AAECC (2005) 16: 113–150
DOI 10.1007/s00200-005-0173-0

Frank S. de Boer · Wieke de Vries · John-Jules Ch. Meyer · Rogier M. van Eijk · Wiebe van der Hoek

Process algebra and constraint programming for modeling interactions in MAS

Received: 9 October 2003 / Revised: 25 June 2004 / Published online: 31 May 2005

Abstract We show how techniques from the realm of concurrent computation can be adapted for modeling the interactions of agents in multi-agent systems. In particular, we introduce a general process-algebraic approach to modeling multi-agent systems. Our approach consists of an integration of the process algebras of Communicating Sequential Processes (CSP) and Concurrent Constraint Programming (CCP) for modeling the communication, synchronization and coordination in multiagent systems, including FIPA-style communication primitives and a programming language for group actions in a multi-agent system.

1 Introduction

A very challenging new domain of applications in computer science concerns the modeling and programming of complex interactions in so-called multi-agent systems (MAS) [9,32,33]. Typically in these systems, pieces of software, called *agents* [34], are working together to perform a certain task, each displaying some autonomous and proactive behaviour, and acting together within a highly non-deterministic and unpredictable environment. (Potential) applications of such systems abound, ranging from e.g. workflow systems and information management systems to systems for electronic commerce and social simulation ([33]). The increasing complexity of such software requires new modeling and programming techniques which allow a formal description of agents at a high-level of abstraction corresponding with that of the application domain.

*also at CWI, Amsterdam

F. S. de Boer* · W. de Vries · J.-J. Ch. Meyer (✉) · R. M. van Eijk
Institute of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: jj@cs.uu.nl

W. van der Hoek
Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, United Kingdom

In our approach we focus on the information-processing aspects of multiagent systems. We view a multiagent system as set of information-processing entities (agents) that act in a particular environment and that communicate with each other through the exchange of information.

Each agent maintains a private store of information that it can query and update with new information. This new information results from direct observations in the environment (such as, for example, in an e-commerce setting, directly inspecting a vendor's assortment) or be the result of communication with other agents. In such communications, agents provide answers (like for instance,"Yes, it does") to questions they have been asked (e.g., "Does vendor X have item Y in its assortment?").

Actions can be performed in the world by an agent itself (like for instance, a vendor adding a particular item to its assortment), but there are also actions that require the joint execution of more than one single agent (e.g., a transaction between a buyer and a seller). In the latter case, agents need to coordinate their activities. That is, they need to synchonise with each other and communicate on the conditions of the joint action (e.g., the price of the item at hand, the delivery conditions and so on). There are situations in which agents even communicate on the identity of participants that should be involved in the joint action (like the exclusion of an unreliable agent in a multi-party transaction).

In this paper we advocate a *process-algebraic* approach (in the sense of [23]) to modeling multiagent systems, in particular the information-processing aspects of these systems. This involves describing multiagent systems by means of *processes*, i.e., terms in a formal (programming) language, together with a precise operational semantics in the form of a formal transition system. The modelling of a multi-agent system in terms of a process algebra forces us to lay bare the computational mechanisms that are essential to the information-processing in multi-agent systems. The approach thus allows us to abstract away from implementation details. Moreover, it enables us to give precise definitions of the operational semantics of the different mechanisms. So our language, like most process algebra's, allows to give abstract specifications of multi-agent systems, and this paper focuses on the design of such systems by a specification language that is rigorously founded by a formal operational semantics. Since we use the standard formalism of structural operational semantics we may also import and apply to multi-agent systems the various concepts that are defined and studied in concurrency theory, like equivalences based on bisimulation or refinement notions based on trace inclusion, but we will not pursue this issue further in this paper. Finally, we benefit from the modularity of the approach: we start with a basic framework on top of which more elaborate features and refinements can be built in possible further extensions.

In more detail, in our framework, an agent is modelled as a computational entity that operates with respect to a private store of information. One part of this store represents the beliefs of the agent about its environment (its belief base) and the other part of this store represent the conditions the agent has on joint actions. Both beliefs and conditions are modelled by so-called 'constraints'.

A constraint is an abstract piece of information that constrains the interpretation of variables. Example of constraints are $Number(x)$, $Item(y)$, $Price(x, y)$ and $x < 25$ which denote pieces of information expressing that x is a number, y is an item, x is the price of y and x is less than 25, respectively. When taken together

these constraints express the information that the price of a particular item is less than 25.

Additionally, the agents interact with each other via a synchronous communication mechanism, in which they either bilaterally exchange information in the form of a question/answer pair or multilaterally communicate with respect to the execution of a particular joint action. In the latter case, the constraints on the action that each of the involved participants has constructed for itself are accumulated, and this overall accumulation can be inspected by each of the participants in turn. If the individual pieces indeed add up to a consistent overall constraint the participants might decide to subsequently start with the execution of the joint action. If not, they might reconsider their individual constraints and repeat the procedure. The underlying *constraint solver* [28] which will be used to find consistent assignments of the variables satisfying the constraints produced by the negotiations, is not explicitly described in our approach but regarded as a parameter of it.

It is important to note that, as a language for coordinating negotiation processes, our proposal abstracts from the particular negotation policies followed by the participating agents. In general, the goal of coordination is to manage the interaction among concurrent processes or agents. Coordination abstracts away the details of computation. Coordination is relevant in the design, development, debugging, maintenance, and reuse of all concurrent systems. A number of software platforms and libraries are presently popular for easing the development of concurrent applications. Such systems, e.g., PVM, MPI, CORBA, etc., are sometimes called middleware. Coordination languages can be thought of as their linguistic counterpart [13,32,33].

This paper is organized as follows. In the next section we discuss our process-algebraic approach, in particular the (two) sources of inspiration that we have found in the literature of concurrency semantics that we will adapt and combine for our purposes. We give the general idea of using transition systems to provide a formal operational semantics to programming language constructs that we will use throughout the paper. In Section 3 we give a concise treatment of bilateral synchronous agent communication in the process algebra ACPL, preparing us for the next step. Then in Section 4 we turn to the main topic of this paper, viz. agent coordination, involving group communication, formation, negotiation and collaboration. To this end we introduce the process algebra GrAPL, of which the syntax is described in Section 5 and the semantics is investigated thoroughly in Section 6. In section 7 we provide two examples to further illustrate the mechanisms of the framwork. In section 8 we wrap up.

2 Our approach: blending CSP and CCP

The starting points of our approach are the well-known process algebras of Communicating Sequential Processes ([20]) and Concurrent Constraint Programming ([27]). These process algebras provide different models of the communication and synchronization in concurrent systems.

CSP models the interaction between concurrent processes in terms of a mechanism for the *synchronous* communication of values along *channels* which connect

the processes. These values are stored in variables which are local to the processes. On the other hand, constraint programming is based on the notion of computing with systems of partial information: in CCP processes interact via a global *store* which *constrains* the values of the global variables. This global store can be updated by so-called 'tell' operations which consist of adding a constraint and thereby constraining further the values of the global variables. The global store can also be queried by 'ask' operations which can be executed only if the current store is *strong enough* to entail a specified constraint. If this is not the case, then the process suspends (waiting for the store to accumulate more information by the contributions of the other processes). The execution of an ask itself leaves the store unchanged. Hence both the tell and ask actions are monotonic, in the sense that after their execution the store contains the same or more information. Therefore the store evolves monotonically during the computation, i.e. the set of possible values for the variables shrinks.

In the remainder of this paper, we will elaborate the further details of the agent communication mechanisms that we discussed in the introduction, using ideas and techniques from CSP and CCP. We will do this in terms of two process algebras: ACPL and GrAPL. In the language ACPL, we focus on the bilateral communcation mechanism of exchanging information through question/answer pairs. In the language GrAPL, we generalise this basic framework to a multilateral communication mechanism for the coordination of joint actions. We define the syntax of these algebras and provide a structural operational semantics in terms of a transition system.

To get a more concrete picture we elaborate on this a little further. Our general methodology for modeling multiagent systems is based on the use of constraint systems for representing the information state (e.g., its 'beliefs' [34]) of an agent. Such a constraint system also defines an entailment relation which thus captures the reasoning capabilities of an agent. Each agent has its own private store of constraints which represent its current beliefs. Agents can update and query their beliefs by the CCP operations of ask and tell. On the other hand, following CSP, agents interact only by communicating their beliefs synchronously along CSP-like channels. However, agent communication generalizes CSP-like communication of simple values because it involves a communication of *beliefs* (rather than just values) in the context of a dialogue: the sent constraint is interpreted as an answer to the query posed by the receiving agent. For this integration of CSP and CCP for agent communication we introduce a formal process algebra called ACPL.

As to the issue of agent coordination: in this paper we introduce a coordination language GrAPL for managing the interaction between negotiating agents. The basic idea underlying GrAPL is to model the process of negotiation in terms of communication and synchronization via a kind of global constraint store (effectively realized by a collection of local stores) which represents the result of the negotiation reached sofar. The act of introducing a new proposal (or bid) is modeled in GrAPL by a communication act which consists of adding a new proposal in the form of a constraint to the current global constraint store. Such a new proposal thus restricts the domain of possible values of the variables of the store which form the subject of negotiation. However, such an update requires an agreement among the agents involved in the negotiation. In GrAPL we model the basic mech-

anisms of accepting and refusing new proposals by means of a powerful dynamic synchronization mechanism of the agents involved in the negotation.

In GrAPL we again employ *synchronous* communication for the following important reason. Synchronous communication in the specification of multi-agent systems allows one to model at a high-level of *abstraction* the commitment and mutual agreement in the formation of groups. It allows to focus on the dynamics of group formation itself and to abstract from implementation details of the underlying protocols, which may, for instance, involve some central control such as a blackbord system.

2.1 Structural operational semantics: transition systems

In the sequel we will emphasize the (formal) semantics of the language constructs that we will introduce. To this end we employ the familiar SOS (Structural Operational Semantics)-style introduced by Plotkin [26,18], where a formal axiomatic transition system is used to give a precise definition of the semantics of language constructs at hand.

Here we give the basic idea. We start out from the abstract notion of a program configuration, which in applications will take a concrete form, but typically is a pair of a syntactic and a semantic entity (or a set of these), the former representing the program that (still) has to be executed and the latter representing a state of the computation comprising the semantic elements of concern (e.g. a program state assigning values to programming variables). Next, we consider transitions of configurations induced by program execution. The possible transitions are given by an axiomatic system. Axioms in this system take the form $c \longrightarrow c'$, indicating that a transition may take place from configuration c to c'. Rules have the form:

$$\frac{\begin{array}{c} c_1 \longrightarrow c_1' \\ c_2 \longrightarrow c_2' \\ \cdots \\ c_k \longrightarrow c_k' \end{array}}{c \longrightarrow c'}$$

Such a rule expresses that if the premises have been derived already also the conclusion transition may be derived. 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 transition is derivable if it is either an axiom or can be derived by a rule from previously derived transitions. In the sequel we will, in fact, use *labelled* transitions. In general, the additional label will record certain information about the specific nature of the transition.

3 Agent communication: ACPL

In [8] we have demonstrated how agent communication can be described in a process-algebraic fashion, in particular by employing a generalisation of the synchronous handshaking communication mechanism of Communicating Sequential

Processes (CSP) [20], integrating elements from the paradigm of Concurrent Constraint Programming (CCP) [27] into the approach. The latter enabled us to treat the exchange of *information*, i.e. constraints, rather than simple *values* as in the original CSP. This again forms the basis for the semantics of (FIPA-style) communication primitives that are typical for agents [10,11].

We illustrate the idea by looking at (a fragment of) the process algebra ACPL (Agent Communication Programming Language) introduced in [7,8] to investigate semantics of some of the basic constructs. (To keep things simple we have left out constructs such as local variable declarations, encapsulation operators, repetition, and recursive procedure calls.) In this section we furthermore suppress the definition of formal syntax as much as possible in order to concentrate on the main ideas. (For a more complete treatment, also dealing with other constructs, we refer to [8, 7,4].)

Suppose we are given a *constraint system* C, that 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 (lub) operation, and *true*, *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 (= lub) 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 or equal 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, denoted as "$\vdash$". For instance, we have $p \wedge (p \rightarrow q) \vdash q$.

In the context of this section we will extend a constraint system to a *belief system*, in order to use the system for dealing with the information ('beliefs') handled by agents. To this end we augment the constraint system by an update operator $\circ$: $\varphi \circ \psi$ stands for the update of the information φ by the information ψ according to some particular information updating procedure (studied in the realm of belief / theory revision / updating [12]). We assume a particular belief system $\mathcal{B}$ to be given.

We now proceed with presenting the elements of the agent communication language ACPL, along with their semantics.

In this context a (local) configuration takes the form $\langle S, \varphi \rangle$, where S is a (complex) program statement (representing the program still to be executed), which is either a basic action (see below) or a composition of these by action prefixing, parallel composition and nondeterministic choice, and φ is a constraint, representing the information available.

We assume a given set *Chan* of (unidirectional) communication channels, with typical element c. The basic actions we consider are the following:

- $c!\varphi$: The execution of an output action 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.

- $c?\varphi$: The execution of an input action, which consists of anticipating the receipt of the information φ along the channel c, also has to synchronise with a corresponding output action $c!\psi$, for some ψ with $\psi \vdash \varphi$.
- $?\varphi$: The execution of a basic action $?\varphi$ by an agent consists of checking whether the private store of the agent entails φ.
- $!\varphi$: The execution of $!\varphi$ consists of updating the belief base with φ.

A labelled (local) transition now takes 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 a basic action of the form $?\varphi$ or $!\varphi$, or l is of the form $c!\varphi$ or $c?\varphi$, in case of a communication step. We employ the symbol $\surd$ to denote successful termination, using identifications $a \cdot \surd \equiv a$ and $S \,\&\, \surd \equiv \surd \,\&\, S \equiv S$, where the operator $\&$ is introduced below.

To give semantics to the basic actions we can use the following axioms:

$$\begin{aligned}
&\langle ?\varphi, \psi \rangle \xrightarrow{\tau} \langle \surd, \psi \rangle \qquad \text{if } \psi \vdash \varphi \\
&\langle !\varphi, \psi \rangle \xrightarrow{\tau} \langle \surd, \psi \circ \varphi \rangle \\
&\langle c!\varphi, \psi \rangle \xrightarrow{c!\varphi} \langle \surd, \psi \rangle \\
&\langle c?\varphi, \psi \rangle \xrightarrow{c?\varphi} \langle \surd, \psi \rangle
\end{aligned}$$

By means of transition rules we can now proceed and give semantics to imperative programming constructs in (complex) statements such as prefixing $a \cdot S$ (a followed by S), (internal) parallelism $S_1 \,\&\, S_2$, and nondeterministic choice $S_1 + S_2$, respectively:

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

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

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

Defining an agent system as a parallel composition of (local) configurations, we can give semantics to these agent systems by means of labelled (global) transitions of the form $A \xrightarrow{l} A'$, where A and A' stand for agent systems that may be either local configurations or parallel compositions of agent (sub)systems.

An interleaving semantics (as in CSP and other process algebras) is then given by the following rule:

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

Here the operator $\parallel$ is used to denote the parallel composition of agent (sub)systems (which is to be distinguished from the operator $\&$ that is used to indicate parallel execution *within* an agent!).

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

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

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

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

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. This rule is a direct analogue of the synchronous communication in CSP, tailored to our setting of information exchange.

Here we end our brief exposition of ACPL comprising FIPA-style agent communication, meant to show how a combination of the process algebras CSP and CCS can be adapted to treat this kind of agent communication in a formal way. In the next section we will generalize from bilateral communication to group communication for the purpose of coordinating task execution in a multi-agent setting.

4 Agent coordination: GrAPL

In this section we show how we can treat agent coordination, and group communication, formation and collaboration in particular. A process-algebraic language for this purpose was introduced in [31,30]. Here we go into more depth, concentrating mainly on the semantical issues, not covered by the latter paper. The language is called GrAPL, which abbreviates Group Agent Programming Language. Groups in GrAPL are dynamic, and can be created at runtime. Coordination and cooperation, which are crucial notions in multi-agent systems, are modelled in GrAPL by means of formally defined primitives for dynamic group communication and synchronisation.

We first give an informal explanation of the main ideas underlying the new features of the programming language GrAPL. The basic idea of GrAPL is that agents synchronously communicate in order to dynamically form groups which

synchronously perform certain actions. In fact, a group is defined in terms of the agents which are involved in a synchronous communication. This is a dynamic notion, which cannot be determined statically. (So we do not consider 'pre-existing' groups that engage in communication!) The agents negotiate about *constraints* on these group actions. The constraints are logical formulas, prescribing properties of the action parameters, e.g., the time and place of a meeting, and the group of participants of the meeting. Subsequently, several agents can synchronously execute the action, obeying the constraints associated with the action.

The programming language GrAPL is again based on a blend of the paradigms of constraint-based reasoning and CSP. As to the former aspect: GrAPL incorporates ideas from constraint programming [27] to enable agents to produce and query constraints about the parameters of the group actions, and can be implemented on top of a *constraint solver* [28], which finds a consistent assignment of the action parameters satisfying the constraints of a group of agents.

As to the CSP aspect in GrAPL : like in ACPL, synchronous communication is used to communicate information. However, contrary to what is the case in ACPL, communication is not bilateral but multilateral. We opt for synchronous communication instead of asynchronous communication, because it allows us to model the *commitment* to the new constraints communicated in the negotiation process at a high level of abstraction. Moreover, synchronous communication implicitly involves a protocol which controls access to the negotiation process.

More specifically, the communication and coordination process which should lead to the synchronous execution of an action by a group of agents, distinguishes two phases. During the first phase, called the *negotiation phase*, groups of agents negotiate about the constraints they impose on a certain group action a by synchronously communicating their constraints on the action parameters. All actions have an implicit parameter which denotes the group involved. By means of this parameter agents may express their constraints on the composition of the group. Group communication updates the constraints of the participating agents on the action a to be the conjunction of the individually proposed constraints. Subsequently, this resulting formula constrains each execution of a for each agent that has participated in the communication, until the constraint on the action a is changed again. In this second phase, called the *execution phase*, a group of agents tries to synchronously execute a group action which was the subject of negotiation. If the actual parameters of all agents in the group are compatible, and the constraints of the agents allow these actual parameters, the action is executed by the group. Otherwise, group action execution fails. The constraints thus *monitor* the execution of the action.

By definition in our set-up, a computation step which describes a synchronous communication only involves agents communicating about the *same* action. The non-determinism in the semantics described in the sequel in fact abstracts from the scheduling of the group communications and the corresponding actions. So agents which want to communicate some information about an action which is not scheduled simply have to wait.

Finally in this informal introduction to our approach, we remark that agents may be involved in different groups. Although in each computation step only one group communication can take place, an agent can participate in different computation steps, which in general will involve different groups of communicating agents.

4.1 Language features

Before plunging into the formal details of the syntax and semantics of the programming language, we will give an intuitive sketch of the new features of the language. As stated before, the focus of this work is group formation, group communication and group action. First, agents synchronously communicate with each other in order to form groups that are committed to performing group actions together. The agents communicate about constraints on the action they might do together. Subsequently, several agents can synchronously execute the action, obeying the constraints associated with the action. Not all actions need to be constrained, but if a set of constraints is associated with an action, then action execution has to obey the constraints, which can specify demands on the parameters of the action and the group of agents participating in the action. In other words, the constraints monitor the execution. To facilitate this, there are three special statements in the programming language. The first two implement group communication (CommGroupAdd and CommGroupReset) and the third implements (group) action execution (simply $\mathsf{a}(t_1, \ldots, t_n)$, where a is an action and $t_1, \ldots, t_n$ are the actual parameters of the action). The 'CommGroup-type' actions may be viewed as multi-lateral generalisations of the bilateral communication primitives $c!\varphi$ and $c?\varphi$ from the previous section: by the synchronous execution of the latter two information is communicated from the one agent to the other along some channel, whereas the synchronous execution of the CommGroup-type actions by agents in a group results in the communication of information (viz. constraints pertaining to actions / tasks to be performed) across this group. However, regarding the aspect of *updating* as a result of the communication the present setting is more primitive than in that of the previous section: there we used an update operator $\circ$ which we did not specify but might involve some sophisticated form of belief revision ([12]). The communication due to the CommGroup-type action of the current section involves just a very primitive form of updating, viz. adding constraints ('expansion'). Since this may easily result in inconsistencies we also introduce the 'reset' variant (CommGroupReset) to enable the agents to start over coordinating / negotiating when these inconsistencies occur.

Actions in GrAPL are parameterised. Each action has a certain arity, which is the number of parameters it needs. An example of an action of arity two is $\mathsf{PlayGame}(v_1, v_2)$, where the first parameter v_1 is the particular game to be played and v_2 is the starting time of the game. Apart from these explicit parameters, each action has one implicit parameter, which is the group of agents which execute the action. This group is always denoted by the special variable g.

Earlier, we mentioned the two phases of the coordination process for a group action. In the first phase, the negotiation phase, the agents interested in establishing conditions for execution of a group action a perform CommGroupAdd and CommGroupReset statements, thereby exchanging proposals for constraints on the action a. As the name of the statements already suggests, CommGroup statements (both kinds) are synchronously performed by a group of agents. The constraints the agents communicate about in GrAPL are constraints on the explicit and implicit parameters of a. So, agents talk about the details of the action and about the group which is going to perform the action.

Each agent has a private, local constraint store, where it keeps the present constraint on each action. We could also have opted for a global constraint store for each action, in which agents communicating about the action write their constraints. Local constraint stores have several advantages over global stores:

- They fit better with the concept of autonomous agents. The intuitive idea of an agent is that it reasons about its own motivations and the circumstances in its environment, and then decides to do certain actions, by itself or in cooperation with other agents. All information the agent needs for this is stored locally (in its mental state) or is received from the environment (through observation and communication).
- They allow multiple groups of agents to discuss and execute the same action. When two disjoint groups of agents negotiate on an action a, then the local constraint stores of the agents in each group contain the constraints which the agents in this group communicated, and no constraints coming from the agents of the other group.
- The local constraint store of an agent represents its own view on the negotiation about a certain action. During each synchronous CommGroup meeting, only communicating agents update their local stores. So, in order to know what's going on, it is essential to be present in meetings, like in many real-world organisations.

Local constraint stores thus match the intuitions associated with agents. Initially, the local constraint stores contain the constraint $\top$, which denotes the logical formula that is always true. So, initially no agent has any demand on any action.

CommGroupAdd and CommGroupReset both take two arguments, a constraint φ and an action a. The difference between CommGroupAdd (φ, a) and CommGroupReset (φ, a) is that in the first case the agent adds φ to its present constraint on a, and then proposes the conjunction, while in the second case, the agent forgets about its current constraint on a and simply proposes φ. When a group of agents is communicating about a group action, some agents in the group may perform a CommGroupAdd action and others may perform a CommGroupReset action. All agents in the group have to agree upon the focus of the communication, that is, the action. Each agent brings its own set of constraints, demanding execution of the action to take place in a certain manner. The composition of the group of communicators must satisfy the demands of each agent. If this is not so, group communication fails. Each successful synchronous combination of group communication actions updates the constraints of the agents communicating on the action discussed to be the conjunction of the proposals, as this is the weakest constraint implying all individual constraints.

After one or more synchronous executions of CommGroupAdd and CommGroupReset statements, the second phase, the execution phase, takes place. (In our approach, we abstract again from the precise way how this phase transition happens. One may think of an arbiter agent (or a timing device) deciding this or of a kind of group process, for which it seems to be a necessary precondition that all agents involved know that sufficient and consistent information about the parameters of an action has been accumulated. One way for an agent to obtain this knowledge is simply to try to execute the action. . .) In the execution phase, a group of agents tries to execute the group action which was the subject of negotiation. Each agent ι chooses actual parameters for the action a, named $t_1^\iota, \ldots, t_n^\iota$

if the action takes n parameters. As the parameters can contain free variables, certain agents in the group can communicate actual parameters to other agents. This implicit communication which takes place as a side-effect of action execution is called *execution-time communication*. We will show how this works in the example in the next subsection. If the actual parameters of all agents in the group are compatible, and the constraints of the agents allow these actual parameters, the action is executed by the group. So, the second 'new' element in our language (next to group communication) is group action execution. As can be seen above and in the example below, the syntax of action execution is conventional; it's the semantics that is different. Whenever an action $\mathsf{a}(t_1, \ldots, t_n)$ is to be executed by an agent, the abstract interpreter of our language checks whether there is a constraint bound to this action. This is done for all agents about to perform the action. There must be no conflicts in the actual parameters and all constraints in constraint stores of agents participating have to be satisfied. If so, the action can be done. Otherwise, group action execution fails.

Each agent has its own belief base, which it uses to store information. The beliefs of an agent can influence the decision making of the agent, for example about the constraints the agent imposes on some action. In this paper, we reserve the term *constraint store* for sets of formulas which describe demands on parameters of certain actions. In constraint programming languages, like for example CCP ([27]), this term covers all information stores. Our use of the term is different; in our view, belief bases need not be constraint stores. Belief bases can be constraints, describing the partial information of the agents about certain world features (represented by variables), or they can be sets of closed formulas, describing the information which the agents hold true of the world state. Because we choose for the second option in this paper (which isn't a principled choice in any way), we only refer to constraints on actions with the phrase 'constraint store'.

4.2 Example

As an example of the constructs of GrAPL, we return to the game playing action. Suppose we have agents with names Gabriël, Jan-Ybo, Martha and Jantina. It is a boring Sunday afternoon, and the agents are talking about playing a game. Initially, all four agents have the empty constraint ($\top$) on PlayGame in their local store. They enter the negotiation phase. The four agents synchronously perform a CommGroupReset(φ_{id}, PlayGame) action. Here, φ_{id} is the constraint of the agent named id, as shown below. Recall that the action PlayGame, introduced above, needs two parameters, which we refer to as v_1 and v_2. As will be explained later, in constraints all agents always use formal parameter v_i to refer to the ith parameter of some action. Each of the agents proposes a different constraint:

φ_{Jantina} equals $v_1 = \text{Rummikub} \wedge v_2 < 16.00$. So, the only game Jantina is prepared to play is Rummikub, and she wants to start before four o'clock. Jantina has no demands on the composition of the group playing the game. $\varphi_{\text{Gabriël}}$ is this constraint: $v_1 = \text{Rummikub} \rightarrow \text{Martha} \in g$. So, Gabriël is willing to play any game at any time, but if the game is to be Rummikub, he wants Martha to play along. $\varphi_{\text{Jan-Ybo}}$ is the constraint $v_2 > 14.30$. His belief base contains the information that his favourite TV-show is on from 13.30 till 14.30 and that he has to do his homework up to 13.30. So, Jan-Ybo has other things on his mind until half past

two. Only at some later time he is prepared to play a game. $\varphi_{\mathsf{Martha}} = \top$. Martha is a very easy agent. She doesn't have any constraints.

These four constraints are consistent; the conjunction of the constraints is equivalent to $v_1 = \mathsf{Rummikub} \wedge 14.30 < v_2 < 16.00 \wedge \mathsf{Martha} \in g$. Consequently, the four individual local constraints are replaced by the constraint just mentioned.

Now, after the negotiation phase, which in this case consists of only one synchronous group communication action, the four agents can actually play the game. They enter the execution phase. Each of the four agents executes a PlayGame action. The actual parameters for this action may differ, as long as they can be unified and they obey the constraints agreed upon. The agents can test their constraint stores for Rummikub to find out which demands the other agents communicated, and to choose appropriate actual parameters. We don't go into this now.

Some actual action parameters can be free local variables. This way, execution time communication about action parameters is possible. This form of communication is a side-effect of action execution. Suppose this is what the four agents try to do:

- Jantina: PlayGame(Rummikub, t_1). So, Jantina doesn't provide a value for the starting time of the game. Instead, she uses a free local variable, denoting that she doesn't want to be the one to choose the definite time, even though she initiated the constraint that the game has to start before 16.00.
- Gabriël: PlayGame(p_1, t_2). Gabriël is open to anything (as long as the constraints are satisfied). He simply uses two free local variables as actual parameters.
- Jan-Ybo: PlayGame(Rummikub, 15.00). So, Jan-Ybo sets the time at which the game will start.
- Martha: PlayGame(p_2, t_3). Martha still is a very easy agent. She is prepared to adjust herself fully to the other agents, as long as the constraints communicated to her in the first phase are respected.

An attempt to synchronously execute this group action will succeed, as the concrete parameters of the four agents can be unified, and the resulting action parameters satisfy the constraints. Jantina and Jan-Ybo agree on the first parameter of the action, and they communicate Rummikub to the other agents. The second parameter in picked by Jan-Ybo to be 15.00, and implicitly communicated to the others. These two actual parameters satisfy the demands made on the formal parameters v_1 and v_2 in the (now identical) constraint stores of the agents. Also, Martha is part of the group playing the game, and so the demand on the group composition (Martha $\in g$) also holds.

But in case Jantina would have demanded the time 14.45 as the second parameter, the group action would have failed. In this case, there are two agents in the group who don't agree on a parameter and execution time communication fails. A group action can also fail when the agents agree on the actual parameters. This happens when the unified parameters resulting from execution time communication conflict with the constraints of the participants. For example, Jan-Ybo can choose the second actual parameter of PlayGame to be 17.00, and tell this to the other agents using execution time communication. Agent Jan-Ybo then chooses to ignore the constraint on PlayGame that the agents agreed upon in the negotiation phase. As the other agents use a free local variable for the second parameter, execution time communication succeeds, but the action can't succeed, as the time 17.00 doesn't obey the constraint on PlayGame. When a group action fails, the agents can wait

until the action can be successfully executed, or they can choose to continue doing other actions from their programs.

The picture sketched above just gives a general impression of the features of the programming language. There are many subtle issues and different interesting options for the precise meaning of group communication and coordination in this language. We will elaborate on this later on in this paper.

5 GrAPL: syntax

5.1 Basic definitions

The programming language we are about to define is based on the principles of constraint programming. The sets underlying GrAPL are:

- $\mathcal{A}$ = the set of atomic actions. Typical elements are a and b. Each action has an arity, which is the number of parameters it takes.
- $\mathcal{I}$ = the set of agent identities. Typical elements are ι and κ.
- $\mathcal{V}$ = the set of variables. There are two kinds of variables; $\mathcal{V} = \mathcal{LV} \cup \mathcal{GV}$. Here, $\mathcal{LV}$ are local variables. Each agent has its own local variables, so the set of local variables is the disjoint union of sets of agent-specific local variables. $\mathcal{GV}$ contains the global (system) variables, defined as $\mathcal{GV} = \{v_k | k \in \mathbb{N}\} \cup \{g\}$.
- $\mathcal{D}$ = the value domain. Elements of this set are used as constants and can be bound to variables. Two subsets of $\mathcal{D}$ are $\mathcal{I}$ and $\wp(\mathcal{I})$.

Local variables are used for processing information only relevant to one agent, such as testing the local constraint store and specifying formulas to be inserted into the belief base. For this last purpose, the programmer must make sure that free local variables are instantiated with ground values at the time the insertion is executed, as the belief base is a closed formula. Local variables are also used to specify actual action parameters, when an agent doesn't care about the precise value of certain formal parameters. Above, we attributed disjoint sets of local variables to all agents. These disjoint sets make it impossible for two agents in a system to use the same local variable name. This prevents name clashes, which could occur during synchronised action. In group action execution, the local variables of a number of agents meet when there is execution-time communication.

So, for example, when two agents ag_1 and ag_2 attempt a group action a, having three parameters, when ag_1's program contains the statement $\mathsf{a}(x, 4, 58.3)$ and ag_2 has the statement $\mathsf{a}(1, x, 58.3)$, then there is no global substitution unifying the actual action, whereas it is clear that the intended group action, viz. $\mathsf{a}(1, 4, 58.3)$, can be executed successfully. In order to prevent these name clashes, we define the sets of local variables in such a way that each agent uses different local variables.

$\mathcal{GV}$ is the set of all variables v_i and g. We use these variables to specify the formal parameters of actions. We adopt the practice to refer to these parameters in a uniform manner. We call the formal parameters of an action a $v_1, v_2, \ldots, v_k$ if the arity of a is k. The implicit formal parameter for the group performing the action is g. In GrAPL-programs, these formal action parameters occur in constraints on actions and in formulas tested on the constraint stores. All agents use the same set of global variables to refer to formal parameters of the whole range of actions. Constraints are always specified relative to an action, so the global variables have

an unambiguous meaning. Thus, the set $\mathcal{GV}$ exactly contains the variables needed to work with constraints. To avoid confusion, global variables are only allowed in conjunction with constraints. Also, global variables are never bound to values. Even if $v_1 = 7$ is a demand on an action a, v_1 is not bound to 7; the constraint just means: "The first parameters of a must be 7", and not that the variable v_1 *always* has the value 7. We do not generate bindings to global variables because they are used in constraints on different actions, and it is undesirable that a constraint like $v_1 = 7$ on *one* action demands the first formal parameter of *every* action to be 7. Because global variables are not bound to values, there can be no value clashes between agents involving global variables. We chose to introduce special global constraint variables because it simplifies communicating about action parameters by groups of agents. Each agent uses the convention of referring to the ith parameter by v_i. So, no unification is necessary and programs become more clear.

GrAPL makes use of a multi-sorted predicate logical language $\mathcal{L}$. Each agent possesses a *belief base*; this contains closed formulas (no free variables) from $\mathcal{L}$. The constraints on actions are also formulas from $\mathcal{L}$, prescribing properties of action parameters. Each agent locally stores the present constraint for each action. More precisely, $\mathcal{L}$ is a multi-sorted predicate logic. The set of variables of the logical language is $\mathcal{V}$ and the set of constants is $\mathcal{D}$. The logic includes set theoretic predicates and functions, such as $\in$, $\subseteq$, $\cup$ and $\cap$, to express properties of the composition of groups of agents, as well as predicates and functions to describe properties of action parameters. We use φ and ψ to denote arbitrary formulas from $\mathcal{L}$ and $\top$ and $\bot$ to denote the formulas that are always true and false, respectively. We denote the set of free variables in an expression, term, formula, program, or other syntactic form w by $\mathit{free}(w)$ and the set of all variables by $\mathit{var}(w)$. We assume the logic $\mathcal{L}$ is equipped with an entailment relation, denoted by $\models$.

As an example of the use of $\mathcal{L}$ for formulating constraints, we give some constraints on the action $\mathsf{MoveObject}(v_1, v_2, v_3)$. Here, the first formal parameter is the object to be moved, the second parameter is the original location of the object and the third parameter is the location to which the object has to be moved. A very simple constraint is: $v_1 = \mathsf{table}$. If an agent has this constraint on action $\mathsf{MoveObject}(v_1, v_2, v_3)$, then it is only willing to move the table; any attempt of this agent to move something else will fail. If the predicate logic contains a function distance which takes two locations and yields the distance between these two locations, then another simple constraint is $\mathsf{distance}(v_2, v_3) < 10$. An agent having this constraint associated with the MoveObject action, is not prepared to move something over a distance which is 10 or greater. A last example of a simple constraint is $\mathsf{James} \in g$. This means that the agent having this constraint is only prepared to move things when James is part of the group performing the MoveObject action.

By using logical operators, more complex constraints are obtained. An example is the constraint $\mathsf{distance}(v_2, v_3) \geq 20 \rightarrow (v_2 = \mathsf{Utrecht} \wedge \mathsf{Max} \notin g) \vee \#(g) > 5$, which states that when the distance an object has to moved over is 20 or more, the agent having this constraint only agrees to help if there are at least five agents cooperating or if the moving starts in Utrecht and the agent doesn't have to cooperate with Max.

If a certain constraint is associated with an action a, then this constraint monitors future executions of a. Agents can repeatedly communicate with each other, using $\mathsf{CommGroupAdd}(\varphi, \mathsf{a})$ and $\mathsf{CommGroupReset}(\varphi, \mathsf{a})$. By doing this, the agents

define the parameter space of the actual parameters of a. When agents perform CommGroupReset statements, then their local constraint stores are re-initialised with fresh constraints, and when agents add constraints using CommGroupAdd, they narrow down the possible values for the formal parameters of the action.

Actions are primitive notions; their meaning and effects are laid down in semantic functions. Nevertheless the agent program can influence the meaning of actions, because agents can constrain the set of permissible actual parameters by group communication.

5.2 Programs

We denote the set of agent programs by $\mathcal{P}$. In order to define this set, we first define the set of *basic statements* $\mathcal{S}$.

Definition 1 (Basic statements)
The set $\mathcal{S}$ of basic statements is the smallest set containing:

- skip
- $?\varphi$, where $\varphi \in \mathcal{L}$ and $var(\varphi) \cap \mathcal{GV} = \emptyset$.
- $?(\varphi, \mathsf{a})$, where $\varphi \in \mathcal{L}$ and $\mathsf{a} \in \mathcal{A}$.
- $!\varphi$, where $\varphi \in \mathcal{L}$ and $var(\varphi) \cap \mathcal{GV} = \emptyset$.
- CommGroupAdd(φ, a), where $\mathsf{a} \in \mathcal{A}$ and $\varphi \in \mathcal{L}$.
- CommGroupReset(φ, a), where $\mathsf{a} \in \mathcal{A}$ and $\varphi \in \mathcal{L}$.
- $\mathsf{a}(t_1, \ldots, t_k)$, where $\mathsf{a} \in \mathcal{A}$, the arity of a is k and all t_i are terms of $\mathcal{L}$, such that for all $i \in \{1, \ldots, k\} : var(t_i) \cap \mathcal{GV} = \emptyset$.

The language includes a statement for doing nothing, skip.

There are two kinds of tests, namely tests of the belief base (simply denoted $?\varphi$) and tests of the constraint bound to an action (denoted by $?(\varphi, \mathsf{a})$). These tests check whether the formula φ is logically entailed by the belief base or the current constraint on a, respectively. Both kinds of tests can yield bindings of values to variables, but these variables never are global variables. For tests of the belief base, this is achieved syntactically, by forbidding global variables in the formula tested. Because global variables are used by all agents to refer to the formal parameters of all actions, it would be unpractical to generate bindings to global variables. Besides the practical reason of making no bindings to global variables, there also is a conceptual reason for excluding global variables from some statements. This is that we introduced global variables specifically for constraint handling. Therefore, they are forbidden in formulas tested on and inserted into the belief base, and also in actual action parameters. This way, we maintain a clear separation between global and local processing, which adds clarity and elegance.

In tests of actions, we do allow global variables, as the constraint on the action can contain global variables. For example, suppose an agent has the constraint $v_1 \leq 10$ on action a, meaning that the first parameter of a must not be larger than 10. The test $?(v_1 = 10, \mathsf{a})$ tests whether the constraint on a implies that the first parameter of a is 10. As this is not a logical consequence of $v_1 \leq 10$, the test fails. In case the current constraint on a would have been $v_1 = 10$, then the test succeeds. The semantics of GrAPL is defined such that this successful test doesn't result in a binding of the value 10 to the global variable v_1. In case we would

perform $?(v_1 = x, \mathsf{a})$, where x is a local variable, and the constraint on a implies that $v_1 = 10$, then only the local variable x is bound to 10.

Roughly speaking, the statement $!\varphi$ adds the information φ to the belief base. (Actually, we will see that in our semantics this is done in a more refined way, involving some kind of belief revision.) As mentioned above, we want global variables only to be used in conjunction with constraint stores, so we forbid them to occur in new belief base formulas. Another issue is that free local variables are not allowed in new belief formulas, as the belief base has to be a set of *closed* formulas. But we won't demand that $free(\varphi) = \emptyset$, because this would mean that each belief inserted into the belief base must already be completely specified at compile time. So, we allow free local variables in the new belief formula φ, but we demand that each free variable is *guarded* by a preceding test or action execution, which yields a value for this variable. (Note that action execution can generate bindings, through execution time communication.) Here is a simple example to make matters clear:

Example 1 (Guarded statements)

$$?(Book(x) \wedge Desirable(x));$$
$$\mathsf{GetPrice(x,p)};$$
$$!(Price(x) = p)$$

In this peculiar program, the agent inspects his beliefs for a desirable book, goes to check its price and then adds the price found to its beliefs, regarding that particular book. So, the execution of the !-statement is guarded.

The most novel statements of the programming language are CommGroupAdd(φ, a) and CommGroupReset(φ, a). Here, a is the action the agent communicates about and φ is a constraint stating demands of the agent on future executions of the action a. Using these statements, agents synchronously communicate about the details of the action and about the group which is going to perform the action. Each agent in a group of communicators executes either a CommGroupAdd-statement or a CommGroupReset-statement. Arbitrary combinations of these two statements are allowed. Group communication succeeds if every agent in the group of communicators approves of the presence of all communicators. Thus, group communication posits demands on the group of agents that communicates as well as on the group of agents that executes the action later on. The global variable g refers to both these groups.

If an agent executes CommGroupAdd(φ, a), then it proposes its previously accumulated constraint on action a strengthened with φ. If an agent executes CommGroupReset(φ, a), then it erases its present constraint on a and offers φ as a fresh proposal. In both cases, the resulting constraint on a will be the conjunction of the proposals of all communicators. The local bindings of a are updated accordingly. If the agents disagree, the resulting constraint will be $\bot$. We allow this because $\bot$ in a constraint store indicates that group communication has failed, and agents can test their constraint stores to find out whether this is the case. Subsequently, the resulting formula constrains each execution of a for each agent that has participated in the group communication, until the constraint on the action a is changed again. As the constraints are local and communication is synchronous, it is impossible for one agent to alter the constraints of another agent, without communicating with the other agent.

The syntax allows free local variables in CommGroupAdd statements and CommGroupReset statements. For these variables, we make the same restriction as we did for the free local variables in !-statements; they must be guarded, as a constraint containing a free local variable doesn't buy you much. When the free local variables in new constraints are guarded, they act as place-holders, lying in the scope of a binding statement (tests or action executions). This implies that at runtime, the local variables are instantiated with ground terms.

The last basic statement is action execution, denoted by $\mathsf{a}(t_1, \ldots, t_k)$ (sometimes abbreviated to $\mathsf{a}(\bar{t})$). We use this statement both for individual action and group action. The constraints associated with the action in the local states of agents trying to perform the action determine how many agents are needed, and sometimes make demands on their identities. If a group of agents (possibly consisting of only one member) tries to synchronously execute an action, the constraints of the agents on this action have to be consistent with each other and the actual parameters (the terms $t_1, \ldots, t_k$) and the group composition have to satisfy all constraints. In implementations of GrAPL, a constraint solver has to be plugged in to check this.

Another aspect of action execution is *execution time communication*. If one or more agents use a free variable in an actual parameter, and at least one agent specifies a definite value for this parameter, then the last agent *communicates* the value to the other agent(s). This form of communication generates bindings to the free variables used by the listening agents.

Example 2 (Jogging agents) Two agents, James and Clare, arrange to go jogging. They discuss and subsequently execute $\mathsf{Jog}(v_1, v_2)$, where v_1 and v_2 are the formal parameters of the action, denoting the starting time and the distance to be jogged, respectively. Each agent has a constraint it wants to impose on the parameters of Jog. In these constraints, the agents use the formal parameters v_1 and v_2 to refer to the explicit parameters of the action. Each action also has one implicit parameter, denoted by g, which is the group composition. These are the constraints of Clare and James:

$$\begin{aligned} \text{James: } \varphi &: v_1 > 19.00 \wedge (v_2 = 7 \vee v_2 = 8) \wedge \mathsf{Clare} \in g \\ \text{Clare: } \psi &: v_1 < 20.00 \wedge (v_2 = 8 \vee v_2 = 9) \wedge \\ &\quad (v_1 > 19.00 \rightarrow \mathsf{James} \in g) \end{aligned}$$

So, James wants to start jogging after 19.00 o'clock, he wants to run 7 or 8 km, and he wants Clare to join him. Clare on the other hand only wants James to jog with her when she leaves after 19.00 o'clock, she wants to start before 20.00 o'clock, and she wants to run 8 or 9 km.

They synchronously communicate:

$$\begin{aligned} &\text{James: } \mathsf{CommGroupReset}(\varphi, \mathsf{Jog}) \\ &\text{Clare: } \mathsf{CommGroupReset}(\psi, \mathsf{Jog}) \end{aligned}$$

The result of this synchronous communication is a new constraint, which holds for future executions of Jog of both agents:

$$19.00 < v_1 < 20.00 \wedge v_2 = 8 \wedge \mathsf{James} \in g \wedge \mathsf{Clare} \in g$$

Next, the agents synchronously execute:

$$\begin{aligned} &\text{James: } \mathsf{Jog}(19.30, 8) \\ &\text{Clare: } \mathsf{Jog}(x, 8) \end{aligned}$$

Note that there is execution-time communication here. James communicates the time 19.30 to Clare; Clare uses a free variable as first actual parameter, thereby indicating she is expecting James to pick the definite time. The constraint solver checks whether the actual parameters satisfy the constraints of James and Clare. This is the case, so the action is successful. In case James had performed Jog(y, 8) instead of Jog(19.30, 8), there would have been multiple possibilities for the first parameter. We come back to this issue later.

Having defined the set $\mathcal{S}$ of basic statements, we now define the programs of GrAPL.

Definition 2 (Agent programs) The set $\mathcal{P}$ of valid single-agent programs is the smallest set containing the following programs:

- α, where $\alpha \in \mathcal{S}$.
- if φ then π_1 else π_2, where $\varphi \in \mathcal{L}$, $var(\varphi) \cap \mathcal{GV} = \emptyset$ and $\pi_1, \pi_2 \in \mathcal{P}$.
- if φ for a then $\pi_1$else π_2, where $\varphi \in \mathcal{L}$, $\mathsf{a} \in \mathcal{A}$ and $\pi_1, \pi_2 \in \mathcal{P}$.
- $\pi_1; \pi_2$, where $\pi_1, \pi_2 \in \mathcal{P}$.
- $\pi_1 + \pi_2$, where $\pi_1, \pi_2 \in \mathcal{P}$.

We defined programs for single agents here. A multi-agent system simply is a set of single agent programs. These will be executed in parallel.

More complex programs can be formed using the if–then–else constructs, sequential composition and non-deterministic choice. The composed statement if φ then π_1 else π_2 first checks whether φ can be inferred from the belief base of the agent. If this is the case, π_1 is executed, and if not, π_2. The statement if φ for a then π_1 elseπ_2 is similar, except that this statement tests the constraint bound to the action a. Inclusion of these statements is useful, because it enables testing whether something *can't* be inferred, which is not possible with the test statements $?\varphi$ and $?(\varphi, \mathsf{a})$. In particular, we can use the statement if $\bot$ for a then π_1 else π_2, which checks whether the constraint on a has become inconsistent (because of group communication), and chooses an appropriate course of action. Statements like these allows the programmer to explicitly encode backtracking mechanisms in negotiation.

6 GrAPL: semantics

6.1 Basic definitions

To define the semantics of an agent system in GrAPL — consisting of a number of agent programs in parallel — we first have to provide some definitions.

First, we have to define the nature of agent configurations.

Each agent has a local configuration, which is a quadruple $\langle \mu, \delta, \iota, \pi \rangle$. The first element of the configuration is μ, which stores the constraints bound to actions. These bindings are local to the agent. This way, it is possible for two agents to have different constraints for some action. This happens when the agents are part of separate groups which have communicated about the action independently. The constraints contain no free local variables. The function μ is total, as each action is initially bound to $\top$ (no constraints). When the constraint on a certain action a is updated, we use the notation $\mu[\psi/\mathsf{a}]$, which denotes the function which is equal to μ except for the constraint on a, which has become ψ. The second configuration

element is the belief base of the agent, denoted by δ. We don't allow free variables in the belief base. Finally, $\iota \in \mathcal{I}$ is the identity of the agent and π is the agent program fragment still to be executed.

Definition 3 (Agent configuration)
A *local agent configuration* A_ι is a quadruple $\langle \mu, \delta, \iota, \pi \rangle$. Here,

- $\mu : \mathcal{A} \to \mathcal{L}$ where for all $\mathsf{a} \in \mathcal{A} : \mathit{free}(\mu(\mathsf{a})) \cap \mathcal{LV} = \emptyset$.
- $\delta \subseteq \mathcal{L}, \mathit{free}(\delta) = \emptyset$, and $\mathit{var}(\delta) \cap \mathcal{GV} = \emptyset$.
- $\iota \in \mathcal{I}$
- $\pi \in \mathcal{P}$

A *multi-agent system* consists of a number of agents executing in parallel. A *global system configuration* simply is a set of local agent configurations of all agents present in the system. The set of agents present in a system is fixed. So, we just as well assume that $\mathcal{I}$ is exactly the set of all agents in the system. Then, we have:

Definition 4 (System configuration) A *global system configuration* is a set $\{A_\iota | \iota \in \mathcal{I}\}$ of local agent configurations.

As seen above, we use μ to store the bindings to actions. But local variables can also be bound to values, for example when the belief base is tested. In our semantics we use *ground substitutions* to implement this. A substitution is ground if it binds variables to terms without variables in them.

Definition 5 (Substitution)

- A *substitution* θ is a finite set of pairs (also called *bindings*) of the form $x_i := t_i$, where $x_i \in \mathcal{LV}$ and t_i is a term of $\mathcal{L}$, $x_i \neq x_j$ for every $i \neq j$, and $x_i \notin \mathit{free}(t_j)$ for every i and j.
- A *ground* substitution θ is a substitution such that for every pair $x := t \in \theta$ the term t is ground, i.e. $\mathit{free}(t) = \emptyset$.
- The *domain of* a substitution θ, denoted by $\mathit{dom}(\theta)$, is the set of variables x for which θ contains a pair $x := t$.

By definition, substitutions can only bind local variables. Global variables can be used as formal parameters for different actions and in many constraint stores. Creating a binding to a constraint variable thus could potentially influence the meaning of many constraint stores, and so we don't allow it.

We define application of a substitution only informally, as it is a well-known notion and a complete formal definition would involve a lot of notational clutter.

Definition 6 (Application of a substitution) Let e be any syntactic expression, be it from $\mathcal{L}$ or $\mathcal{P}$, and let θ be a substitution. Then $e\theta$ denotes the expression where all free variables x in e for which $x := t \in \theta$ are simultaneously replaced by t.

We sometimes use substitutions to *unify* different sets of terms. We need some definitions for the notions of unifier and most general unifier.

Definition 7 (Unifiers and most general unifiers)

- Let θ and η be two substitutions. Then, the *composition* $\theta\eta$ is the substitution $\{(x := (x\theta)\eta) | x \in dom(\theta) \text{ or } x \in dom(\eta)\}$.
- Let$\{e_i | i = 1, \ldots, n\}$ be a set of expressions. A substitution θ is a *unifier* of these expressions if $\forall i, j \in \{1, \ldots, n\} : e_i\theta = e_j\theta$.
- Let $\{e_i | i = 1, \ldots, n\}$ be a set of expressions. A substitution θ is a *most general unifier* of these expressions if θ is a unifier of $\{e_i | i = 1, \ldots, n\}$ and for all other unifiers ζ it holds that $\zeta = \theta\eta$, for some substitution η.

So, the application of the composed substitution $\theta\eta$ means that first θ is applied, and then η. A unifier of a set of expressions is a substitution such that the expressions are all equal after application of the substitution, and a most general unifier is a unifier that keeps the expressions as general as possible.

The semantics of GrAPL we will give in the next two subsections is defined on two different levels, the *local agent level* and the *global system level*, like in the case of ACPL. Each agent can perform social actions (group communication and action execution) and individual actions (the other statements). Individual actions don't depend on or influence the other agents. The meaning of these actions can be defined locally. The transitions of these actions are labelled with the symbol τ. At the global system level, these actions simply are interleaved with the actions of other agents. The outcome of social actions depends on the behaviour of the other agents. The meaning of these actions can only be defined at the global system level. Nevertheless, at the local level, a dummy local transition step is generated for social actions, labelled with the action details necessary to define the semantics of the social action globally.

6.2 Local semantics

In general, a local transition looks like $\langle \mu, \delta, \iota, \pi \rangle \xrightarrow{l}_{\theta} \langle \mu', \delta', \iota, \pi' \rangle$. We use *labelled* transitions, because sometimes information present in the local level is needed to synchronously execute certain statements at the global level. We use the label τ for marking internal individual agent steps, resulting from local reasoning. When the statement locally executed is group communication or group action execution other labels are used to represent information necessary for the global semantics, as explained later on. The transition arrow is subscripted with a substitution, which contains bindings created by tests or communication that have to be passed on to the rest of the agent's program.

First, we give the transition rules for the basic statements. These are followed by the transition rules for composite programs. We again use the symbol $\surd$ to denote the empty program remainder; this results if there are no more statements left to be executed.

The first transition rule we give is for doing nothing, that is, execution of skip.

$$\frac{}{\langle \mu, \delta, \iota, \mathsf{skip} \rangle \xrightarrow{\tau}_{\emptyset} \langle \mu, \delta, \iota, \surd \rangle}$$

As expected, skip doesn't affect anything, except the program to be done next.

We continue with tests of the belief base and actions, respectively. Testing generally yields values for the free variables in the formula tested, that is, substitutions. For example, if the belief base contains the formula Birthday(Wieke, 12-12), then performing the test ?Birthday(Wieke, d) yields the value 12-12 for the variable d. The only difference between the two test statements is the domain of the substitution yielded. As there can occur global variables in formulas tested on constraints of actions, and bindings to global variables are undesirable, we exclude these from the domain of the substitution.

Let θ be a ground substitution such that $dom(\theta) = free(\varphi)$.

$$\frac{\delta \vdash \varphi\theta}{\langle \mu, \delta, \iota, ?\varphi \rangle \xrightarrow{\tau}_{\theta} \langle \mu, \delta, \iota, \surd \rangle}$$

Let θ be a ground substitution such that $dom(\theta) = free(\varphi) \setminus \mathcal{GV}$.

$$\frac{\mu(\mathsf{a}) \vdash \varphi\theta}{\langle \mu, \delta, \iota, ?(\varphi, \mathsf{a}) \rangle \xrightarrow{\tau}_{\theta} \langle \mu, \delta, \iota, \surd \rangle}$$

So, if tests are successful, they always yields ground terms for all free local variables in the tested formula. This doesn't have to mean that the belief base or constraint store uniquely determines a value for each variable. In case this is not so, there are multiple possible outcomes (substitutions yielded) for the test.

In the transition resulting from the transition rule, the obtained values are stored in the substitution attached to the arrow. In the transition rules for the composite programs, these values will be substituted throughout the rest of the program. In case no suitable substitution can be found, there is no transition generated; the test fails.

Inserting something into the belief base causes a *belief revision*. As in the previous section, we abstract from the belief revision process, by supposing a belief revision function $\rho : \wp(\mathcal{L}) \times \mathcal{L} \to \wp(\wp(\mathcal{L}))$. This function takes the old belief base and a formula to be inserted, and yields the set of new belief bases that could result from belief revision. One of these is non-deterministically chosen. Then, this is the transition rule for insertion into the belief base:

$$\frac{\delta' \in \rho(\delta, \varphi)}{\langle \mu, \delta, \iota, !\varphi \rangle \xrightarrow{\tau}_{\emptyset} \langle \mu, \delta', \iota, \surd \rangle}$$

The condition of this transition rule formalises the process of belief revision. As insertion of a belief formula doesn't yield new values for variables, the substitution yielded is the empty one. Note that we stipulated that free variables occurring in formulas inserted into the belief base are guarded. This means that when these insertions are executed, the free variables have been replaced by ground values from the domain $\mathcal{D}$. This is necessary as the belief base is defined to be a set of closed formulas.

Next are the CommGroup-statements, CommGroupAdd and CommGroupReset. Locally the resulting set of constraints cannot be found, as this is also determined by the other communicators. Therefore, the update of bindings to actions takes place globally. The local transition is a dummy transition, which has the purpose to deliver information on the particulars of the CommGroup-statement, in the label of the transition arrow, to prepare a transition step of the whole system.

$$\frac{}{\langle \mu, \delta, \iota, \mathsf{CommGroupReset}(\mathsf{a}, \varphi) \rangle \xrightarrow{\mathsf{a}::\varphi}_{\emptyset} \langle \mu, \delta, \iota, \surd \rangle}$$

$$\langle \mu, \delta, \iota, \mathsf{CommGroupAdd}(\mathsf{a}, \varphi)\rangle \xrightarrow{\mathsf{a}::\mu(\mathsf{a})\wedge\varphi}_{\emptyset} \langle \mu, \delta, \iota, \surd\rangle$$

As these rules yield dummy transitions, the resulting configuration is not computed locally. So, the resulting configuration $\langle \mu, \delta, \iota, \surd\rangle$ at the right hand sides of the transition arrows isn't the actual local configuration resulting from group communication. More specifically, in the rules above the constraint store component doesn't seem to be affected by the CommGroup, while we know that group communication updates the constraint stores. This update is computed globally, and overwrites the unchanged constraint store function μ. The only relevant information in the resulting configurations is the program component, which will be built up by a bottom-up application of the local transition rules.

The label $\mathsf{a} :: \psi$ above the transition reads "I propose to do a under the constraint ψ." In case of a CommGroupAdd-statement, ψ is the conjunction of the stored constraint on a and φ, meaning that the agent strengthens its present constraint with φ. In case of a CommGroupReset-statement, ψ is φ, meaning that the agent overwrites its stored constraints and offers the fresh proposal φ.

Note that group communication doesn't yield any bindings to local variables; the resulting substitution is $\emptyset$. The only thing communicated are constraints on global variables; local variables are not involved in this form of communication.

Group action execution is influenced by all group members, so local semantics also yields a dummy transition labelled with action details, which serves to prepare a global transition step. Action execution can be done individually or groupwise. There is no real difference between these two options. Individual actions simply are group actions where the group only has one member. Locally, an agent tries to execute $\mathsf{a}(\bar{t})$. The terms $\bar{t}$, which are the actual parameters of the action a, may contain free variables, indicating that the agent hasn't chosen a specific value for some formal parameters. In the global semantics, a substitution for these free variables is generated. Also globally, the local belief base is updated to reflect changes in information on the state of the world after the action has been done. Now, this is the local transition rule:

$$\langle \mu, \delta, \iota, \mathsf{a}(\bar{t})\rangle \xrightarrow{\mathsf{a}(\bar{t})}_{\emptyset} \langle \mu, \delta, \iota, \surd\rangle$$

We use the label $\mathsf{a}(\bar{t})$, which indicates that the agent intends to perform a with actual parameters $\bar{t}$. Again, the fact that in the resulting configuration the belief base hasn't changed doesn't mean anything, as the change to the beliefs is computed globally.

Now, we arrive at the program constructors. We employ the convention that $\surd; \pi$ equals π.

First, we define the semantics of the if–then–else-statements. These come in two variants, one which tests the belief base and one which tests the constraint on an action. The semantics of both variants is similar. There are two transition rules for each variant, one for a succeeding test and one for a failing test.

We start with if–then–else-statements that test the belief base. In the following transition rules, let Θ be the set of ground substitutions θ such that $dom(\theta) = free(\varphi)$.

$$\frac{\theta \in \Theta, \delta \vdash \varphi\theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\theta} \langle \mu, \delta, \iota, \pi_1 \rangle}$$

$$\frac{\nexists \theta \in \Theta : \delta \vdash \varphi\theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\emptyset} \langle \mu, \delta, \iota, \pi_2 \rangle}$$

Note that if the test fails, there is no substitution to propagate. These rules give the semantics of the test in the if–then–else statement, and the choice made between the two programs. It isn't surprising that these transition rules resemble those of $?\varphi$ and $?(\varphi, \mathsf{a})$.

For testing the constraint on a, we have the following two rules, in which Θ is the set of ground substitutions θ such that $dom(\theta) = free(\varphi) \setminus \mathcal{GV}$.

$$\frac{\theta \in \Theta, \mu(\mathsf{a}) \vdash \varphi\theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ for a then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\theta} \langle \mu, \delta, \iota, \pi_1 \rangle}$$

$$\frac{\nexists \theta \in \Theta : \mu(\mathsf{a}) \vdash \varphi\theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ for a then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\emptyset} \langle \mu, \delta, \iota, \pi_2 \rangle}$$

For sequential composition, the rule is entirely conventional, except for *substitution propagation*. A substitution resulting from testing or communicating must be applied to the remainder of the program left to be executed. If we have $\pi_1; \pi_2$ and executing the first statement of π_1 yields a substitution θ and a remaining program π_1', then this substitution has to be applied to π_2 when the sequential composition is executed.

$$\frac{\langle \mu, \delta, \iota, \pi_1 \rangle \xrightarrow{l}_{\theta} \langle \mu', \delta', \iota, \pi_1' \rangle}{\langle \mu, \delta, \iota, \pi_1; \pi_2 \rangle \xrightarrow{l}_{\theta} \langle \mu', \delta', \iota, \pi_1'; \pi_2\theta \rangle}$$

Note that the substitution isn't applied to π_1' in this rule. This is not necessary, because the transition in the antecedent of the rule takes care of this. The transition in the consequent of the rule still carries the substitution θ, as there might be other parts of the program not mentioned in the rule, to which the substitution still has to be applied.

Next are the rules for non-deterministic choice.

$$\frac{\langle \mu, \delta, \iota, \pi_1 \rangle \xrightarrow{l}_{\theta} \langle \mu', \delta', \iota, \pi_1' \rangle}{\langle \mu, \delta, \iota, \pi_1 + \pi_2 \rangle \xrightarrow{l}_{\theta} \langle \mu', \delta', \iota, \pi_1' \rangle}$$

$$\frac{\langle \mu, \delta, \iota, \pi_2 \rangle \xrightarrow{l}_{\theta} \langle \mu', \delta', \iota, \pi_2' \rangle}{(\sigma, \delta, \iota, \pi_1 + \pi_2) \xrightarrow{l}_{\theta} (\sigma', \delta', \iota, \pi_2')}$$

Again, the substitution has already been processed on the non-deterministic alternative chosen, and it only needs to be propagated.

6.3 Global semantics

To obtain the semantics of a multi-agent program, we use an interleaving semantics with a handshaking mechanism for synchronisation. Group communication and group action need to shake hands; the other basic statements are interleaved. All local transitions of these statements are labelled with the symbol τ. So, let $\iota \in \mathcal{I}$ be some agent taking a local execution step, let $A_\kappa = \langle \mu_\kappa, \delta_\kappa, \kappa, \pi_\kappa \rangle$ and $A'_\kappa = \langle \mu'_\kappa, \delta'_\kappa, \kappa, \pi'_\kappa \rangle$, where κ is some element of $\mathcal{I}$.

$$\frac{A_\iota \xrightarrow{\tau}_\theta A'_\iota}{\{A_\kappa | \kappa \in \mathcal{I}\} \longrightarrow \{A'_\iota\} \cup \{A_\kappa | \kappa \in \mathcal{I} \setminus \{\iota\}\}}$$

One agent program is executed for one transition step, while the configurations of the other agents don't change.

6.3.1 Group communication

We now continue with group communication. When a group communicates about an action, the local constraints of the agents communicating are updated to be the conjunction of the proposed constraints. This might be an inconsistent formula, if the agents have conflicting interests. The resulting constraints can *control* the participation of agents in the *future execution* of the action communicated about, but we also want to *control the group of agents that takes part in the group communication itself*. If, for example, one or more agents demand that agent Jane should be excluded from the *group doing* the action discussed, it could be useful to forbid this agent to be part of the *group negotiating* the constraints on the execution of the action. So, we demand that agents may only participate in communication about the details of a future group action as long as presence of these agents is not inconsistent with the demands of each agent on the group composition. If the group communicating violates the demands of (at least) one group member, then the group communication fails, in the sense that no global transition is generated. Of course, the demands on the group composition must also hold when the action is to be executed by some group.

We need only one transition rule to cater for both CommGroupAdd and CommGroupReset statements, as the only difference between them is the proposed constraint (either a strengthening of the current constraint or a completely fresh proposal). We dealt with this locally; the label of the local transition contains the constraint proposed. Now, this is the transition rule for group communication:

Let $J \subseteq \mathcal{I}$ be a set of agent names and let $A_\iota = \langle \mu_\iota, \delta_\iota, \iota, \pi_\iota \rangle$.

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\mathsf{a}::\psi_\iota}_\emptyset \langle \mu_\iota, \delta_\iota, \iota, \pi'_\iota \rangle}{\{A_\iota | \iota \in \mathcal{I}\} \longrightarrow \{\langle \mu'_\iota, \delta_\iota, \iota, \pi'_\iota \rangle | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}}$$

where $\mu'_\iota = \mu_\iota[\bigwedge_{\iota \in J} \psi_\iota / \mathsf{a}]$ and the following condition holds:

$$\text{for each } \iota \in J \text{ it holds that } \quad g = J \ \wedge \ \psi_\iota \ \not\vdash \ \bot$$

The antecedent of this transition rule consists of (dummy) local transitions for group communication. The labels above the arrows have to match on a, the action

discussed. Each agent ι brings its own constraint, ψ_ι. As a result, a is constrained by $\bigwedge_{\iota \in J} \psi_\iota$ in the new constraint stores μ'_ι. Note that this globally computed update of the constraint stores overwrites the resulting constraint stores from the dummy local transitions (which are still μ_ι, like in the configurations before execution of the group communication). The group communication takes place in a synchronised execution step. In this respect, CommGroup execution is similar to the synchronous communication primitives in (for example) CSP [20]. For the agents not participating in the communicative action, the local state stays the same.

The condition that for each $\iota \in J$ it holds that $g = J \wedge \psi_\iota \not\vdash \bot$ controls the composition of the group of agents communicating about the action. The formula $g = J$, stating that g is the group of communicating agents, should be consistent with the constraints of the agents. If the constraint of one of the communicating agents implies that there have to be at least three agents involved in the action, then each group communicating about this action must also contain at least three agents. As another example, if the constraint of one the agents (say ι_1) implies that agents ι_3 and ι_6 have to participate in the group action, then group communication with ι_1 in the group can't succeed if these agents don't put in a word.

Note that the transition rule and the associated condition *don't* demand that the constraints of the communicating agents are consistent, that is, that $\bot$ can't be inferred from $\bigwedge_{\iota \in J} \psi_\iota$. As long as the group of communicators is consistent with the constraints of the agents, the group communication succeeds. In case the agents have conflicting demands on action details, the constraint on a will become $\bot$ in the constraint stores of the agents. The reason for this choice, which may seem strange, is that $\bot$ in a constraint store has a signalling function. The agent can test its constraint store, and when it turns out that the constraint is $\bot$, the agent knows that there was disagreement in an earlier negotiation round. Subsequently, the agent can try to re-initiate the communication about the group action by performing a CommGroupReset, which will remove $\bot$ from the constraint store if the constraints of the agents communicating in this new negotiation round aren't inconsistent. It is worthwhile to observe that an agent cannot execute a CommGroupAdd statement if its own corresponding constraint store is inconsistent. In such a case it can only participate in the negotation process by a CommGroupReset statement.

We also like to point out the highly dynamic nature of our model of the negotiation process which allows for the run-time formation of different groups of agents participating to the negotiation. Moreover, each agent has its own local view of the negotiation which will be updated by the execution of the CommGroup statements. As an example, an agent whose local constraint store is inconsistent may return to an earlier stage of the negotiation by the execution of a CommGroupReset statement which involves communication with agents whose local stores record such an earlier stage.

It is important to note that group communication essentially is non-deterministic. The resulting constraint store is fixed, being the conjunction of the constraints of the communicators, so this is not the non-deterministic element. But the group of agents communicating allows many possibilities, as long as the constraints are not too strong. It is possible to limit this kind of non-determinism by adding another demand to the transition rule for group communication. For instance, we could demand the communicating group to be maximal or minimal with respect to set inclusion.

6.3.2 Group action execution

The transition rule for action execution poses a dilemma. We have to decide in which way actual parameters of an action can be determined when the agents executing the action are not specific about them.

Example 3 (Jogging Agents, ctd) We revisit agents James and Clare, that have negotiated about jogging in the park together. They have agreed upon constraints on these parameters, so their constraint stores associated with Jog are the same. They contain the constraint: $g = \{\text{James}, \text{Clare}\} \wedge v_1 = 19.00 \wedge 8 \leq v_2 \leq 10$. These constraints are not decisive; there are still three possibilities for v_2 (assuming the value is a natural number). So, what happens when the agent try to jog together? There are a number of cases:

(a) James and Clare both try to execute Jog(19.00, 8). So, their action parameters agree and satisfy the constraints. The action will take place.
(b) James tries Jog(x_1, 8) and Clare tries Jog(19.00, y_2), where both x_1 and y_2 are free (local) variables. This implements execution time communication: James tells Clare that they will run 8 km., and Clare communicates that they will start at 19.00. All parameters are determined and satisfy the constraints, so the action Jog(19.00, 8) will be jointly executed.
(c) James tries Jog(x_1, 8) and Clare doesn't feel like making any decisions, so she tries Jog(y_1, y_2). The outcome of this is not immediately clear. It is clear that James communicates the distance parameter to Clare. The first parameter is not talked about, and seen mathematically, this is not necessary either, as the value of v_1 is fixed by the constraints agreed upon. So, one choice is to let the above group action succeed; Jog(19.00, 8) is synchronously executed. The other choice is to let this group action fail, because there is no run-time agreement on all action parameters. There is something to say for both options.
(d) James tries Jog(19.00, x_2) and Clare tries Jog(19.00, y_2). It seems intuitively justified that this group action has to fail, as there is no clarity about the distance to be jogged, neither in the action parameters of the agents nor in the constraints. But another view is that the value space for the second parameter still contains three possible values, and therefore there are three possible executions of the above statement, in which James and Clare jog 8, 9 or 10 kilometres, respectively.

Thus, it is not immediately clear what is *the* right semantics for group action execution. In this paper, we create three alternative semantics. Before presenting these options, we will pinpoint the subtle semantical issues at hand.

In group formation and group action, there are two types of communication. There is communication *during the negotiation phase*, performed through one or more CommGroup-statements. Also, there is *execution time communication*, which takes place if some agents participating in the actual group action don't instantiate all action parameters with ground values, but use free local variables for these. Typically, *during the negotiation phase* each agent will make sure all its important demands on the action are incorporated into the set of constraints agreed upon. An agent might be willing to drop some of its less important demands, if this is the only way to form a group, but it will usually hold on to its major constraints. After group formation, the constraint stores of the agents need not be decisive on each action

parameter. Often, there will still be a *parameter space* from which each choice is perfectly acceptable to all agents involved. The choice from this parameter space can be made *during the execution phase*. An agent can pick a value for an action parameter from the set of values allowed by the constraint on the action, and use this value for the parameter.

The combination of execution time communication and negotiation phase communication makes matters opaque here. For ease of formulation, we introduce a new term. We call a formal parameter of an action a *definite for* a group of agents J if the conjunction of the constraints associated with a by the agents in J allows only one value for that parameter. This is the formal definition:

Definition 8 (Definite action parameters) Let $J \subseteq \mathcal{I}$ be a set of agent identities, and let $\langle \mu_\iota, \delta_\iota, \iota, \pi_\iota \rangle$ be the local configuration of agent ι. A formal parameter v_k of an action a is *definite* for J if there exists a value $d \in \mathcal{D}$, such that $\bigwedge_{\iota \in J} \mu_\iota(\mathsf{a}) \not\vdash \bot$ and $\bigwedge_{\iota \in J} \mu_\iota(\mathsf{a}) \vdash v_k = d$.

We will sometimes be sloppy, and call action parameters definite without referring to an agent group. In the above definition, $\bigwedge_{\iota \in J} \mu_\iota(\mathsf{a})$ is the joint constraint of the agents in J on action a. If this constraint implies that the formal parameter v_k must have a certain ground value d, then this parameter is definite. We need to exclude the case that the joint constraint is inconsistent, as then everything can be derived from it, and the constraint thus determines nothing.

In (c) of the jogging example, the essential action parameter is the first one, the time James and Clare will go jogging. This parameter *is definite* for the agents; their constraint stores fix the time at 19.00. So, group communication about the constraints on Jog has settled on a value. Now we focus solely on execution time communication. During execution time communication, both agents use a free variable for the first parameter, thereby indicating that they are willing to let the other agent determine the value of the first parameter. But as none of the agents supplies a value, execution time communication alone can't fix the first parameter. Still focusing only on execution time communication, we can choose between two possible scenarios: the communication could succeed where the value is non-deterministically chosen from the domain of the first parameter, or the communication, and thereby the overall group action, could fail. In the former case, the constraints of both agents will single out the only possible group action parameter.

In (d) of the same example, the second parameter is the essential one. This parameter *is not definite*; the constraints still allow three possible values. This is the only difference between (c) and (d). Both agents again use a free variable for the parameter in their action call, so run-time communication could fail or yield success with a non-deterministic outcome. In case we choose the second option, the range of possible values for the second parameter will be considerably narrowed by the constraint stores, yielding three possible action executions.

So it seems that there are two options for the semantics of group action. In the first, execution time communication doesn't have to be conclusive to yield action success. If it isn't, then actual action parameters are chosen in a non-deterministic manner. In the second option, we demand run-time communication to be conclusive, and the action will fail if this isn't the case. There is still a third option, which lies in between these two alternatives.

Although there are three options for the semantics, we don't need three different transition rules. The difference will be made in the subtle choices in the conditions

associated with the rule. First, we will present the transition rule with the most permissive conditions, formalising the semantics where run-time communication need not be conclusive.

Group action execution has a synchronised semantics, just like group communication. Action execution can be done individually or group-wise. Individual actions simply are group actions where the group has only one member. There are two conditions associated with action execution. First, the actual action parameters of all agents in the group have to be compatible. We model this by requiring the parameters to be *unifiable*. So, if for example one agent in the group tries the value 5 for the first action parameter, and another group member tries 6, then no successful group action is possible. Secondly, the constraints of all agents in the group on the action have to be obeyed. This is the *monitoring* of the action execution; the group action has to be executed according to the agreements made by the agents prior to the action.

Action execution might change the belief base of the agent. We assume that for each agent ι we have a belief revision function ξ_ι, which models this change. This function takes an action with definite parameters and the old belief base, and returns the set of belief bases that could result from a revision of the belief base with the effects of the action. One of these candidates is non-deterministically chosen. By introducing the function ξ_ι, we abstract from details of belief revision; again, see [12] for details.

For the global transition for action synchronisation, we use the same conventions as in the previous transition rule. So, let $J \subseteq \mathcal{I}$ be a set of agent names, let $A_\iota = \langle \mu_\iota, \delta_\iota, \iota, \pi_\iota \rangle$, let θ be a ground substitution and let a be an action with formal parameters $\bar{v}$.

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\mathsf{a}(\bar{t}_\iota)}_{\emptyset} \langle \mu_\iota, \delta_\iota, \iota, \pi'_\iota \rangle}{\{A_\iota | \iota \in \mathcal{I}\} \longrightarrow \{\langle \mu_\iota, \delta'_\iota, \iota, \pi'_\iota\theta \rangle | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}}$$

where $\delta'_\iota \in \xi_\iota(\mathsf{a}(\bar{t}_\iota\theta), \delta_\iota)$ and the following conditions hold:

- $dom(\theta) = \bigcup_{\iota \in J} free(\bar{t}_\iota)$ and θ is a unifier of $\{\bar{t}_\iota | \iota \in J\}$
- $\bigwedge_{\iota \in J} \mu_\iota(\mathsf{a}) \ \wedge \ \bar{v} = \bar{c} \ \wedge \ g = J \ \not\vdash \ \bot$, where $\bar{c} = \bar{t}_\iota\theta$ for any $\iota \in J$

The first condition above demands that the substitution θ provides a ground value for the free variables in the actual action parameters of all agents. Moreover, it states that θ has to be a *unifier* of the actual parameter settings of all agents in the group; this means that $\bar{t}_\iota\theta = \bar{t}_\kappa\theta$ for each $\iota, \kappa \in J$. This explains why the definition of $\bar{c}$ (a shorthand for the unified actual parameters) in the second condition ends with 'for any $\iota \in J$'. The substitution θ instantiates all free variables in the action parameters of the agents in the group in such a way, that all agents use the same set of actual parameters. Only this way, the action can be done in a coordinated manner.

The action with the unified actual parameters constitutes an input to the belief update function. Also, the substitution θ is applied to the programs left to be executed, because in general action execution generates bindings to free variables which have to be passed on to the remainders of the programs. Both these updates overwrite elements in the resulting configurations of the dummy local transitions.

The second condition states that the actual values of the parameters of the group action and the composition of the group have to be consistent with the constraints of

all participating agents. We explain this formula in detail. The formula $\bigwedge_{\iota \in J} \mu_\iota(\mathsf{a})$ is the result of adding up the constraints that the participating agents associated with a. To this formula, two conjuncts are added. The first formula, $\bar{v} = \bar{c}$, states the unified actual values of the formal parameters. The second formula, $g = J$, states the actual composition of the group about to perform the action. If there is a clash between the constraints of the agents and the actual action parameters, then an inconsistency can be derived, resulting in failure of the execution.

In an implementation of the language, a constraint solver can check consistency. Sometimes, a group action execution allows more than one possible unifier θ. In this semantic variant, we leave this non-determinism unresolved; alternatively, it could give rise to failure, as is the case in the semantic variant we will present later on.

Example 4 (Jogging agents, ctd) Going back to the jogging example, this choice of conditions implements the case that both the (*c*) and (*d*) succeed: The constraint on Jog is $g = \{\mathsf{James}, \mathsf{Clare}\} \wedge v_1 = 19.00 \wedge 8 \leq v_2 \leq 10$. We revisit the third and fourth case of the orgininal example:

(*c*) James tries Jog(x_1, 8) and Clare tries Jog(y_1, y_2). As the free variables in the actual parameters are $\{x_1, y_1, y_2\}$, this is the domain of θ. The substitution θ must be a ground substitution, and a unifier of $\{(x_1, 8), (y_1, y_2)\}$. It is clear that $y_2\theta$ must be 8. Furthermore, $x_1\theta = y_1\theta = t$, where t can be any time point according to the first condition of the global transition rule. The second condition demands that the unified actual parameters obey the constraint on Jog, which means that the only legitimate value for t is 19.00. The instantiated action Jog(19.00, 8) is executed.

(*d*) James tries Jog(19.00, x_2) and Clare tries Jog(19.00, y_2). Now, $dom(\theta) = \{x_2, y_2\}$. The ground substitution θ must be a unifier of $\{(19.00, x_2), (19.00, y_2)\}$. In order to unify x_2 and y_2, any distance can be assigned to them, but as the constraint on the distance must be obeyed there are three options: $x_2\theta = y_2\theta = 8$, $x_2\theta = y_2\theta = 9$ and $x_2\theta = y_2\theta = 10$. One of these options is randomly picked, and the Jog action takes place accordingly.

So, with these conditions, run-time communication need not be conclusive. This happens if all agents in the group J use a variable for a particular parameter of a. The substitution θ, being ground, fixes a value for this parameter in an arbitrary manner. The second condition of the transition rule checks this value against the aggregate constraints of the agents. If the parameter is definite (that is, completely determined by the constraints), then the range of values that θ can assign to the parameter is reduced to one possibility. If the parameter is not definite, then multiple substitutions are possible, yielding a non-deterministic action execution.

If we want to implement the option for the semantics where execution time communication has to be conclusive in order to successfully execute the group action, we have to replace the first demand with this one:

- $dom(\theta) = \bigcup_{\iota \in J} free(\bar{t}_\iota)$ and θ is the *most general unifier* of $\{\bar{t}_\iota | \iota \in J\}$

With this demand, there is only one permittable ground substitution θ, namely the most general unifier of the actual action parameters of all agents. In case there are one or more action parameters for which none of the agents determines a domain value, then the most general unifier will not be a ground substitution, and the group

action can't be successful. Only if a value for each action parameter is fixed solely through run-time communication (without looking at the constraints), this variant of the semantics will yield action success.

Example 5 (Jogging agents, ctd) We look at the third and fourth case again:

(*c*) James and Clare try to execute Jog(x_1, 8) and Jog(y_1, y_2), respectively. The most general unifier of these action parameters is $\{x_1 := w, y_1 := w, y_2 := 8\}$, where w is some variable, which isn't a ground substitution. So, this group action won't generate a global transition step.
(*d*) James and Clare attempt Jog(19.00, x_2) and Jog(19.00, y_2), respectively. The most general unifier of the parameter tuples is $\{x_2 := u, y_2 := u\}$, where u is a variable. Again, this isn't a ground substitution, so no global transition results.

There is a third option for the semantics. In this variant, group action execution is successful if each formal action parameter is definite for the group attempting the action or run-time communication is conclusive about its actual value. If this is so, there is only one unifying substitution that yields actual parameters satisfying the constraints of the agents. For this semantical variant, we add an extra condition to the two original conditions, resulting in these conditions:

- $dom(\theta) = \bigcup_{\iota \in J} free(\bar{t}_\iota)$ and θ is a unifier of $\{\bar{t}_\iota | \iota \in J\}$
- $\bigwedge_{\iota \in J} \mu_\iota(\mathsf{a}) \ \wedge \ \bar{v} = \bar{c} \ \wedge \ g = J \ \not\vdash \ \bot$, where $\bar{c} = \bar{t}_\iota\theta$ for any $\iota \in J$
- There is only one ground substitution θ that satisfies the two conditions above.

It might very well be possible to devise more, also interesting, variants of the global semantics for group action.

7 GrAPL: examples

To illustrate the mechanisms of the semantics and show the usefulness of our language, we give two additional examples. The first example is very simple in nature, and we use it to explain the functioning of the two-layered semantics.

7.1 Arranging a meeting

Dr. A and Prof. B are two scientists who would like to cooperate with each other. So, they want to make an appointment for a meeting which lasts a day. They communicate with each other about the action Meet. This action has one explicit parameter, the date of the meeting. These are the programs they employ:

Program of A:
CommGroupReset($g = \{\mathsf{A}, \mathsf{B}\} \wedge$
$(v_1 = 26\text{–}1 \vee v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1)$, Meet);
if $v_1 = x$ for Meet
then Meet(x)
else CommGroupAdd($v_1 = 26\text{–}1 \vee v_1 = 29\text{–}1$, Meet); Meet($y$)

Program of B:
CommGroupAdd($g = \{\mathsf{A}, \mathsf{B}\} \wedge$
$(v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1 \vee v_1 = 31\text{–}1)$, Meet);
if $v_1 = w$ for Meet
then Meet(w)
else CommGroupAdd($\top$, Meet); Meet(z)

Both A and B have the demand that the group which meets has to consist of the two of them. Also, they both initially propose three possible dates for the meeting. Note that two of these dates are possible to both scientists. They communicate with each other, which succeeds because the group communicating {A, B} satisfies the constraints of both agents. If another agent would have tried to join in the negotiation, this would not have succeeded. A uses a CommGroupReset statement and B uses a CommGroupAdd statement in the first group communication. As we assume that the constraint on Meet for both agents is $\top$ before execution of the program, it doesn't matter which of both CommGroup statements the agents use in the first communication round, as the resulting demand proposed turns out the same ($\varphi = \top \wedge \varphi$). The updated constraint stores of both agents then contain the formula $g = \{\mathsf{A}, \mathsf{B}\} \wedge (v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1)$ as the new constraint on Meet. Then, both agents test whether their constraints fix one date for the action parameter, using the test in the if–then–else statement (we explain this test later on). If this would be so, they would meet at this date. But in this case, there are still two dates possible. So, B leaves it up to A to strengthen her preferences, as he adds the empty constraint $\top$ to his present constraint. A then narrows down her possible dates to two, and the scientists communicate for the second time. Now, their constraint stores allow only one date, which is 29–1. So, they meet then.

We now look at how the interplay of local and global semantics yields a trace of this system. Each global trace represents an actual computation of the system. Local transitions play an auxiliary role in the construction of global traces. To build a system trace, we alternately apply local and global transition rules. Local transitions of individual actions are interleaved in the global trace, and dummy local transitions for group actions lead to one synchronised global transition.

We start with the local semantics of the first CommGroup statements in both programs. The local transition rule for CommGroupReset yields a dummy transition for A, with labelled arrow $\xrightarrow{\mathsf{Meet}::\varphi_1} \emptyset$, where $\varphi_1 \equiv g = \{\mathsf{A}, \mathsf{B}\} \wedge (v_1 = 26\text{–}1 \vee v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1)$. Similarly, the local transition rule for CommGroupAdd yields a dummy local transition for B, with labelled arrow $\xrightarrow{\mathsf{Meet}::\psi_1} \emptyset$, where $\psi_1 \equiv g = \{\mathsf{A}, \mathsf{B}\} \wedge (v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1 \vee v_1 = 31\text{–}1)$. The global transition rule for group communication checks whether the group of communicators satisfies the demands of both agents, and as this is the case, updates the constraints of both agents on Meet to be the conjunction of the proposals, which is (equivalent to) $g = \{\mathsf{A}, \mathsf{B}\} \wedge (v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1)$.

Now, both agents locally test their constraint on Meet, as part of the if–then–else statements. We focus at the test of A; the test of B is analogous. The formula tested is $v_1 = x$, where x is a free local variable. We look at the local transition rule for if–then–else statements that test the constraint stores. If μ is the present constraint store function of A, then a ground substitution θ is sought with

$dom(\theta) = free(v_1 = x) \setminus \mathcal{GV} = \{v_1, x\} \setminus \mathcal{GV} = \{x\}$, such that $(v_1 = x)\theta$ follows from the constraint $\mu(\mathsf{Meet})$. As the current constraint on Meet still allows two values for v_1 ($v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1$), such a substitution can't be found. The test for definiteness thus fails, and the second transition rule for the if–then–else statement is used, which results in A taking the else branch. In case the constraint would have been $v_1 = 29\text{–}1$, then the test would have succeeded, yielding the substitution $\{x := 29\text{–}1\}$. This substitution then would have been applied to the rest of the program of A, resulting next in an execution of Meet(29–1). Like A, B also tests the constraint on Meet and takes the else branch. As these tests are individual actions, the global semantics interleaves them.

Both agents again arrive at a group communication statement, a CommGroupAdd in both cases. B chooses not to strengthen the current constraint on Meet; he adds $\top$, resulting in a transition $\xrightarrow{\mathsf{Meet}::\psi_2}_{\emptyset}$, where $\psi_2 \equiv g = \{\mathsf{A}, \mathsf{B}\} \wedge (v_1 = 29\text{–}1 \vee v_1 = 30\text{–}1)$. A adds the constraint $v_1 = 26\text{–}1 \vee v_1 = 29\text{–}1$, resulting in a dummy local transition with arrow $\xrightarrow{\mathsf{Meet}::\varphi_2}_{\emptyset}$, where $\varphi_2 \equiv g = \{\mathsf{A}, \mathsf{B}\} \wedge v_1 = 29\text{–}1$. The global transition rule combines these dummy local steps into a synchronous global step, in which the constraint on Meet of both agents is updated to be φ_2. Now, the date has been agreed upon.

Finally, both agents try a Meet-action, where the actual parameter of A is the free local variable y and the actual parameter of B is the free local variable z. The local transition rule for action execution generates two dummy local transition, with arrows $\xrightarrow{\mathsf{Meet}(y)}_{\emptyset}$ and $\xrightarrow{\mathsf{Meet}(z)}_{\emptyset}$, respectively. The global transition rule will find a substitution unifying the two free variables y and z. Whether these two local transitions combine into an actual synchronised Meet action depends on the semantic variant employed for the global transition rule for action execution. Recall that we introduced three variants in the previous section. Suppose we use the third variant. Then, according to the first condition, the domain of the ground substitution θ associated with the global transition rule must be the set of free variables in the actual parameters of the actuators, which in this case is the set $\{y, z\}$. This condition also states that θ is a unifier of the actual parameters. As both agents use a free local variable, many unifiers are possible. A unifier of the action parameters is $\{y := d, z := d\}$, where d is any specific date. The second condition of the transition rule for group action execution states that the actual action parameter after substitution has to satisfy the aggregate constraints of both agents. To be specific, if the date substituted for y and z is d, then the second condition in this specific situation is:

$$(g = \{\mathsf{A}, \mathsf{B}\} \wedge v_1 = 29\text{–}1) \wedge (v_1 = d) \quad \wedge (g = \{\mathsf{A}, \mathsf{B}\}) \quad \not\vdash \quad \bot.$$

The only value for d which satisfies this, is 29–1. So, there is only one substitution θ possible, namely the substitution $\{y := 29\text{–}1, z := 29\text{–}1\}$. The third condition of the global transition rule (uniqueness of the unifier) is also satisfied. This means that the agents meet on 29–1.

The Meeting will also take place in the first semantic variant of group action execution; in the second variant, it won't.

The above programs are oversimplified, for expository reasons. For example, if the agents initially have inconsistent demands, then their resulting constraint stores will contain $\bot$ for Meet. But as every formula is a logical consequence of $\bot$, the

test $v_1 = x$ (or $v_1 = w$) would succeed for any value for x (or w, respectively). So, the agents should test for $\perp$ first, in a realistic multi-agent program where the agents have no foreknowledge of the demands the other agent is going to make.

The reader might wonder how the negotiation phase is programmed in a realistic agent system, as in the example above both agents seem to 'know' which communication steps are expected. As agents are supposed to be autonomous, it might seem strange that agents anticipate each others communication moves. To be specific, when the agents A and B start their negotiation phase, they synchronously execute their CommGroup statements, with the action Meet as the action discussed. Only if both agents synchronously perform CommGroups about the same action, the negotiation can start. There isn't an initiator of the conversation; both agents act simultaneously. It all seems a bit too symmetrical to be realistic.

There are several responses to these questions. In the first place, GrAPL is an *abstract* programming language for group coordination. Thus, the statements for communication and action execution are on a high abstraction level. In an implementation of GrAPL, the synchronous primitives are likely to be implemented using a series of asynchronous communication primitives. So, on a lower abstraction level, the agents send messages to and fro according to a lower-level communication protocol which has an initiating agent and isn't symmetrical. The implementation underlying CommGroupAdd and CommGroupReset statements might be like this: the first agent arriving at a CommGroup statement on a certain action a broadcasts its initial demands to all agents that aren't excluded from the group of communicators by its own constraints. If there are agents in this group of potential negotiation partners which are willing to communicate about this action (which means that they also have a CommGroup statement with the action a as second parameter), then these agents all send their constraint on the parameters of a to the initiating agent. This agent waits for incoming messages with demands on a until enough agents have reacted to form a group of communicators for this communication round which satisfies the demands of all members of this group. Then, the initiating agent sends a message to the agents in this communicating group containing the resulting constraint (which is the conjunction of the constraints of the communicators).

The advantage of using the abstract synchronous statements of GrAPL over the asynchronous statements of the underlying communication protocols is that the program code for multi-lateral negotiation and group action execution can be more compact. The agent metaphor has been claimed to provide powerful abstract notions that can aid in the construction of the complex software needed nowadays, and GrAPL is an agent programming language in this spirit.

7.2 Negotiation

Returning to the abstraction level of GrAPL, we go into the use of the coordination primitives of GrAPL to construct program code for negotiation. In general, there are two ways to do this. The programmer can write *negotiation protocols* in GrAPL, or he can choose for *ad hoc* negotiation.

In the second case, it is unclear how the conversation between negotiating agents will take place, so the agent program should contain many branches to take care of the different scenarios. When an agent reaches a CommGroup statement, it tries to

execute it, but if there is no suitable group of communicators which synchronises with the agent, then the agent waits until such a group does arrive (see the global transition rule for group communication; if there is no suitable group of communicators, no global transition is yielded and execution stalls for this branch of the agent program). During the negotiation phase, the agents can test the constraint on the action, to find out whether the negotiation has already converged to definite action parameters or whether the negotiation has led to conflicting demands on the action. Depending on the outcome of these tests, the agents decide whether they will try to communicate again and in which way (weaker or stronger demands), or to stop negotiating and start the execution phase.

In case the agents use negotiation protocols, then the program code is less messy. As all agents are assumed to use the same protocol, each agent knows which synchronisation points to expect, and thus less branching is required. We will show an example of a coordination protocol for a group action next. In order to write down the protocol, we assume GrAPL includes recursive procedures.

We give a negotiation protocol, for groups of benevolent agents discussing a future group action. We first show the protocol, and then explain it:
NegotiationProtocol(a, $\psi(g)$, $\varphi_{min}(\bar{v})$, $\varphi_{sat}(\bar{v})$, $\varphi_{opt}(\bar{v})$) :

CommGroupReset($\psi(g) \wedge \varphi_{min}(\bar{v})$, a);
if $\perp$ for a
then ($\psi'(g) := Adjust(\psi(g))$;
    NegotiationProtocol(a, $\psi'(g)$, $\varphi_{min}(\bar{v})$, $\varphi_{sat}(\bar{v})$, $\varphi_{opt}(\bar{v})$))
else (CommGroupAdd($\varphi_{sat}(\bar{v})$, a);
    if $\perp$ for a
    then CommGroupReset($\psi(g) \wedge \varphi_{min}(\bar{v})$, a)
    else (CommGroupAdd($\varphi_{opt}(\bar{v})$, a);
        if $\perp$ for a
        then CommGroupReset($\psi(g) \wedge \varphi_{min}(\bar{v}) \wedge \varphi_{sat}(\bar{v})$, a)
        else skip));

The negotiation procedure takes five parameters, namely the action discussed (a), a demand on the group composition ($\psi(g)$), the minimal demand of the agent on the formal action parameters $v_1, v_2, \ldots, v_k$, where k is the arity of the action ($\varphi_{min}(\bar{v})$), a demand on the formal parameters that formalises what the agent would like best ($\varphi_{opt}(\bar{v})$) and a demand on these parameters that formalises a satisfactory intermediate solution ($\varphi_{sat}(\bar{v})$). The agent first tries to find a group satisfying its group demand $\psi(g)$, which agrees with its minimal demand on the action parameters. The agent waits until it can communicate with a group satisfying $\psi(g)$. If this group doesn't agree with the minimal demand on the action parameters, then the first test for $\perp$ succeeds, and the agent alters its constraint on the group composition (we don't go into details of the procedure *Adjust*). The recursive call to *NegotiationProtocol* results in an attempt to agree on the minimal parameter demands with a new group. As soon as there is a group that agrees with the minimal demands (and has minimal demands that this agent agrees with), then the agent tries to constrain the action parameters further according to its preferences. It starts carefully, by adding $\varphi_{sat}(\bar{v})$. If the others don't agree with this ($\perp$ for a), then the agent resets the constraint to the previous parameter constraint, on which the agents agreed. As the other agents do the same thing, this group communication will always succeed,

and the protocol is finished. If there is agreement in this round, the agent tries strengthening the constraint on a by adding $\varphi_{opt}(\bar{v})$. The other agents also follow the protocol, so they do the same. If this succeeds, then the protocol is finished; otherwise, the agents reset the constraint to their previous agreement.

Note that we assume that the group of agents negotiating is fixed during execution of one call of *NegotiationProtocol*. This can be achieved by always using a $\psi(g)$ of the form $g = J$, with J a set of agent names.

8 Related work and conclusion

Process algebra was developed in the 80s for describing distributed/concurrent computations. Seminal work on this topic includes Hoare's Communicating Sequential Processes (CSP) [20,5], Milner's Calculus of Communicating Systems (CCS) [22], Hennessy's Algebraic Theory of Processes (ATP) [16], and Bergstra & Klop's Algebra of Communicating Processes (ACP,ACP1) [3].

Structural Operational Semantics (SOS) was introduced by Plotkin and Henessy [17,26,18] to give a structured, rigorous treatment of the operational semantics of a programming language, based on a formal system for deriving transitions. Later this has become the predominant approach to giving operational semantics, particularly so in concurrency semantics (cf. [1]).

Recently process-algebraic techniques and SOS have found their way to AI and agent research in particular. We mention a few examples excluding our own work.

Chen and De Giacomo [6] propose a process-algebraic approach to reasoning about action, which in its turn influenced the work on the programming language ConGolog (see [14], where De Giacomo *et al.* define the semantics of this language ConGolog in an SOS-like style).

Kinny [21] proposes an process-algebraic language to describe the complex behaviour of agent architectures like PRS [15]. Furthermore, an algebraic approach coupled with an SOS-style semantics is employed by Omicini *et al.* to describe particularly the issue of coordination within (multi-) agent systems [29,25].

Directly related to our work is that on the agent programming language AgentSpeak by Bordini *et al.* [24], where an operational semantics is given for the (speech act-based) communication primitives in this language.

In this paper we have illustrated the use of process-algebraic concepts and techniques for dealing with multi-agent systems. In particular, we have shown how blending the algebras CSP and CCP can be fruitfully employed for modeling and programming agent communication and coordination.

After having briefly illustrated the main idea for the relatively simple case of bilateral synchronous communication of information, we turned to the more general case of multilateral agent communication and task coordination.

We proposed a constraint programming language for agents with novel primitives for group communication and group cooperation. The statements for group communication are very expressive and allow groups of agents to negotiate over the conditions of execution of future group actions. These conditions are constraints on the composition of the group of participants and the parameters of the action. Communication is synchronous, allowing the dynamic formation of groups. In a language with only bilateral communication, programming a negotiation phase would be much more involved than in the language introduced here. If agreement is

reached in the negotiation phase, the constraints agreed upon monitor the execution of the action discussed. The agents have to stick to their word; only then, group action can be successful. This way, social cohesion is enforced. Action execution is also synchronous, which is an intuitive manner to implement group action.

Our programming language only provides primitive means for negotiation. More sophisticated negotiation protocols or mechanisms can be programmed in GrAPL.

Many coordination problems in agent systems are about finding a solution on which all agents agree in some solution space; constraint solving is especially apt for this. A successful application of constraint-based approaches in artificial intelligence depends on suitably encoding the problems into constraints. But proving that this is possible for any coordination issue agents could encounter doesn't yield a practical coordination language. As we want to focus at applicability, and constraint programming and solving have proven their practical worth, we believe GrAPL is a significant contribution.

We provided the language with a well-defined operational semantics, built in a similar manner as the languages ACPL and 3APL [19]. This gives the language a clear and unambiguous meaning, and offers a solid basis for building an implementation. Such an implementation involves the development of suitable constraint solvers which is one of the main topics of future research.

Acknowledgements We like to express our gratitude to the editors of this volume and the four anonymous referees for their valuable comments and suggestions to improve this paper.

References

1. de Bakker, J.W., de Vink, E.P.: Control flow semantics. MIT Press, Cambridge, MA (1996)
2. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control **60**, 109–137 (1984)
3. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theoret. Comput. Sci. **37**(1), 77–121 (1985)
4. de Boer, F.S., van Eijk, R.M., van der Hoek, W., Meyer, J.-J.Ch.: 'Fully-Abstract model for the exchange of information in multi-agent systems. Theoretical Computer Science **290**(3), 1753–1773 (2003)
5. Brookes, S.D., Hoare, C.A.R., Roscoe, W.: A theory of communicating sequential processes. J. ACM **31**, 499–560 (1984)
6. Chen, X.J., De Giacomo, G.: Reasoning about nondeterministic and concurrent actions: a process algebra approach. Artif. Intell. **107**(1), 63–98 (1999)
7. van Eijk, R.M.: Programming languages for agent communication. Ph.D. Thesis, Utrecht University (2000)
8. van Eijk, R.M., de Boer, F.S., van der Hoek, W., Meyer, J.-J.Ch.: Process algebra for agent communication: a general semantic approach. In: M.-Ph. Huget (ed.) Communication in Multiagent Systems Agent Communication Languages and Conversation Policies LNCS 2650, pp. 113–128, Springer, Berlin Heidelberg New York (2003)
9. Ferber, J.: Multi-Agent systems. Addison-Wesley, Harlow (1999)
10. Finin, T., McKay, D., Fritzson, R., McEntire, R.: 'KQML: an information and knowledge exchange protocol'. In: K. Fuchi, T. Yokoi (eds.) Knowledge Building and Knowledge Sharing, Ohmsa and IOS Press (1994)
11. FIPA. Foundation for intelligent physical agents. Communicative act library specification. http://www.fipa.org, 2000.
12. Gärdenfors, P., Rott, H.: Belief revision. In: D.M. Gabbay et al. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol 4, pp. 36–132, Clarendon Press, Oxford (1995)

13. Gelernter, D., Carriero, N.: Coordination languages and their significance. Comm. ACM **35**(2), 97–107 (1992)
14. De Giacomo, G., Lespérance, Y., Levesque, H.J., ConGolog : A concurrent programming language based on the situation calculus. Artificial Intelligence **121**, 109–169 (2000)
15. Georgeff, M.P., Ingrand, F.: Decision-Making in an Embedded Reasoning System. Proc. IJCAI-89, pp. 972–978, Detroit, MI (1989)
16. Hennessy, M.: Algebraic theory of processes. MIT Press, Cambridge, MA (1988)
17. Hennessy, M., Plotkin, G.D.: Full abstraction for a simple parallel programming language. In: Proc. MFCS'79, LNCS 74, pp. 108–120, Springer, Berlin Heidelberg New York (1979)
18. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. Wiley, New York (1990)
19. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J.Ch.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Syst. **2**, 357–401 (1999)
20. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Englewood Cliffs, NJ (1985)
21. Kinny, D.: The Ψ calculus: an algebraic agent language. In: J.-J. Ch. Meyer, M. Tambe (eds.) Intelligent Agents VIII LNAI 2333, pp. 32–50, Springer, Berlin Heidelberg New York (2002)
22. Milner, R.: A calculus of communicating systems. LNCS 92, Springer, Berlin (1980)
23. Milner, R.: Operational and algebraic semantics of concurrent processes. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, pp. 1201–1242, Elsevier/ The MIT Press, Amsterdam/Cambridge(Mass.) (1990)
24. Moreira, A.F., Vieira, R., Bordini, R.H.: Operational semantics of speech-act based communication in agentspeak. In: M. d'Inverno, C. Sierra, F. Zambonelli, (eds.) Proc. EUMAS 2003 Oxford (2003)
25. Omicini, A., Ricci, A., Viroli, M.: Formal Specification and Enactment of Security Policies through Agent Coordination Contexts. In: M. d'Inverno, C. Sierra, F. Zambonelli (eds.) Proc. EUMAS 2003, Oxford (2003)
26. Plotkin, G.: A structural approach to operational semantics. Techn. Report DAIMI FN-19, Aarhus University, Aarhus (1981)
27. Saraswat, V.A.: Concurrent constraint programming. The MIT Press, Cambridge, Massachusetts (1993)
28. Tsang, E.P.K.: Foundations of constraint satisfaction. Academic Press, London and San Diego, 1993, ISBN 0-12-701610-4
29. Viroli, M., Omicini, A.: Specifying Agent Observable Behaviour. In: C. Casterfranchi, W.L. Johnson, (eds.) Proc. AAMAS 2002 Bologna, pp. 712–720, ACM Press (2002)
30. de Vries, W.: Agent interaction: abstract approaches to modelling, programming and Verifying Multi-Agent Systems, Ph.D. Thesis, Utrecht University (2002)
31. de Vries, W., de Boer, F.S., Hindriks, K.V., van der Hoek, W., Meyer, J.-J.Ch.: A programming language for coordinating group actions. In: B. Dunin-Kęplicz, E. Nawarecki (eds.) From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS'01), LNAI 2296, pp. 313–321, Springer, Berlin Heidleberg New York (2002)
32. Weiss, G. (ed.): Multiagent systems. The MIT Press, Cambridge, Massachusetts (1999)
33. Wooldridge, M.J.: An introduction to multiagent systems. Wiley, Chichester (2002)
34. Wooldridge, M.J., Jennings, N.R.: Intelligent agents: theory and practice. The Knowl. Eng. Review **10**(2), (1995)