S that still remains to be executed. In order to obtain from ψ' the resulting store, we remove the constraints on the local variable x from it and add the remainder to the old private store ψ, yielding the new state $\psi \circ \exists_x \varphi'$. Finally, the constraints on the local variable x need to be stored for later use; hence, the statement $\mathsf{loc}_x^{\varphi'} S'$ denotes the part of the program that needs to be executed next.

Note that no information on the local variable x can be communicated, because by definition x does not occur free in φ in case l is of the form $c?\varphi$ or $c!\varphi$.

Definition 10 *(Transition for procedure calls)*

$$\langle p(\bar{y}), \psi \rangle \xrightarrow{\tau} \langle S, \psi \rangle \text{ where } p(\bar{y}) :- S \in D$$

The transition of a procedure call is given by the replacement of the call by the body of the procedure.

Definition 11 *(Transition for skip)*

$$\langle \mathsf{skip}, \psi \rangle \xrightarrow{\tau} \langle E, \psi \rangle$$

The rule shows that the statement skip always succeeds and has no effects on the information store ψ.

3.3 Global Transitions of ACPL

A *global* transition is of the form $A \xrightarrow{l} A'$, where l indicates whether the transition involves an internal computation step, that is, $l = \tau$, or a communication, that is, $l = c!\varphi$ or $l = c?\varphi$.

Definition 12 *(Transitions for multi-agent systems)*
The following rule describes parallel composition by interleaving of the basic actions:

$$\frac{A_1 \xrightarrow{l} A_1'}{\begin{array}{c} A_1 \parallel A_2 \xrightarrow{l} A_1' \parallel A_2 \\ A_2 \parallel A_1 \xrightarrow{l} A_2 \parallel A_1' \end{array}}$$

In order to describe the synchronisation between agents we introduce a synchronisation predicate $|$, which is defined as follows. For all $c \in \mathit{Chan}$ and $\varphi, \psi \in \mathcal{B}$, if $\varphi \vdash \psi$ then

$$(c!\varphi \mid c?\psi) \text{ and } (c?\psi \mid c!\varphi).$$

In all other cases, the predicate $|$ yields the boolean value false. We then have the following synchronisation rule:

$$\frac{A_1 \xrightarrow{l_1} A_1' \quad A_2 \xrightarrow{l_2} A_2'}{A_1 \parallel A_2 \xrightarrow{\tau} A_1' \parallel A_2'} \text{ if } l_1 \mid l_2$$

This rule shows that an action of the form $c?\psi$ only matches with an action of the form $c!\varphi$ in case ψ is entailed by φ. In all other cases, the predicate $|$ yields false and therefore no communication can take place.

Finally, *encapsulation* of communications along a set of channels H is described by the rule:

$$\frac{A \xrightarrow{l} A'}{\delta_H(A) \xrightarrow{l} \delta_H(A')} \text{ if } \mathit{chan}(l) \cap H = \emptyset$$

where chan is defined by $\mathit{chan}(c!\varphi) = \mathit{chan}(c?\varphi) = \{c\}$ and $\mathit{chan}(\tau) = \emptyset$.

4 Agent Communication Languages

The framework ACPL provides a general basis for the semantics of agent communication languages. Consider the different layers of the agent communication language KQML in Figure 1. Constraint systems can be used to represent the content layer. With respect to the second layer we assume an extension of the entailment relation of the constraint system that includes speech acts. For example, a KQML expression consisting of a content expression φ that is encapsulated in a message wrapper containing the speech act `untell`, which allows to derive negative information in terms of the closed world assumption [26], is represented by the expression $\texttt{untell}(\varphi)$. This operator can be defined by an extension of the information ordering of the constraint system. For instance, given the constraints ψ and φ, we define:

$$\psi \nvdash \varphi \;\;\Leftrightarrow\;\; \psi \vdash \texttt{untell}(\varphi).$$

Assuming that ψ represents the belief base of an agent, this rule formalises the closed world assumption. Assuming $\neg p \nvdash p$, we can for instance derive: $\neg p \vdash \texttt{untell}(p)$. Note that we cannot derive $\texttt{untell}(p) \vdash \neg p$.

The anti-monotonicity property of the `untell` operator is expressed by:

$$\texttt{untell}(\psi) \vdash \texttt{untell}(\varphi) \;\;\Leftrightarrow\;\; \varphi \vdash \psi.$$

So, for instance, we have $\texttt{untell}(p) \vdash \texttt{untell}(p \wedge q)$, or in other words $\texttt{untell}(p \wedge q)$ contains less information than $\texttt{untell}(p)$.

The general use of the entailment relation in the semantics of communication actions allows us to abstract from among others the following :

- the particular *syntax* of information, for instance, $\texttt{untell}(p \wedge q)$ entails $\texttt{untell}(q \wedge p)$ and vice versa.
- redundant logical strength, e.g., $\texttt{untell}(p)$ entails $\texttt{untell}(p \wedge q)$.
- the *kind* of communicated information, e.g., simple constraints on the domain of discourse or information containing speech acts.

The third layer involves the communication channel and the direction of communication. At this level, we consider the interplay between sending and anticipating the receipt of information. 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 $c\,!\,\texttt{speech\_act}_1(\varphi_1)$ and an agent that performs a matching communication act of the form $c\,?\,\texttt{speech\_act}_2(\varphi_2)$ along the same channel c. For them to match it is required that the sent message $\texttt{speech\_act}_1(\varphi_1)$ contains at least as much information as the message that is anticipated to be received, or in terms of the entailment relation:

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

For instance, employing the above stipulations, we have that $c\,!\,\neg p$ matches with $c\,?\,\texttt{untell}(p)$, but $c\,!\,\texttt{untell}(p)$ does not match with $c\,?\,\neg p$.

Thus, the basic communication mechanism of ACPL is *synchronous*, i.e., the receipt of information takes place at the same time at which it is sent. Our choice for synchronous communication is motivated by the fact that in the field of concurrency theory, asynchronous communication can be modelled in terms of synchronous communication [20]. In particular, asynchronous communication can be modelled in our agent framework by means of communication facilitators. This is illustrated in the next example.

Example 13 Sending a question $\mathsf{ask}(\varphi)$ along a channel c without waiting for its answer (to be received along a channel d) can be described by the following code:

$$c!\mathsf{ask}(\varphi) \cdot (S_1 \,\&\, (d?\varphi \cdot S_2 + d?\mathsf{untell}(\varphi) \cdot S_3)),$$

where S_1 represents the remaining activities of the agent and S_2 and S_3 represent the agent's subsequent responses to the receipt of the answers φ and $\mathsf{untell}(\varphi)$, respectively.

Secondly, the corresponding receipt of a question $\mathsf{ask}(\varphi)$ along a channel c, will be handled by the addressed agent's communication facilitator Fac which satisfies the following recursive equation:

$$Fac :- c?\mathsf{ask}(\varphi) \cdot ((e?\varphi \cdot d!\varphi) \,\&\, Fac),$$

where e denotes an internal channel connecting the facilitator with the agent. Note that consequently this facilitator in fact describes a bag of received requests (along channel c) for answering ψ.

Obviously, this basic form of asynchronous dialogue can be extended to more involved patterns of interaction.

As an additional example we consider the KQML speech acts `ask_one`, which is used to ask for one instantiation of the specified question. In the concurrency framework of ACP [3], value-passing can be modelled by synchronisation of actions and non-deterministic choice. Similarly, in our framework, we can model the generation and communication of solutions to constraints as described by the KQML speech acts like `ask_one` in the following way.

Example 14 Anticipated responses (to be received along channel c) to the KQML expression $\mathtt{ask\_one}(\varphi)$ can be modelled as follows:

$$\sum_{i \in I} c?\varphi(\theta_i) \cdot S_i,$$

where $\sum$ represents non-deterministic choice and the set $\{\theta_i \mid i \in I\}$ denotes the set of all suitable substitutions, $\varphi(\theta_i)$ denotes the application of the substitution θ_i to φ and S_i represents the corresponding subsequent reaction.

For instance, consider the question $\mathtt{ask\_one}(\mathit{price}(x, \mathit{item464}))$ to ask for the price of a particular item. Let us assume that prices can be any natural number between 0 and 1000. The reception of an answer to the question can then be described by:

$$\sum_{i \in [0..1000]} c?\mathit{price}(i, \mathit{item464}) \cdot S_i.$$

5 Conclusions and Further Reading

In this paper, we have considered the process algebra ACPL, which models the basics of agent communication. This algebra combines the information-processing aspects of CCP with a generalisation of the synchronous handshaking communication mechanism of CSP. The operational semantics of ACPL is given in terms of a transition system that consists of local and global transition rules. The local rules describe the operational behaviour of agents, like the local effects of communication actions. The global rules define the operational behaviour of multi-agent systems including the matching of communication actions. We have shown how ACPL provides a general basis to address the semantics of agent communication languages.

The basic algebra ACPL has been extended in several different directions. We conclude the paper by considering a number of these extensions.

Full Abstraction The operational semantics of ACPL describes the behaviour of a multi-agent system in terms of its computations. In general, however, we are not interested in all details of the behaviour that a particular system exhibits. That is, we want to reason about a multi-agent system at a higher level of abstraction, namely at a level that captures the aspects that are visible to an external observer. In [4], we consider the observable behaviour of a multi-agent system to be the final information stores as computed by the individual agents in a system. In order to describe this notion of observable behaviour in a compositional way, we have developed a form of denotational semantics, called *failure semantics*, which is shown to be a fully-abstract characterisation of the notion of observables. For instance, with respect to this observable behaviour, the following algebraic laws hold. If $\varphi \vdash \psi$ then:

$$\begin{aligned} c!\varphi\ &= c!\varphi + c!\psi \\ c?\psi &= c?\psi + c?\varphi \end{aligned}$$

The crucial observation here is that any communication action that for instance matches $c\ !\ p$ also matches $c\ !\ p \wedge q$. In general, we could say that sending a message includes sending all messages that contain less information. Similarly, any communication action that matches $c\ ?\ p \wedge q$ also matches $c\ ?\ p$. In other words, anticipating the receipt of a messages includes anticipating the receipt of all messages that contain more information.

Specification and verification 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, like for instance the conversation policy [16] 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. In [12], a compositional verification calculus for ACPL is defined. This calculus can be used to verify that a particular multi-agent system satisfies its specification. On the basis of this calculus it is possible to implement (semi-)automatic verification procedures. This is a subject of future research.

Translations In [11], we consider communicating agents that employ different vocabularies to represent information. In order to communicate some translations between these vocabularies need to generated. Instead of being defined in advance, we consider translations that are dynamically constructed during execution of the system. These translations are based both on the information that the agents exchange and the underlying ontologies that they employ. This yields a framework that can be used to study and analyse experiments as performed in the research on the origins of language, like for instance dialogue games in which the purpose of communication is to develop a mutual understanding of the agents' vocabularies [31]. This is also a subject of further research.

Agents and objects In [10], it is studied in which way concepts and techniques that have been developed in the object-oriented paradigm, can be adopted and adjusted for multi-agent systems. In particular, we study in which way the rendezvous communication mechanism of object-oriented programming can be generalised to structure the exchange of information between agents. A central concept of this chapter is the concept of a *question invocation* by analogy with the concept of a *method invocation* from object-oriented programming.

Groups Finally, we mention here some related work by De Vries *et al.* [33,34], where an abstract programming language for agent interaction is proposed, called GrAPL. This language contains constructs for coordinating group activity (group communication, formation, and collaboration). There are definite similarities between GrAPL and ACPL: both are given operational (process-algebraic) semantics, both employ the idea of a CSP-like synchronisation for communication between agents, and in both languages the communicated information is viewed as comprising constraints in a constraint system. The main difference is that in ACPL bilateral communication is used while in GrAPL this idea is extended to groups of agents that tell each other constraints about (parameters of) actions to be performed by them. In [33, Chapter 6] this idea is extended further to the communication of partial plans, i.e. orders on actions, as well as who is willing to do what, between agents in a group so that in effect the agents are able to perform distributive planning.

References

1. J.L. Austin. *How to do Things with Words*. Oxford University Press, Oxford, 1962. 114
2. J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: real time and discrete time. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 1999. 115
3. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984. 115, 118, 124
4. F.S. de Boer, R.M. van Eijk, W. van der Hoek, and J.-J.Ch. Meyer. Fully-abstract model for the exchange of information in multi-agent systems. *Theoretical Computer Science*, 290(3):1753–1773, 2003. To appear. 115, 125
5. F.S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'92*, pages 463–477, Cambridge, Massachusetts, 1992. MIT Press. 115

6. S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, 31:499–560, 1984. 115
7. E.W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. 114
8. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976. 114
9. 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. 115
10. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Generalised object-oriented concepts for inter-agent communication. In C. Castelfranchi and Y. Lesperance, editors, *Intelligent Agents VII, Proceedings of 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2000)*, volume 1986 of *Lecture Notes in Artificial Intelligence*, pages 260–274. Springer-Verlag, Heidelberg, 2001. 115, 126
11. 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. 115, 126
12. R.M. van Eijk, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. A verification framework for agent communication. *Journal of Autonomous Agents and Multi-Agent Systems.* To appear, 2003. 115, 125
13. 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. 113
14. FIPA. Foundation for intelligent physical agents. Communicative act library specification. http://www.fipa.org, 2000. 113
15. P. Gärdenfors. *Knowledge in flux: Modelling the dynamics of epistemic states*. Bradford books, MIT, Cambridge, 1988. 116
16. 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. 125
17. 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. 114
18. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland Publishing, Amsterdam, 1971. 116
19. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 115
20. H Jifeng, M.B. Josephs, and C.A.R. Hoare. A theory of synchrony and asynchrony. In *Proc. of the IFIP Working Conference on Programming Concepts and Methods*, pages 446–465, 1990. 124
21. Y. Labrou, T. Finin, and Y. Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, 14(2):45–52, 1999. 113
22. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980. 115
23. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999. 115
24. 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. 113

25. G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981. 115, 118
26. R. Reiter. On closed world data bases. In H. Gaillaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76, New York, 1978. Plemum Press. 123
27. V.A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Massachusetts, 1993. 115
28. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 333–352, 1991. 116
29. J.R. Searle. *Speech acts: An essay in the philosophy of language*. Cambridge University Press, Cambridge, 1969. 114
30. M.P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998. 113
31. L. Steels. The origins of ontologies and communication conventions in multi-agent systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(2):169–194, 1998. 126
32. B. Thomsen. A calculus of higher order communicating systems. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 143–153, 1989. 115
33. W. de Vries. *Agent Interaction: Approaches to Modelling, Abstraction, Programming and Verification of Multi-Agent Systems*. PhD thesis, Utrecht University, Mathematics and Computer Science, 2002. To Appear. 126, 126
34. W. de Vries, F.S. de Boer, K.V. Hindriks, W. der Hoek, and J.-J. Ch. Meyer. Programming language for coordinating group actions. In B. Dunin-Keplicz and E. Nawerecki, editors, *From Theory to Practice in Multi-Agent Systems*, volume 2296 of *Lecture Notes in Artificial Intelligence*, pages 313–321. Springer-Verlag, Heidelberg, 2002. 126